Smart-ML: A System for Machine Learning Model Exploration using Pipeline Graph

Dhaval Patel, AI Applications, IBM Research, NY, USA, pateldha@us.ibm.com
Shrey Shrivastava, AI Applications, IBM Research, NY, USA, shrey@ibm.com
Wesley Gifford, AI Applications, IBM Research, NY, USA, wesley@us.ibm.com
Stuart Siegel, AI Applications, IBM Research, NY, USA, stus@us.ibm.com
Jayant Kalagnanam, AI Applications, IBM Research, NY, USA, jayant@us.ibm.com
Chandra Reddy, AI Applications, IBM Research, NY, USA, creddy@us.ibm.com

2020 IEEE International Conference on Big Data (Big Data) | 978-1-7281-6251-5/20/$31.00 ©2020 IEEE | DOI: 10.1109/BigData50022.2020.9378082

***Abstract*—In this paper, we describe an overarching ML system with a simple programming interface that leverages existing AI and ML frameworks to make the task of model exploration easier. The proposed system introduces a new programming construct namely pipeline graph (a directed acyclic graph) consisting of multiple machine learning operations provided by different ML repositories. End user uses the pipeline graph as a common interface for modeling different ML tasks such as classification, regression, and timeseries prediction, while enabling efficient execution on different environments (Spark, Celery and Cloud). We further annotated the pipeline graph with a hyper-parameter grid and an option to try-out a wide range of optimization strategies (i.e., Random, Bayesian, Bandit, AutoLearn, etc). Given a large pre-defined pipeline graph along with its hyper-parameters, we provided a general-purpose, scalable and efficient pipeline-graph exploration technique to provide the automated solutions to a variety of ML tasks. We compare our automated approach to several state-of-the-art automated AI systems and find that we achieve performance comparable to the best results, while often producing simpler pipelines using off the shelf components. Our evaluation suite consists of experiments on 60+ classifications and regressions datasets.**

I. Introduction

The application of Artificial Intelligence (AI) and Machine Learning (ML) has yielded important gains in many business and social domains including health care, heavy industries, supply chain management, and transportation logistics. These gains have often been hard fought, however. This is because there are many barriers to the use of AI/ML, particularly in small and intermediate sized enterprises. Typically, the task of finding the "best" machine learning model requires many steps including data preparation (e.g., scaling, dimensional reduction, feature engineering), parameter optimization, and model validation and selection. These steps may be spread across multiple libraries. As discussion in [1], these steps are tedious and error-prone and lead to the emergence of brittle "pipeline jungles". Additionally, the need to perform these steps on a range of environments, whether it is an analyst's stand-alone workstation or a vendor provided platform as a service (PaaS) or hybrid cloud offering, further complicates the task. Moreover, many enterprises do not have the in-house data science and/or engineering expertise to support these activities. In order to facilitate the use and resulting benefits of AI/ML at small or medium sized enterprises, there is a need for frameworks that make the modeling process easier for expert and non-expert alike. To that end, we present an automation and abstraction library that leverages existing AI and ML frameworks to make the task of model exploration easier as well as efficient.

A. Motivation

The term "predictive modeling" in the field of data science refers to the process of analysis to discover the best data transformations and modeling forms for ultimately drawing accurate and meaningful inferences from new realizations of (scoring) data. Predictive modeling is of fundamental interest to researchers in machine learning, pattern recognition, databases, statistics, artificial intelligence, knowledge acquisition for expert systems, and data visualization. In the machine learning community, researchers and or data scientists build "Pipeline" to define a series of steps to be performed on an input dataset for building model. As supported by the literature, it is not an easy task to know beforehand what options will work well for a given dataset. As a result, user needs to build and manage many pipelines manually. Thus, a tool that allows data scientists to efficiently explore multiple pipelines is needed.

Fig. 1: An example of pipeline graph for a classification task

To make the model exploration task easier, we introduce a concept of "Pipeline Graph". A pipeline graph defines the nature of and ordering of the operations to perform when exploring predictive models for tasks such as classification, regression, or clustering. It prescribes a series of steps such as feature scaling, feature transformation, feature selection, model learning, etc to find the best solution for given training data instance. In Figure 1, a pipeline graph for classification task is provided for better understanding. In this graph, each layer is composed of multiple options. For example, the first layer in Figure 1, is a feature scaling related operations. Ideally, feature scaling can be done using a standard scaler or 0-1 normalization or even an outlier-aware robust scaler. Similarly, there are multiple machine learning algorithms available for building supervised classification models. A path through the pipeline graph represents a single ML pipeline and end user is interested in identifying the best path (or top-k paths) for a given dataset.

B. Contribution

A pipeline graph by itself wouldn't offer much benefit beyond accounting convenience. To be useful, we couple the pipeline graph with a framework that provides support for the following:

1) A simpler *Define-by-run* programming API to support range of users from expert data scientists to a relative novice. This includes support for constructing a pipeline graph to explore multiple algorithms (including custom ones an expert user writes) as well as out-of-the-box support for different types of parameter optimization (grid search, randomized, Bayesian, Hyperband, derivative free optimization, etc.);
2) An *Easy-to-setup* versatile architecture that can be used to support a scalable execution. The size of pipeline graph can be very high to generate better result. The execution models include parallelization via Apache Spark [2], Celery asynchronous worker queues [3], and support for cloud vendor AI service offerings;
3) A novel heuristic-based search algorithm, namely DAG-AI, to efficiently explore a very large pipeline graph with continuous result updating (in a "leader board" sense) leading to AutoML system;
4) The ability to use multiple machine learning ecosystems and frameworks (e.g., scikit-learn [4], Keras [5], pyearth [6], XGBoost [7]) as part of the same pipeline graph definition providing a seamless AutoBazaar API [1]. In most cases, framework objects such as transformers and estimators can be used without the need to wrap them with special platform-specific code;
5) Out-of-the-box and ready to use discrete and continuous hyperparameter grids for many commonly available algorithms for classification, regression and anomaly detection task;
6) Scripts for running comprehensive benchmarks to compare the framework to other frameworks that seek to make AI/ML automated;
7) We conducted extensive set of experiments on 100+ datasets. The results are summarized in the paper and also stored the raw results for future usage.

II. Pipeline Graph

In this section, we formerly introduce the pipeline graph. The pipeline graph contains multiple end-to-end ML/AI pipelines that are prescriptions for how input data are to be manipulated and ultimately modeled. The prescriptions includes scaling operations, explored models, the definition of hyperparameter (HP) grids and how they are explored (e.g., random-search, grid-search, among others). Figure 1 provides a graphical representation of a pipeline graph created for an auto-classification task. In this example, the user is setting three stages: i) feature scaling, ii) feature selection, and iii) classification. The feature scaling stage contains three popular methods MinMaxScaler, RobustScaler, StandardScaler and an option to exclude the stage (i.e., "noop" meaning no operation). The next stage is feature selection using PCA or SelectKBest or NoOp. The final stage consists of classification models to explore.

A pipeline graph, denoted as $G(V, E)$, is a directed acyclic rooted graph (DAG) with a set V of vertices and a set E of edges. Each vertex $v_i \in V$ represents an operation to be performed on the input data, and an edge $e_i \in E$ represents the ordering of operations between vertices. In Appendix VIII-A, we have given a code snippet for constructing pipeline graph, its parameter DAG and its execution.

A. Pipeline Node

Each vertex v_i in G is referred to as a *pipeline node*. In our design, pipeline nodes consist of a name and the operation it performs and a reference to the python object containing the functional implementation. It is represented by a tuple v_i = $(name_i,\ object_i)$. For example, tuple ("pca", $\mathrm{PCA}()$) represents a pipeline node, with name "pca", that performs principle component analysis. The nodes in Figure 1 are labeled with the node name.

The name given to each node in the pipeline graph should be unique. The node name is a tag that enables a user to supply additional information that can be used to control the behavior of the node. One such example is to specify a parameter value prefixed by the node name adopting the convention used in scikit-learn. For example, "gradientboostingregressor__n_estimators" would associate with the parameter "n_estimators" of the scikit-learn object GradientBoostingRegressor. In Section III-A, we discuss the usefulness of adopting this standard convention for preparing hyperparameter grids for parameter optimization. If node name is not given by user, then it is inferred automatically from $operation_i$.

The operation performed by a pipeline node is either of two types: Transform(_.transform) or Estimate(_.fit). An Estimate operation is typically applied to a collection of data items to produce a trained model. A Transform operation uses trained model on individual data items or on a collection of items to produce a new data item.

B. Pipeline Path

A pipeline path of a pipeline graph $G(V, E)$, denoted as P_i = $\{v_{root} \rightarrow v_1 \rightarrow ... \rightarrow v_j \rightarrow ... \rightarrow v_k\}$, is a directed path of nodes $v_j \in V$ that starts from the root node v_{root} and ends at leaf node v_k. Briefly, a pipeline path chains together AI/ML-related operations common to the overall task of building predictive models. For example, the left most pipeline path of the pipeline graph in Figure 1 is given as:

$$P_1 = \{\text{start} \rightarrow \text{Robust Scaler} \rightarrow \text{PCA} \rightarrow \text{Gaussian NB}\}$$

There are 48 total pipeline paths to explore in Figure 1. This is calculated as follows:

4	(feature scalers on first level)
×3	(feature selectors on second level)
×4	(classifiers on third level)
48	(pipeline paths for exploration)

The above calculation assumes a fixed point in the hyper-parameter space for pipeline node objects (vertices in G). We discuss more exhaustive hyperparameter exploration later.

III. Pipeline Graph Evaluation

The objective of pipeline graph evaluation is to identify a pipeline path P^* from a user-defined pipeline graph that performs best for a given dataset. Normally, each path P_i in a pipeline graph is evaluated for a given dataset D, and the path with the best performance is selected. Formally,

$$P^* = \underset{P_i \in \text{paths}(G)}{\arg\max} \; \Phi(P_i, D) \tag{1}$$

where $\Phi(P_i, D)$ in Equation 1 represents the performance of pipeline path P_i for dataset D. A user can specify existing or custom performance measures as part of a cross validation or training/validation set strategy.

Figure 2 outlines the process of calculating the performance measure for pipeline path P_i using the K-Fold cross validation strategy. This strategy is a widely used technique for IID data. The input dataset D is partitioned into K equally sized folds (see top left corner). Next, the data from $K-1$ folds are used to train the pipeline path P_i and data from a remaining (single) fold is used to obtain prediction using the trained path. The prediction result of each fold is compared with the ground truth to obtain the fold-level performance. The process is repeated for all the other folds one by one, and the average performance is computed as a final performance measure. Some examples of other cross validation strategies are nested K-Fold cross validation, train/test split, timeseries split, etc.

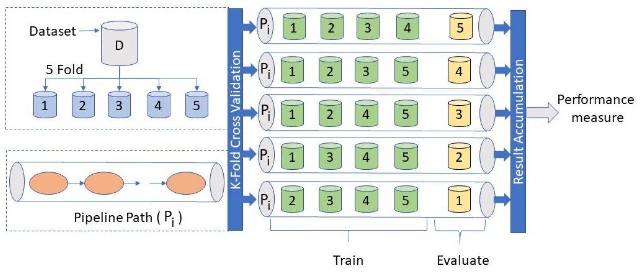


Fig. 2: Cross-Validation Based Pipeline Path Evaluation

To support cross-validation-based performance evaluation, the pipeline path must provide two functions: `train` and `score`. Due to space limitation, we did not discuss internal working of `train` and `score` in detail, but they use Transform and Estimate operations of node as discussed in Section II-A. For the working example given in Figure 1, the total number of pipeline paths for evaluation, using 5-fold cross-validation strategy would be 240:

48	(pipeline paths computed before)
×5	(5-fold cross validation)
240	(pipeline paths for evaluation)

A. *Hyperparameters Grid and its Tuning*

While the pipeline graph provides the aforementioned flexibility to add a sequence of machine learning algorithms of a user's choice, it also contains another important capability. Many machine learning algorithms come with a set of parameters, such as the choice of the kernel function in SVM, the depth of the tree in decision tree classification, the number of projected dimensions in PCA transformation, etc. Such algorithm specific parameters are referred as "hyperparameters" [8].

In our pipeline graph design, the set of hyperparameters associated with a pipeline node (v) are represented by $\mathcal{H}(v)$. Each hyperparameter $\rho \in \mathcal{H}(v)$ is further associated with a set of values $\mathcal{V}(\rho)$, i.e.:

$$\langle v, \rho \rangle \rightarrow \mathcal{V}(\rho) \quad \forall \rho \in \mathcal{H}(v)$$

For example, for the PCA algorithm, we may be interested in setting the parameter for the number of components to value in the set $[3, 5, 7, 10]$.

$$\langle v = \text{PCA}, \rho = \text{n\_components} \rangle \rightarrow [3, 5, 7, 10]$$

This functionality is achieved in our system by making the hyperparameter an attribute of the pipeline node and separately defining mappings which associate an algorithm and hyperparameter to a set of values. Note that while we refer to $\mathcal{V}(\rho)$ as a set of values, it may be a discrete set or continuous values sampled from a range or a particular distribution.

Typically, machine learning algorithms have more than one hyperparameter so users wish to try combinations of parameters for different algorithms within a pipeline path. To support this concept, we define "hyperparameter grids" for many common machine learning tasks grouped by function (e.g., a regression grid, a classification grid, etc.). We also define coarse and fine grids which differ in size and thus computational burden they impose.

The grid of available points in the hyperparameter space for a node v can then be defined as the generalized Cartesian product:

$$\Theta(v) = \prod_{\rho \in \mathcal{H}(v)} \mathcal{V}(\rho)$$

With a slight abuse of notation, we can further represent the hyperparameter grid for a sequence of nodes in a path ($P = \{v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k\}$) as:

$$\Theta(P) = \Theta(v_1) \times \Theta(v_2) \times \cdots \times \Theta(v_k)$$

1) Hyperparameter Tuning: As shown earlier, a pipeline path is initialized with default parameters. In the presence of a hyperparameter grid, the pipeline paths can be initialized with the parameter values present in $\Theta(P)$. For example, consider a following pipeline path:

$$P_i = \{\text{start} \rightarrow \text{Robust Scaler} \rightarrow \text{PCA} \rightarrow \text{SGD Classifier}\}$$

In the absence of a hyperparameter grid, each node in the above path P_i is initialized with its default parameters. However, in the presence of the $\Theta(P)$, there are now 12 (4 options of n_components $\times$ 3 options for SGD classifier penalty) different parameter combinations for pipeline path P_i. The node name is playing an important role in identifying the association between a node in pipeline path and the hyperparameter grid we defined.

We now extend our hyperparameter tuning analysis of a pipeline path to the entire pipeline graph of Figure 1. In our working example, the total number of pipeline paths to be evaluated is 48. Out of these 48 paths, 16 ($4 \times 2 \times 2$) involve a pipeline node with hyperparameters using hyperparameter grid $\Theta(P)$. Utilizing the entire hyperparameter grid to unroll these 16 paths, we generate 105 total pipeline paths. Thus, using $\Theta(P)$ in conjunction with 5-fold cross-validation strategy results in a significantly higher number of path evaluations as shown below:

$$\begin{array}{rr} 32+105 & \text{(pipeline paths)} \\ \times 5 & \text{(5-fold cross validation)} \\ \hline 685 & \text{(pipeline paths for evaluation)} \end{array}$$

Hyperparameter tuning is the process of identifying the best parameter for a given algorithm. It is an active area of research interest as identifying the best parameter from large search space is computationally expensive. In our context, hyperparameter tuning is performed at the pipeline path level (i.e., we try to discover the best parameter value for each pipeline path.) We revise equation 1 to accommodate hyperparameter-tuning-based best path discovery as follows. Let P^*, θ^* be the best pipeline path (as per the evaluation metric) with corresponding hyperparameter value respectively. Then:

$$P^*, \theta^* = \underset{P \in \text{paths}(G),\, \theta \in \Theta(P)}{\arg\max} \Phi(P, D; \theta) \tag{2}$$

where $\Phi(P, D; \theta)$ is the evaluation metric (performance) on data D for pipeline path P when initialized with parameters θ.

IV. Pipeline Execution Engine

The objective of the pipeline execution engine is to discover the best model (i.e., best pipeline path in our case) for an input dataset D. Since the computational workload originating from the execution of pipeline graph is very high, we employ two complementary strategies for speeding up pipeline execution: 1) parallel and, optionally, distributed execution with an option for time-bounded execution of pipeline paths, and 2) an optimization-based hyperparameter tuning approach.

We implemented pipeline execution engine to support various computing infrastructures such as a single node with multiple cores (e.g., a single user's laptop), spark clusters, celery distributed workers, and cloud services to achieve different levels of parallelism. In this paper, we discuss design of execution engines for Apache spark (SparkEngine) in detail. The idea of Celery distributed workers (CeleryEngine) and Machine Learning services on Cloud (MLEngine) follow similar concept.

A. Task-level parallelism

A task is a discrete section of computational work. For example, an execution of a single pipeline path is treated as one task. Our first strategy is to run multiple tasks (i.e., pipeline paths) in a parallel (preferably distributed) and time-bounded manner. Actually, we can create tasks at multiple levels of granularity to achieve different levels of parallelism:

- **Path-level** parallelism involves running each pipeline path in parallel. In this setting, each task evaluates a hyperparameter grid for a given pipeline path and outputs an optimal parameter set for the pipeline path based on the best cross-validation error. We refer this option as a "path_learning".
- **Parameter-level** parallelism involves running each pipeline path for different hyperparameter combinations in parallel. In this setting, each task produces a cross-validation error for a given pipeline-path and a parameter realization in the parameter hyperspace. We refer to this option as "param_learning".

Our execution engine supports parameter and path-level parallelism. The latter is used by default unless a user specifies a hyperparameter grid or when a particular hyperparameter optimization strategy only supports sequential operation (such as those that estimate gradients within the search routine).

B. Hyperparameter optimization

Our second strategy is to use an optimization technique to reduce the size of the hyperparameter search space (compared to, say, fully enumerated grid search) for each pipeline path. In the literature, various hyperparameter optimizations are proposed recently [9]–[11]. These optimizer vary in exploration techniques as well as the form of the input hyperparameter grid. At present, our framework supports six different optimization schemes: Complete search (compS) [12], Random search (randS) [13], Evolutionary search (evolS) [14], Hyperband search (hbandS) [15], Bayesian-optimization search (bayesS) [12], and RBFOpt search (rbfOptS) [16]. A user can select one of the optimizer listed above for discovering the best combination of hyperparameter values and pipeline path.

C. SparkEngine: Spark Execution Engine

Apache Spark is a distributed general-purpose cluster-computing framework [2]. Spark provides a programming model and computing framework for writing an application that can be executed in parallel. At an abstract level, every application in spark starts with a design of a resilient distributed dataset (RDD). Briefly, a RDD is a collection of homogeneous elements (i.e., tasks or data) that are partitioned across different processes of a given node or different nodes for parallel execution. For example, each pipeline path of pipeline graph can be an element of RDD if our goal is to achieve path-level parallelism. Once a RDD is materialized, an execution over the RDD can take place. The RDD execution is a process in which a common operation is applied on each element of RDD. Spark uses a master-worker architecture to execute in a distributed fashion. The master node executes a driver program that prepares an RDD and distributes it across worker nodes on which executors run. The driver program and the executors run in their own processes.

Given a pipeline graph with a hyperparameter grid and data, the spark driver first prepares an RDD named PathRDD. Each element of PathRDD is a tuple as given below:

$$\langle P, \Theta(P) \rangle$$

The number of elements in PathRDD is equal to the number of pipeline paths in input pipeline graph, G. This RDD is used if we want to achieve path-level parallelism.

In the second step, a PathRDD is exploded into Path-ParamRDD to obtain parameter-level parallelism. Each element of Path-ParamRDD is a tuple as given below:

$$\langle P, \theta_i \rangle \quad \theta_i \in \Theta(P)$$

where P refers to a pipeline path, and θ_i refers to one specific set of parameters to be applied to P. Generally, the parameter space is sampled and only a subset of the available points in the space will be evaluated. The Path-ParamRDD allows for a high level of parallelism that can be used for hyperparameter optimization methods. During execution, Path-ParamRDD and PathRDD are equally distributed across the worker nodes of the spark cluster for parallel execution. Briefly, each worker node runs several tasks and each task is allocated a fixed pre-identified portion of the RDD. The input data D is also shipped to all the worker nodes. We use a broadcast variable, a type of shared variable in spark, to cache data D (possibly all in-memory depending on size and available cluster RAM) of all worker nodes. As a result, only one copy of data D is produced on each worker node.

In the third step, we initiate a map operation to obtain a new RDD resultRDD. The map operation is performed on each element of the input RDD. Depending on the level of parallelism, there are two types of map operations that can be performed: a) apply "param_learning" on Path-ParamRDD (See Listing **??**), or b) apply "path_learning" on PathRDD. The end result of each element of resultRDD is a tuple as given below:

$$\text{Path-ParamRDD: } \langle P, \theta_i, \text{best score} \rangle$$
$$\text{PathRDD: } \langle P, \text{best score} \rangle$$

In the fourth and final step we sort results by performance metric. In the case of resultRDD generated from Path-ParamRDD, we apply a reduce operation that aggregates all the elements of the resultRDD using pipeline path specific maximum function and returns the final result to the driver program. In case of resultRDD generated from ParamRDD, we get the a best model for each path.

V. Automatic Learning on Pipeline Graphs

With recent interest in developing automated AI systems [17], such as AutoML [18], DAUB [19], TPOT [20], AutoSklearn [21], and AutoKeras [22], we also extend the core capability of the pipeline execution engine to support automated learning. We refer to this autolearning functionality as DAG-AI. First, an end user needs to prepare a deep pipeline graph for a given ML task. For an entry level data scientist, our system has prebuilt pipeline graphs for classification, regression, imbalanced learning, etc. For example, the pre-built classification graph has depth 5, 130+ nodes, 160k paths and a hyper-parameter grid with $\approx$150 entries. The output of DAG-AI on a pipeline graph is a best performing pipeline path with a parameter configuration for a given dataset. However, the best performing pipeline path may not contain all the nodes along the paths in the graph. In other words, DAG-AI method explores variable length pipeline paths from a given pipeline graph.

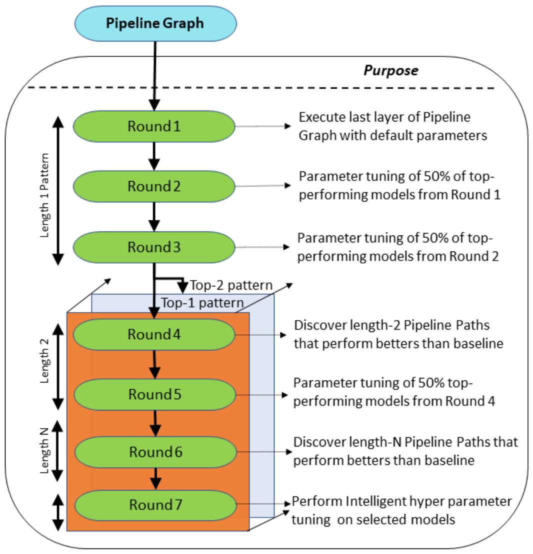
A. *Pattern Mining for Pipeline Graph*

We extend a sequential coverage approach used for mining frequent (utility) patterns to generate a variable length machine learning pipeline from a given pipeline graph. The frequent pattern mining algorithms initially obtain all the frequent length 1 patterns from the input dataset. In the next step, algorithm generates a new candidate pattern by first trying to "extend" an existing pattern or second trying to "enlarge". For example, assume given a pattern $\{\langle a, b \rangle \rightarrow \langle c \rangle\}$, the extend operation adds an additional item in $\langle c \rangle$ such as $\{\langle a, b \rangle \rightarrow \langle c, d \rangle\}$ and enlarge operation adds an additional item after $\langle c \rangle$ such as $\{\langle a, b \rangle \rightarrow \langle c \rangle \rightarrow \langle d \rangle\}$. In our context, the "extend" operation initiates a randomized hyper-parameter search (randS) on a given pattern, whereas an "enlarge" operation add a new node at the start of given pattern. Note that, the newly added node via enlarge operation is reachable from given path on an underlying Pipeline Graph. To the best of our knowledge, the idea of formalizing the mapping of extend and enlarge operation from data mining field to machine learning world is not discussed any where else.

B. *DAG-AI*

The underlying solution designed to implement DAG-AI system aims to discover the best pipeline path with its parameter as early as possible. Discovering reasonable best solution as early as possible eliminates the need of running experiments for longer duration. Our idea is to execute a pipeline-graph iterative over multiple rounds and progressively generates the results at the end of each round. The results generated at the end of current round is available to the user for quick inspection, as well as, used in subsequent iteration to prune the search space.

Figure 3 outlines the DAG-AI execution spread over 7 rounds. In the first round, only the last layer of pipeline graph is executed. The execution is limited to the default parameter and does not involve any hyperparameter tuning. Note that, the last



Pipeline Graph
Purpose
Round1
Execute last layer of Pipeline Graph with default parameters
Round2
Parameter tuning of 50% of top-performing models from Round 1
Round3
Parameter tuning of 50% of top-performing models from Round 2
Length 1 Pattern
Top-2 pattern
Top-1 pattern
Round4
Discover length-2 Pipeline Paths that perform betters than baseline
Round5
Parameter tuning of 50% top-performing models from Round 4
Length 2
Round6
Discover length-N Pipeline Paths that perform betters than baseline
Length N
Round7
Perform Intelligent hyper parameter tuning on selected models

Fig. 3: Automated Learning Methodology

layer of pipeline graph is composed of machine learning models and it is compulsory component of any modeling activity. The execution can use spark or celery or cloud engine for speedup. The model performance obtained at the end of the first round act as an baseline for subsequent operation. We select around 50% of the top-performing models to become a candidate for the second round. The first round result is also available to the user.

In the second round, we initiated a random search based hyperparameter tuning on the models that are selected in previous round. A randomized hyperparameter tuning is highly parallel activity. The number of parameters to be tried out for each models in the current round is adjustable but we keep it a small value such as 10. In early stage of execution, we like to control the exploration search space. In this round, we run nearly 10+ different models with 10 different randomly generated parameter values. Out of 10+ models, we select nearly 50% of the top-performing models to become a candidate for the third round.

In the third round, we yet again initiated a random search based hyperparameter tuning on the models that are selected in previous round. Compared to previous round, the number of parameters to be tried out for each models in this round is more and adjustable. At present, we run nearly 5+ different models with 30 different parameter values in this round. Note that, some model does not have many parameters to be tuned.

After successful completion of first three rounds, we identified length-1 pipeline paths (along with the parameters) that perform betters. To reduce exploration of huge search space, we select k ($\leq$ 5) top-performing models here-after and derive k pipeline-graphs, one for each top-performing model. Each of these new pipeline-graph's last layer has only one node. Let us denote a new pipeline graph for k^{th} top-performing model as PG_k. For example, let us assume that a "RandomForestClassifier" is a top performing algorithm in Figure 1, then the resultant pipeline graph only has one node in the last layer.

In round 4 and 5, we focus on discovering length 2 pipeline paths for each top-performing models. Given a pipeline graph G_k for a k^{th} top-performing model, we decompose G_k into multiple pipeline graphs of depth 2. Note that, the last layer in each decomposed graph is same. Next, we process each decomposed graph in two stages (i.e., round 4 and round 5) to discover a length-2 pipeline paths that perform better than the pipeline path from which it got enlarged. Round 4 is similar to Round 1, where pipelines with default parameter is tried, whereas, round 5 is similar to round 2, where randomized hyper parameter tuning is conducted on the top-performing paths outputted by round 4. After round 4 and round 5, we know what are the nodes other than nodes from last stages also helps to improve the performance. In Round 6, we use these nodes to grow a longer length patterns.

Up until round 6, we use highly parallelized randomized search operations. In round 7, we apply intelligent search mechanisms for promising pipeline paths that have been discovered. In particular, given a path, we apply evolS, hbandS, bayesS, and rbfOptS on each path to discover a hyper parameter tuning that can help to improve the performance. Instead of applying intelligent method on each and every pipeline paths, we apply it on top-performing pipeline-paths to improve the execution time. It has been shown in many places that randomized hyper-parameter tuning indeed achieve comparable results.

VI. Qualitative Evaluation

In this section, we qualitatively evaluate the DAG-AI method.

A. AI System Setup

We consider a variety of AI systems. The detailed configuration is given in Appendix.

CatBoost [23] : We tested two versions of CatBoost algorithms: CatBoost$_{\text{Def}}$ (default parameters only) and CatBoost$_{\text{Opt}}$ (optimization to select parameters).

TPOT. [20] We tested two version of TPOT algorithms, TPOT$_{\text{S}}$ and TPOT$_{\text{L}}$, using default parameters[1] except "n_jobs", "max_time_mins", "early_stop", "generations" and "population_size".

AutoSklearn. [21] We run AutoSklearnRegressor and AutoSklearnClassifier algorithms using default parameters[2] except "time_left_for_this_task". Very recently, **AutoSklearn** (2.0) is available [24].

AutoML. We use python API end points to execute the classification and regression model of AutoML. We use default parameters setting[3] except "max_runtime_secs" and "exclude_algos".

DAG-AI. We have tested two versions of DAG-AI systems: DAG-AI$_{\text{Def}}$ and DAG-AI$_{\text{Stack}}$.

[1]http://epistasislab.github.io/tpot/api/
[2]https://automl.github.io/auto-sklearn/master/api.html
[3]http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.htmlautoml-interfacex‘x

DAUB. We have tested a tool based on DAUB functionality [19] using a default set of parameters for automated discovery of scikit-learn pipelines.

B. Experimental Setup

Figure 4 gives an overview of experimental setup. Detail discussion is given in Appendix.

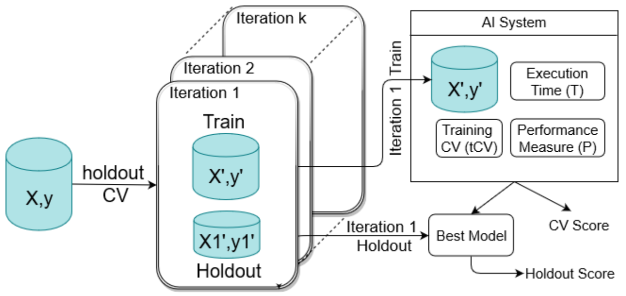


Fig. 4: Experimental setup for qualitative evaluation

All AI systems is running on a dedicated node with 32 cores and 64 GB RAM memory. Although DAG-AI can scale to as many nodes, we provide an equal infrastructure while running experiment. The dataset is collected from pmlb[4] and openML[5].

C. Ranking Based Benchmark Comparison

For the comprehensive comparison of AI systems we conducted extensive experiments: 3000+ binary classification experiments and 1500+ regression experiments. We group experiments based on performance metric and execution time, and then use ranking approach to summarize the outcome at group level. To maintain the continuity, this section focused on classification task.

Our first group is based on the *accuracy* performance metric and a *10 minute* execution time limit. Since the majority of experiments from CatBoost$_{\mathrm{Def}}$ and DAUB$_{\mathrm{E}}$ also finish around 10 minutes or much before that, we also included these systems in the analysis. Next, we obtain rank (lower is better) of each AI System based on the holdout score of the outer level cross-validation. Note that comparing AI systems using training CV score is not advisable, as we do not have complete control on the internal CV and the selected model may overfit the training data. All together, we obtained a total of 351 rankings spread across 47 datasets. Note that we have eliminated several datasets due to extreme class imbalance issues. Figure 5 shows a box plot of ranks obtained by different AI systems. We observed that TPOT$_{\mathrm{L}}$(Mean=2.85, Std=1.87), DAG-AI (Mean=2.68, Std=1.79) and DAG-AI$_{\mathrm{Stack}}$(Mean=2.65, Std=1.63) are the top three competitive systems.

Similarly, Figure 6 shows the box plot of rank for the *ROC AUC* performance metric and *10 minute* execution time. In this case, we obtained a total of 355 rankings based on the holdout score. Again, we observe that TPOT$_{\mathrm{L}}$(Mean=3.68, Std=2.48), DAG-AI (Mean=3.41, Std=2.37) and DAG-AI$_{\mathrm{Stack}}$ (Mean=3.06, Std=2.01) are the top three competitive systems. We also noticed that CatBoost, AutoML and DAUB are consistent in their performance across different groups (including Figure 5). One possible reason is that these systems frequently select a tree based ensemble model apart from doing additional parameter tuning and feature engineering.

Since, the performance of TPOT$_{\mathrm{L}}$ and DAG-AI are similar, we also made an attempt discover a way to improve TPOT. We identified two possible approaches: a) use of TPOT$_{\mathrm{S}}$ – a lower

[4]https://github.com/EpistasisLab/penn-ml-benchmarks
[5]https://www.openml.org/

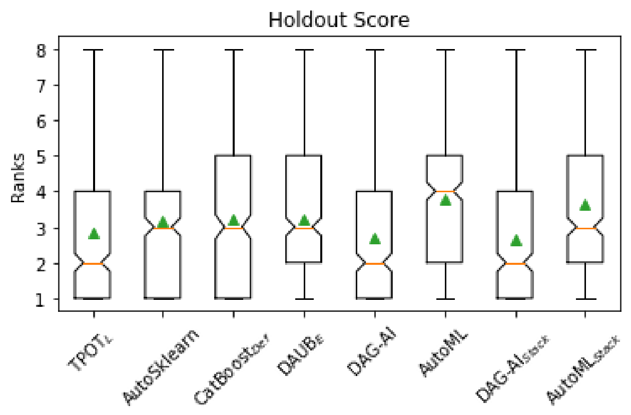


Fig. 5: AI system-wise box plot of rank for the accuracy metric and 10 min execution time.

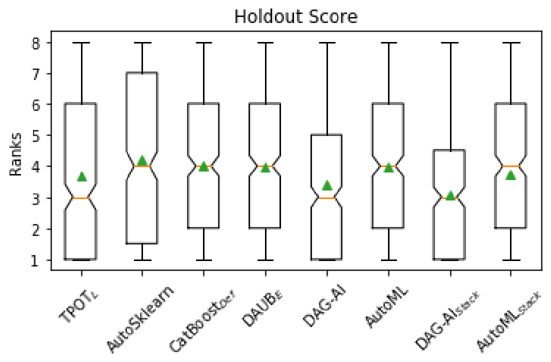


Fig. 6: AI system-wise box plot of rank for the ROC AUC metric and 10 min execution time.

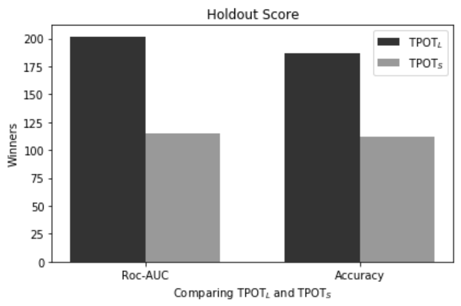


Fig. 7: Comparison of TPOT$_{\mathrm{L}}$ with TPOT$_{\mathrm{S}}$ for 10 min execution time.

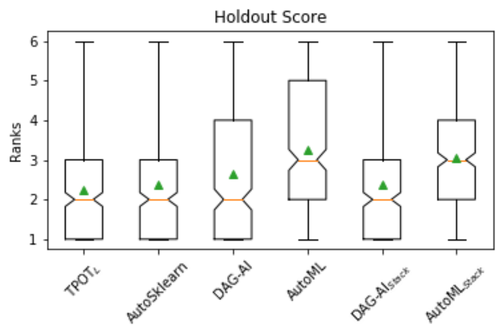


Fig. 8: AI system-wise box plot of rank for the accuracy metric and 180 min execution time.

configuration setting, or b) run TPOT or for a significantly longer time such (180 minutes) for each outer fold. As shown in Figure 7, the first option is not a viable solution, as the result obtained using TPOT_L is better than TPOT_S.[6] Figure 8 shows a comparison of holdout performance for the *accuracy* metric with *180 minutes* execution time. In this Figure, we see that the optimization driven systems (TPOT and AutoSklearn) are competitive, whereas optimization-free exploration method (DAG-AI and AutoML) achieve poorer ranks. We note that an exception to this behavior is the stacking-based DAG-AI system.

D. Pipeline Complexity Based Comparison

The output of all the AI systems considered here is a machine learning pipeline. This pipeline can be as simple as a single model or as complex as a tree/ensemble structure of other sub-pipelines. Before a model can be deployed for a real-time production use case, a data scientist will normally review the discovered pipeline for validation and improved understanding. It is also generally accepted that a simpler pipeline with slightly less performance is preferred over one which has high complexity. In this subsection, we investigate the output of AI systems from two perspective: a) unique number of components selected by the system (richness and diversity), and b) the number of components used to build the pipeline (simplicity). For analysis, we used the pipeline produced for each outer cross validation fold. We include AutoML, TPOT, AutoSklearn and DAG-AI in this context.

1) Number of Unique Components: Each dataset requires a different level of prepossessing or feature transformation. Thus, it is important that the correct transformer is available to the AI system, otherwise it is likely to be realized by a complex pipeline. Our classification pipeline graph is diverse and rich as it consists of almost all the possible operators we can include in the graph. When we calculated the unique number of components selected by AI systems, we noticed that DAG-AI selected 74 different components over all the classification experiments we conducted. Compared to DAG-AI, TPOT selected 44 different components, followed by AutoSklearn (33 components) and followed by AutoML (15 components). The diversity of available transformers allows an AI system to choose the most appropriate technique – potentially avoiding synthesizing a specialized pipeline which may lead to overfitting.

[6]Note that ties are ignored when counting winners.

We also noticed that the frequently used components to construct the pipeline overlap across different AI systems. These components include: XGBoost, gradient boosting classifier, polynomial feature generation, random forest classifier.

2) Pipeline Length: Next, we obtain the number of components used to build the resulting pipelines. AutoML only makes use of a single model so the length of pipeline is always 1, except when an ensemble model is used. In the case of DAG-AI, the final output is a path from the pipeline graph. Thus, the maximum number of components depends on the depth of the graph, which is 5 for the experiments here.

TPOT and AutoML are capable of building nested pipelines and ensemble pipelines, respectively. Figure 9 shows the histogram of the number of pipeline vs. pipeline length (for DAG-AI and TPOT) or ensemble size (for AutoSklearn). We noticed that the majority of pipelines generated by DAG-AI are of length 1 or 2. Having many pipelines with length 2 implies the need for data transformation, scaling, feature prepossessing before the data is used for modeling. In the case of TPOT, the pipeline length is under-estimated value as we considered a pipeline also as a single component. There is a sizable population of experiments where TPOT generates a complex result. AutoSklearn generates ensembles of many simple pipelines and each simple pipeline is at least length 3 longer. Highly complex pipelines can be difficult to verify or interpret. For DAG-AI understanding the output is considerably easier, yet it achieves performance equal to other state-of-the-art algorithms.

E. DAG-AI Algorithm Analysis

Optimization based approach makes some initial guess at the start and then use either derivative or derivative free information to search the space. Compared to the optimization driven search exploration, we have heuristic based method to explore the search space. In this section, we investigate how well our search technique evolve as time progress as well as how early the good solution can be obtained. We use results of our empirical study to identify the best solution for any given dataset. We treat the empirical best results as a ground truth to compare the performance of our search strategy. Briefly, we run our method on 10 classification datasets for 20 $minutes$ using $accuracy$ as a performance measure. At the end of each round, we capture the best cross validation score till so far and then compute how far is the current solution from the ground truth.

Figure 10 shows the outcome where y axis be the gap between empirically best solution minus the current solution. We observed that different algorithm produce results at different point of time (Each marker indicates completion of one round), but majority of experiments reduce the accuracy gap within 7 to 10 minutes. By this time, all the algorithms has completed three rounds. Indirectly, by completing third round, we have

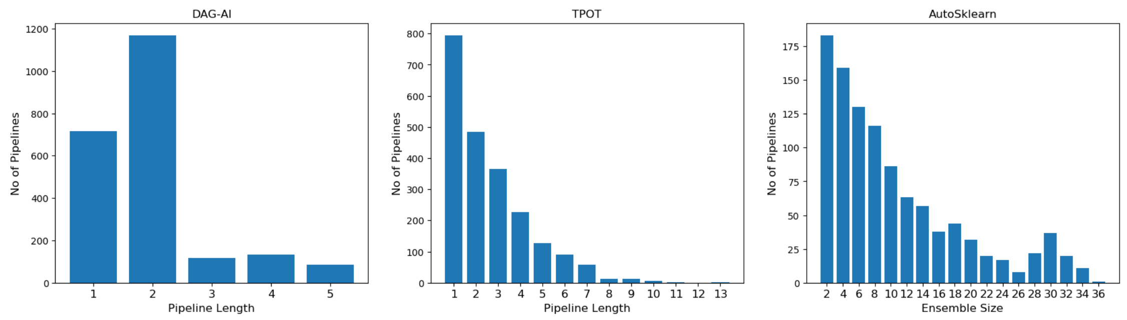


Fig. 9: Distribution of pipeline length or ensemble size for generated pipelines

rough estimates on the accuracy value. Compared to recent paper on curating ML pipelines in interactive way [25], our approach adopted a different approach where end user curate the pipeline graph and our system help to prioritise the execution of pipeline.

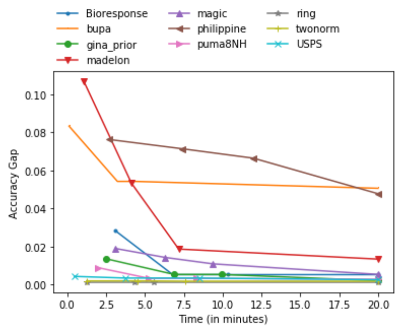


Fig. 10: Study of Converge of DAG-AI Algorithm

F. Scaling the Exploration of Pipeline Paths

In this section, we also show the results of a scaling analysis we performed for the three classification datasets. For each dataset, we ran a predefined classification pipeline and hyperparameter grid using random hyperparameter sampling. This combination results in approximately 18,000 pipeline paths to be explored. We explore the impact of using 80, 160, and 320 cores on total run time. For this analysis, we used our CeleryExecutor framework. Worker queues were deployed as Cloud Foundry (CF) instances communicating with a redis instance acting as both the message broker and result store. All infrastructure was provisioned on a popular cloud vendor's CF service. This approach is convenient as it is vendor agnostic because CF defines a standard deployment and management API. As can be seen in Figure 11, performance scales approximately linearly as the number of available cores (and by extension Celery workers) increases. Scaling is slightly sub-linear due mostly to additional communication bottlenecks introduced by task and data serialization between workers and message broker.

VII. Conclusion

In this paper, we have described a smart and versatile system that facilitates the task of building predictive AI/ML models

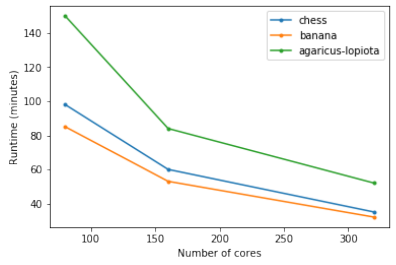


Fig. 11: Scalability Test of Large Pipeline Graph

in much easier way. We have witness usefulness of pipeline graph in many real project as it allow user to assemble the components from different libraries in more structured way. Expert user can create his own pipeline graph along with a parameter grid for optimization. Moreover, pipeline graph is empowered by different execution environments (spark, celery, cloud) and an optimization technique.

References

[1] M. J. Smith, C. Sala, J. M. Kanter, and K. Veeramachaneni, "The machine learning bazaar: Harnessing the ML ecosystem for effective system development," *CoRR*, vol. abs/1905.08942, 2019. [Online]. Available: http://arxiv.org/abs/1905.08942
[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2934664
[3] "Celery," https://github.com/celery/celery, accessed: 2020-15-07.
[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.
[5] "Keras," https://github.com/fchollet/keras, accessed: 2020-15-07.
[6] "py-earth," https://github.com/scikit-learn-contrib/py-earth, accessed: 2020-15-07.
[7] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *KDD*, 2016.
[8] S. Liu, P. Ram, D. Bouneffouf, G. Bramble, A. R. Conn, H. Samulowitz, and A. G. Gray, "Automated machine learning via ADMM," *CoRR*, vol. abs/1905.00424, 2019. [Online]. Available: http://arxiv.org/abs/1905.00424
[9] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," *ArXiv*, vol. abs/1907.10902, 2019.
[10] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *NIPS*, 2011.

[11] P. Koch, O. Golovidov, S. Gardner, B. Wujek, J. Griffin, and Y. Xu, “Autotune: A derivative-free optimization framework for hyperparameter tuning,” in *KDD*, 2018.
[12] “Scikit-optimize,” https://github.com/scikit-optimize/scikit-optimize, accessed: 2020-15-07.
[13] J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization,” *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012.
[14] “sklearn-deap,” https://github.com/rsteca/sklearn-deap, accessed: 2020-15-07.
[15] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, “Hyperband: A novel bandit-based approach to hyperparameter optimization,” *J. Mach. Learn. Res.*, vol. 18, pp. 185:1–185:52, 2016.
[16] A. Costa and G. Nannicini, “Rbfopt: an open-source library for black-box optimization with costly function evaluations,” *Mathematical Programming Computation*, vol. 10, pp. 597–629, 2018.
[17] M. Zöller and M. F. Huber, “Survey on automated machine learning,” *CoRR*, vol. abs/1904.12054, 2019. [Online]. Available: http://arxiv.org/abs/1904.12054
[18] “H2o_package,” http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html, accessed: 2020-15-07.
[19] A. Sabharwal, H. Samulowitz, and G. Tesauro, “Selecting near-optimal learners via incremental data allocation,” in *AAAI*, 2015.
[20] R. S. Olson and J. H. Moore, “Tpot: A tree-based pipeline optimization tool for automating machine learning,” in *Automated Machine Learning*, 2016.
[21] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, “Efficient and robust automated machine learning,” in *NIPS*, 2015.
[22] H. Jin, Q. Song, and X. Hu, “Auto-keras: An efficient neural architecture search system,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2019, pp. 1946–1956.
[23] L. Ostroumova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, “Catboost: unbiased boosting with categorical features,” in *NeurIPS*, 2017.
[24] M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer, and F. Hutter, “Auto-sklearn 2.0: The next generation,” *CoRR*, vol. abs/2007.04074, 2020. [Online]. Available: https://arxiv.org/abs/2007.04074
[25] Z. Shang, E. Zgraggen, B. Buratti, F. Kossmann, P. Eichmann, Y. Chung, C. Binnig, E. Upfal, and T. Kraska, “Democratizing data science through interactive curation of ml pipelines,” in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD ’19. New York, NY, USA: ACM, 2019, pp. 1171–1188. [Online]. Available: http://doi.acm.org/10.1145/3299869.3319863

VIII. Appendix

A. API for Pipeline Graph Construction

Listing 1 shows a code snippet in the Python programming language that constructs a pipeline graph for performing a classification task. The method ‘set_stages’ (See Line 11 in Listing 1) takes a list of lists of python tuples (if the user wants to give each pipeline node a custom name) or a list of lists of python objects if the system default name for objects is acceptable. In the example we set three stages: feature scaling, feature selection, and classification. Again, note that for compactness we did not use the recommended procedure of inserting (name, object) tuples relying instead on the system to internally use the python object’s string representation for the node name. For example, StandardScaler() will be treated as (“standardscaler”, StandardScaler()).

In Listing 1, the NoOp operation allows user to skip scaling altogether. Custom objects can be inserted at other phases as well as long as they are compatible with scikit-learn’s pipeline construct (e.g., a valid transformer/estimator). The design is flexible and user is free to add as many stages and as many hyperparameter options per stage. Using a similar API, we can construct a pipeline graph for a wide range of predictive modeling as well tasks such as regression, anomaly detection, and timeseries prediction. We also have “add_stages()” to construct a complex parallel pipeline graph.

```
1  def construct_graph():
2    pgraph = PipelineGraph()
3    stages = []
4    # stage 1: feature scaling
5    stages.append([MinMaxScaler(),StandardScaler()]
6    # stage 2: feature selection
7    stages.append([PCA(), SelectKBest(),NoOp()])
8    # stage 3: classifiers (estimators)
9    stages.append([RandomForestClassifier(),SGDClassifier()])
10   # populate pgraph
11   pgraph.set_stages(stages)
12   return pgraph
```

Listing 1: Construct custom classification pipeline graph

Listing 2 is a code sample showing how to prepare a hyperparameter grid `hpGrid` for a few different types of pipeline nodes (such as random forest classifier). In this example, we add five parameter domains in `hpGrid`, first two for feature selection and remaining three for classification. Each option is initialized with a list of options to test. For example, here we are interested in exploring the impact of the depth of a decision tree classifier by evaluating three different values - 5, 20 or “auto”. This is an example of a discrete hyperparameter grid. In our library, we have hand-crafted parameter grids for classification, regression task covering over 50 estimators and transformers available in scikit-learn, mlxtend, and other popular libraries. Since, many users are familiar with scikit-learn syntax, we adopted convention used in scikit-learn to prepare the grid.

```
def prepare_hyperparam_grid():
  hpGrid = dict{}
  g = hpGrid # for compactness
  # feature selection parameters
  g['pca__n_components'] = [3, 5, 7, 10]
  g['selectkbest__k'] = [10, 20, 50]
  # classification model parameters
  g['randomforestclassifier__depth'] \
   = [5, 'auto', 20]
  g['randomforestclassifier__n_estimators'] \
   = [20, 50, 100, 500]
  g['sgdclassifier__penalty'] \
    = ['l0','l1', 'elasticnet']
  return g
```

Listing 2: Hyperparameter grid creation

Listing 3 is a code sample show an API for executing a pipeline graph. User can configure the needed parameters including execution environment, optimizer, etc. The output of a call is model, best score and a path.

```
'''
pgraph is an instance of a PipelineGraph
D is input data
'''
def pipeline_graph_evaluation(pgraph,D):
    pgraph.set_cross_validation(k=5)
    pgraph.set_scoring('f1-score')
    exec_engine = \
      ExecutionEngine(pgraph,
                      D,
                      hpGrids,
                      optimizer,
                      exec_params)
    model, best_score, best_path = \
      exec_engine.execute()
    return model, best_score, best_path
```

Listing 3: Steps for Pipeline Graph Evaluation