HAIPipe: Combining Human-generated and Machine-generated Pipelines for Data Preparation

SIBEI CHEN, Renmin University of China, China
NAN TANG, QCRI / HKUST (GZ), Qatar / China
JU FAN*, Renmin University of China, China
XUEMI YAN, Renmin University of China, China
CHENGLIANG CHAI, Beijing Institute of Technology, China
GUOLIANG LI, Tsinghua University, China
XIAOYONG DU, Renmin University of China, China

Data preparation is crucial in achieving optimized results for machine learning (ML). However, having a good data preparation pipeline is highly non-trivial for ML practitioners, which is not only domain-specific, but also dataset-specific. There are two common practices. Human-generated pipelines (**HI-pipelines**) typically use a wide range of *any* operations or libraries but are highly experience- and heuristic-based. In contrast, machine-generated pipelines (**AI-pipelines**), *a.k.a.* AutoML, often adopt a predefined set of sophisticated operations and are search-based and optimized. These two common practices are mutually complementary. In this paper, we study a new problem that, given an HI-pipeline and an AI-pipeline for the same ML task, can we combine them to get a new pipeline (**HAI-pipeline**) that is better than the provided HI-pipeline and AI-pipeline? We propose HAIPipe, a framework to address the problem, which adopts an enumeration-sampling strategy to carefully select the best performing combined pipeline. We also introduce a reinforcement learning (RL) based approach to search an optimized AI-pipeline. Extensive experiments using 1400+ real-world HI-pipelines (Jupyter notebooks from Kaggle) verify that HAIPipe can significantly outperform the approaches using either HI-pipelines or AI-pipelines alone.

CCS Concepts: • **Information systems → Data analytics**.

Additional Key Words and Phrases: pipeline generation, data preparation, reinforcement learning

ACM Reference Format:
Sibei Chen, Nan Tang, Ju Fan, Xuemi Yan, Chengliang Chai, Guoliang Li, and Xiaoyong Du. 2023. HAIPipe: Combining Human-generated and Machine-generated Pipelines for Data Preparation. *Proc. ACM Manag. Data* 1, 1, Article 91 (May 2023), 26 pages. https://doi.org/10.1145/3588945

1 INTRODUCTION

Data preparation (data prep for short) is the process of turning raw data into a format that is ready for use by data science or ML tasks. This process is often implemented by a series of steps (*a.k.a.* a

*Ju Fan is the corresponding author.

Authors' addresses: Sibei Chen, Renmin University of China, Beijing, China, sibei@ruc.edu.cn; Nan Tang, QCRI / HKUST (GZ), Doha / Guangzhou, Qatar / China, ntang@hbku.edu.qa; Ju Fan, Renmin University of China, Beijing, China, fanj@ruc.edu.cn; Xuemi Yan, Renmin University of China, Beijing, China, yxmlpzt@ruc.edu.cn; Chengliang Chai, Beijing Institute of Technology, Beijing, China, ccl@bit.edu.cn; Guoliang Li, Tsinghua University, Beijing, China, liguoliang@tsinghua.edu.cn; Xiaoyong Du, Renmin University of China, Beijing, China, duyong@ruc.edu.cn.

2836-6573/2023/5-ART91 $15.00
https://doi.org/10.1145/3588945

pipeline) that the data moves through from one step to its subsequent steps. However, data prep is also known to be very hard and time-consuming [26].

Traditional data prep pipelines are human orchestrated, which are abbreviated as HI-pipelines. The pros of HI-pipelines are that users can use "any" operation and can easily inject domain knowledge (*e.g.,* "?" means a missing value for occupation). The cons are that users typically orchestrate pipelines based on their experience, which are heuristic and are rarely optimized, *e.g.,* humans never manually try all different combinations and find the best one.

Recent advances in the ML community have extended AutoML from hyper-parameter tuning and neural architecture search to an attempt that tries to automate the process of data prep for ML (*e.g.,* Auto-SKLearn [9] and DeepLine [12]) – these machine-generated pipelines are called AI-pipelines in this paper. The pros of AI-pipelines are that they use a constrained set of operations, for which they can find the best pipeline through exhaustively evaluating different pipelines within a pre-defined search space. The cons of AI-pipelines are that they lack domain knowledge.

Our **key observation** is that HI-pipelines and AI-pipelines are naturally complementary. That is, the pros of HI-pipelines (*i.e.,* using a wider range of operations based on domain knowledge) are the cons of AI-pipelines, and the cons of HI-pipelines (*i.e.,* experience-based and heuristic-based) are the pros of AI-pipelines (*i.e.,* highly optimized). Therefore, intuitively, HI-pipelines and AI-pipelines should be combined, so as to get the best of both worlds.

In this paper, we study a new problem: Given an HI-pipeline and an AI-pipeline for an ML data prep task, can we generate a better data prep pipeline (*i.e.,* an HAI-pipeline) by combining them?

Example 1. *Consider Figure 1. After loading the dataset (lines 1–4), an HI-pipeline consists of two components h_1 (lines 5-8,* i.e., *dropping rows with missing values) and h_2 (lines 24-28,* i.e., *scaling numerical attributes). Then, we use a machine-generated AI-pipeline that consists of three components a_1 (lines 10–15,* i.e., *encoding categorical attributes), a_2 (lines 17–22,* i.e., *generating interaction features) and a_3 (lines 30–35,* i.e., *removing useless features). A combination of these two pipelines is given in Figure 1, which outperforms both the HI-pipeline and the AI-pipeline.*

To address the problem, we propose HAIPipe, a framework that combines an HI-pipeline with an AI-pipeline for a specific ML task.

There are **two challenges**. (1) How to combine an HI-pipeline and an AI-pipeline? (2) How to obtain an optimized AI-pipeline from various possible combinations?

For Challenge (1), we devise an *enumeration-sampling* approach that first enumerates possible HAI-pipelines, and then employs an active learning strategy to sample a limited number of HAI-pipelines for evaluation, so as to find the one with the highest performance. For Challenge (2), there are a variety of operation families (*e.g.,* `Imputer` and `Scaler`). Within each family, there are many operations (*e.g.,* `MinMaxScaler` and `StandardScaler`). Thus, there exist many possible ways of selecting operations to form an AI-pipeline, which are highly dataset-specific and task-specific. We introduce a reinforcement learning (RL) based approach that learns how to select operations through offline training.

Contributions. Our contributions are summarized as follows.

(1) We formally define data prep pipelines, and introduce HI-pipelines and AI-pipelines (Section 2).

(2) We propose a framework, namely HAIPipe, that combines HI-pipelines and AI-pipelines for data prep (Section 3). We devise an enumeration-sampling strategy that effectively and efficiently combines HI-pipelines and AI-pipelines (Section 4).

(3) We present an RL-based AI-pipeline algorithm that pays particular attention on optimizing data prep operations (Section 5).

(4) We conduct extensive experiments to show that HAIPipe can significantly outperform both the

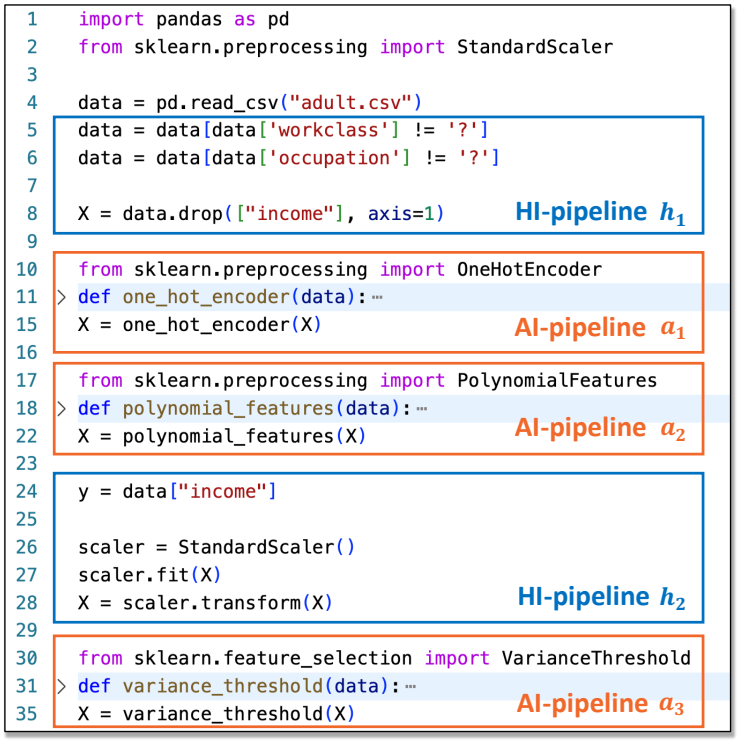


Fig. 1. An example of combining an HI-pipeline and an AI-pipeline. Blue blocks: human-written scripts. Orange blocks: machine-generated scripts.

HI-pipelines and AI-pipelines (Section 6). For reproducibility, we make code and datasets in our experiments public on GitHub[1].

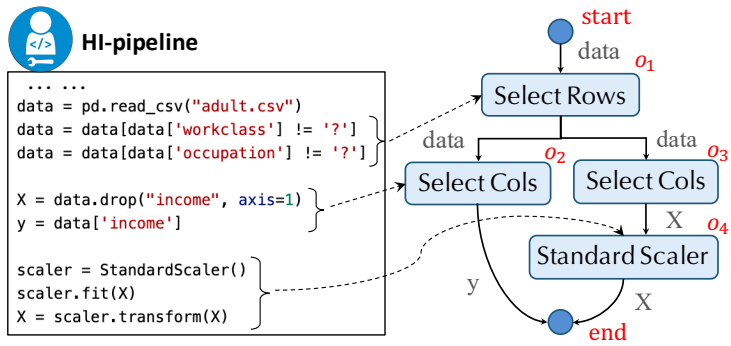
2 DATA PREP PIPELINES

Let $\mathcal{T}(D, M)$ be an ML task with an ML model M and training data D. Without loss of generality, this paper considers classification models for ease of presentation. Our techniques can be easily extended to other supervised ML models. Practically, finding the "best" model M often requires a sequence of *data prep steps* to obtain the "best" data, including imputation, scaling, etc., which are naturally formalized as *operations* as below.

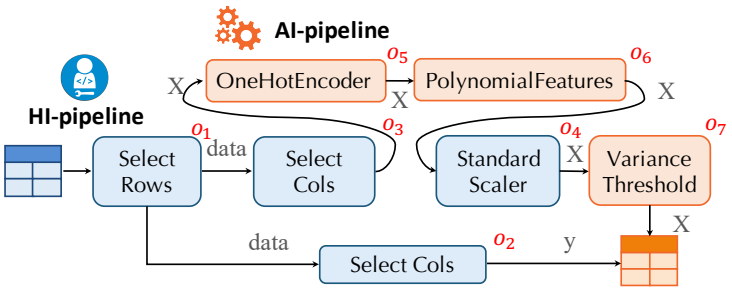
Data prep operations. A *data prep operation* (or operation for short), which is denoted as o, encapsulates a specific functionality that transforms a dataset D into another dataset D', *i.e.,* $D' = o(D)$. Let Σ denote the universe of operations for our task $\mathcal{T}$, which contains a set of diversified *families* of data prep functionalities. Among them, some rely on well-developed algorithms in known Python libraries, such as Scikit-Learn [27]. In this paper, we consider the following families of data prep algorithms implemented in Scikit-Learn (*i.e.,* `sklearn`), likewise in some existing studies [12, 35].

- `Imputer`: a data cleaning step that fills in missing values in dataset D, *e.g.,* `impute.SimpleImputer`.

[1]https://github.com/ruc-datalab/Haipipe



(a) Modeling multiple HI operations as a pipeline.



(b) Combining HI- and AI-pipelines to generate an HAI-pipeline.

Fig. 2. An example of HAI-pipeline generation.

• `Encoder`: a transformation step that converts categorical features in the dataset D into integers or numerical vectors, *e.g.*, `preprocessing.OneHotEncoder`.

• `Scaler`: a standardization step that rescales data in D, *e.g.*, `preprocessing.StandardScaler`.

• `Feature Transformer`: a feature engineering step that either reduces the dimensionality of dataset D (*e.g.*, `decomposition.PCA`), or generates new features by combining features in D (*e.g.*, `preprocessing.PolynomialFeatures`).

• `Feature Selection`: a feature selection step that drops irrelevant or unimportant features in D for dimensionality reduction (*e.g.*, `feature_selection.VarianceThreshold`).

Besides directly using algorithms from known libraries, data scientists may also write specific programs for data prep based on their domain knowledge. For example, in Figure 2(a), a data scientist writes code to select rows in dataset D without "?" in `workclass` and `occupation`, *e.g.*, `data = data[data['workclass'] != '?']`. This is because, the data scientist knows that this symbol represents a missing value, which would then damage downstream ML task.

Data prep pipelines. A typical data prep process is composed of multiple operations that collectively transform a raw dataset D into a prepared dataset $D_{\texttt{train}}$ for training model M. To formalize this, we introduce *data prep pipeline* (or pipeline for short).

Definition 1 (Pipeline). *A pipeline is defined as a directed acyclic graph $G = (O, E)$, where $O \subseteq \Sigma$ is a set of operations and each edge $e_{ij} = (o_i, o_j)$ in E represents a data flow from operation o_i to operation o_j. In particular, there are two special operations in O, namely* `start` *and* `end`*, which indicate the starting and ending of pipeline G, respectively. We also define a partial order $o_i \prec_G o_j$ for operations o_i and o_j, iff there is a path from o_i to o_j in G.*

Intuitively, the partial order can be taken as a constraint to specify which operations should be executed first. Moreover, as we want to identify the "best" pipeline for task $\mathcal{T} = (D, M)$, we introduce $\Phi_{\mathcal{T}}(G)$ to evaluate the performance of pipeline G w.r.t. task $\mathcal{T}$. Specifically, let $D_{\texttt{train}}$ and $D_{\texttt{test}}$ denote the dataset prepared by G and a *holdout* test dataset respectively. We use $D_{\texttt{train}}$ to train model M and evaluate M on $D_{\texttt{test}}$. Then, we obtain an evaluation score, *e.g.,* accuracy for classification, on $D_{\texttt{test}}$, and use the score as the performance of the pipeline $\Phi_{\mathcal{T}}(G)$. For ease of presentation, if the context is clear, we simply use $\Phi(G)$ by omitting the subscript.

Example 2. *Figure 2(a) shows an example pipeline G generated from a data prep program written by a data scientist. The pipeline takes as input a dataset* `data`*, which is loaded from a CSV file. It first uses operation o_1 to select rows without having symbol "?" in attributes* `workclass` *and* `occupation`*. Then, it uses o_2 and o_3 to select columns for generating features* $\mathbf{X}$ *and predict target* $\mathbf{y}$ *respectively. Next, the pipeline leverages o_4,* i.e., *the* `StandardScaler` *in Scikit-Learn to normalize values in* $\mathbf{X}$*. Finally, both* $\mathbf{X}$ *and* $\mathbf{y}$ *are combined to produce the prepared dataset $D_{\texttt{train}}$. The performance $\Phi(G)$ of pipeline G is then evaluated by the holdout test accuracy of model M trained on $D_{\texttt{train}}$. Moreover, there exists a partial order $o_1 \prec o_4$, which indicates that removing rows with missing values should be executed before data scaling, because, otherwise, the pipeline may become invalid as o_4 would encounter unexpected errors.*

As data prep can be orchestrated by both human and ML algorithms, we introduce *human-generated pipelines* (or *HI-pipelines*) and *machine-generated pipelines* (or *AI-pipelines*) as follows.

HI-pipeline & AI-pipeline. An HI-pipeline, denoted as $G_{\texttt{H}}$, is constructed from a human-generated data prep program, as illustrated in Figure 2(a). Typically, an HI-pipeline contains both sophisticated algorithms from known libraries, *e.g.,* `StandardScaler`, and user-written operations/functions, *e.g.,* domain specific row selection criteria. An AI-pipeline, denoted as $G_{\texttt{A}}$, can be generated by leveraging machine learning techniques (*a.k.a.* AutoML).

HI-pipelines and AI-pipelines have *different yet complementary* characteristics. On the one hand, HI-pipelines are manually optimized by injecting domain knowledge, but they might be limited to a small range of families. As shown in our experiments (see Table 4 in Section 6 **Exp-5**), many sophisticated operations, such as `TruncatedSVD` for dimensionality reduction and `KBinsDiscretizer` for discretization are rarely used. On the other hand, although AI-pipelines are extensively explored and optimized over a wide range of operations, they are not highly optimized to recommend operations for specific domains. Therefore, it is highly desirable to generate a hybrid pipeline, denoted by $G_{\texttt{HAI}}$, by carefully combining both HI-pipelines and AI-pipelines.

Example 3. *Figure 2(b) shows an example HAI-pipeline, which is generated by combining the HI-pipeline in Figure 2(a) with an AI-pipeline containing the following operations. (1) o_5 (*`OnehotEncoder`*) transforms categorical values into numerical vectors, (2) o_6 (*`PolynomialFeatures`*) generates polynomial and interaction features, and (3) o_7 (*`VarianceThreshold`*) removes all low-variance features. We can see that the HAI-pipeline applies better encoding for categorical data, generates more features, and removes useless features. Thus, the pipeline would produce a better dataset $D_{\texttt{train}}$ and improve the performance of model training.*

3 AN OVERVIEW OF HAIPIPE

Figure 3 provides an overview of our proposed HAIPipe framework. It takes a human-generated data prep program (HI-program) as input. The core part of HAIPipe is to combine this HI-pipeline with an optimized machine generated AI-pipeline, which outputs an HAI data prep program (HAI-program) that is executable Python code. More specifically, HAIPipe works as follows.

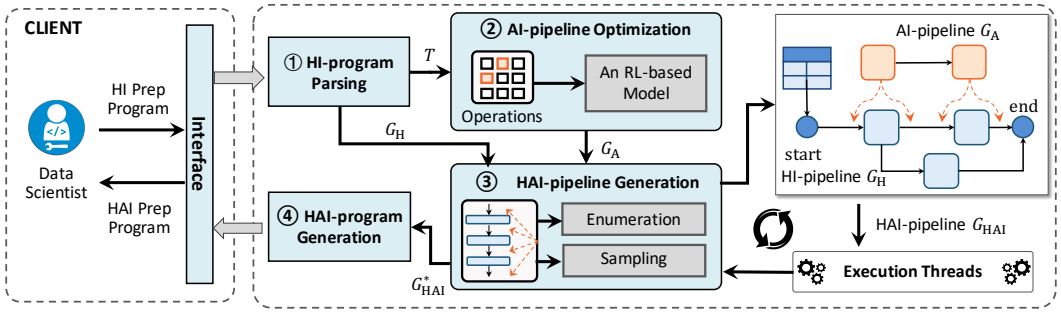


Fig. 3. Architecture of the HAIPipe framework.

Step 1 - HI-program Parsing. The human written program is parsed by the *HI-program Parsing* step into an HI-pipeline G_H and a task $\mathcal{T}$. To this end, this component exploits the Abstract Syntax Tree (AST) technique, like the existing work [22]. Specifically, given a Python program as shown in Figure 4(a), we parse the program to an HI-pipeline in the following steps.

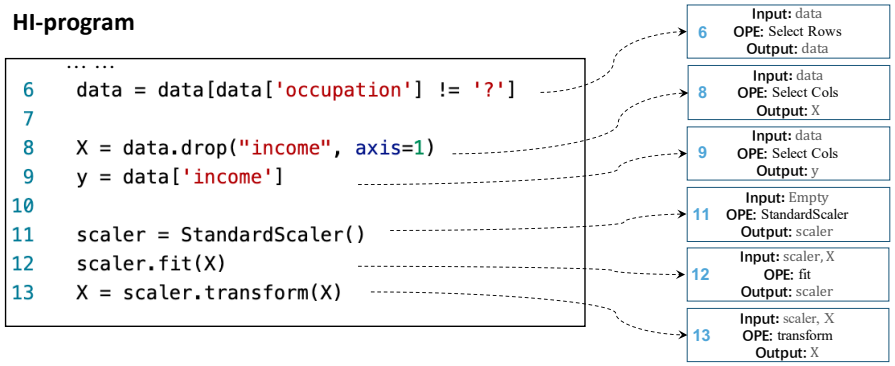
(i) ASTNode Parsing: It extracts atomic data transformation blocks from the code. We first split the code into code segments according to the syntax. Then, we use the Python AST package [20] to parse each code segment into an *ASTNode*, where an *ASTNode* contains information of input, operation, output and position of the corresponding segment, as shown in Figure 4(a).

(ii) Pipeline Construction: We combine individual *ASTNodes* into a directed acyclic graph based on the data-flow among them, *i.e.,* connecting two *ASTNodes* if the output of the former is the input of the latter. Please see the left of Figure 4(b) for an example.

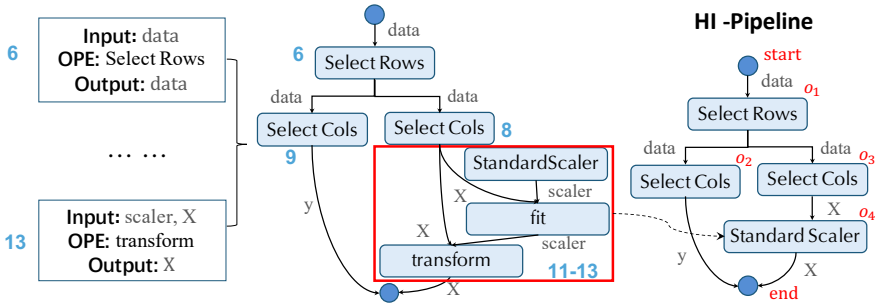
(iii) Simplification: Some data prep operations may contain multiple code segments, *i.e.,* multiple *ASTNodes* in the above data-flow graph. For example, `fit` and `transform` of `StandardScaler` in Figure 4(b) should be an inseparable combination and thus will be merged. To address such issues, we heuristically define a set of rules for simplifying *ASTNodes* in the data-flow graph.

Step 2 - AI-pipeline Generation and Optimization. In this step, we can plug in existing solutions such as DeepLine [12] that directly obtains an AI-pipeline. However, we empirically found that it does not well explore a predefined set of sophisticated operations to find an optimized AI-pipeline. To better serve HAIPipe and handle the challenge that there are a large variety of operation families (*e.g.,* `Imputer` and `Scaler`), and within one family there are many operations (*e.g.,* `MinMaxScaler` and `StandardScaler`), we introduce a novel reinforcement learning (RL) based approach. It first learns to progressively select operations during offline training. During online inference, it uses the learned RL model to generate an AI-pipeline G_A. See Section 5 for more details.

Step 3 - HAI-pipeline Generation. This step decides how to combine the AI-pipeline G_A generated in Step 2 with the HI-pipeline G_H parsed from the HI data prep program in Step 1, and finally outputs an HAI-pipeline G^*_{HAI} with the highest performance. Take Figure 2(b) as an example. Operations o_5 (`OneHotEncoder`) and o_6 (`PolynomialFeatures`) should be inserted before o_4 (`StandardScaler`) as the newly generated features by o_5 and o_6 also need to be normalized by o_4. Moreover, o_7 (`VarianceThreshold`) is better to be executed after o_4, because the feature selection algorithm in o_7 removes the features with low variances, which requires that all features have similar scales. To tackle this problem, we devise an *enumeration-sampling* approach that first enumerates possible HAI-pipelines, and then employs an active learning strategy to sample a limited number of HAI-pipelines for evaluation. More details will be described in Section 4.



(a) ASTNode Parsing: extracting data transformation blocks from the code.



(b) Pipeline Construction and Simplification: generating and simplifying a data prep pipeline.

Fig. 4. Overview of HI-program Parsing.

Step 4 - HAI Program Generation. After obtaining the HAI-pipeline G_{HAI}, the *HAI Program Generation* step generates a data prep program, which is then returned to the data scientist. The key issue is to ensure successfully translating an HAI-pipeline into executable program . Recall that we have recorded the positions (*i.e.,* code-segment lines) corresponding to different operations in the pipeline. Thus, we just need to add/delete the code segments at the corresponding positions of the HI-pipeline. To ensure successful execution, we implement a module to handle runtime errors such as missing package / missing data files by programmatically installing missing packages or fixing the erroneous data paths, similar to what Auto-suggest [34] proposed.

4 HAI-PIPELINE GENERATION

HAI-pipeline generation aims to combine the parsed HI-pipeline G_{H} with the generated AI-pipeline G_{A}. Obviously, there are many ways of combining an HI-pipeline and an AI-pipeline. To reduce the search space, we do not change the order of operations in either HI-pipelines or AI-pipelines. The reason is two-fold. First, some operations (*e.g.,* removing missing values) may be prerequisites for their following operations (*e.g.,* generating interaction features). Thus, changing the operation ordering has a risk of inducing run-time errors. Second, the order of operations in the AI-pipeline may be optimized, *e.g.,* referring to our reinforcement learning model in the next section. Based on the above intuition, we define an HAI-pipeline as follows.

Definition 2 (HAI-pipeline). *Given an ML task $\mathcal{T}(D, M)$, let $G_{\mathsf{H}} = (O_{\mathsf{H}}, E_{\mathsf{H}})$ and $G_{\mathsf{A}} = (O_{\mathsf{A}}, E_{\mathsf{A}})$ denote an HI-pipeline and an AI-pipeline respectively. An HAI-pipeline is defined as $G_{\mathtt{HAI}} = (O_{\mathtt{HAI}}, E_{\mathtt{HAI}})$ such that:*

(1) Operations $O_{\mathtt{HAI}}$ in $G_{\mathtt{HAI}}$ is a subset of the union between O_{H} and O_{A}, i.e., *$O_{\mathtt{HAI}} \subseteq O_{\mathsf{H}} \cup O_{\mathsf{A}}$;*
(2) Operations originally in G_{H} (or in G_{A}) preserve their original partial order relationships, i.e., *$\forall o_i, o_j \in O_{\mathtt{HAI}} - O_{\mathsf{A}}$ (or $\forall o_i, o_j \in O_{\mathtt{HAI}} - O_{\mathsf{H}}$), if $o_i \prec_{G_{\mathsf{H}}} o_j$ (or $o_i \prec_{G_{\mathsf{A}}} o_j$), then $o_i \prec_{G_{\mathtt{HAI}}} o_j$.*
(3) Operations originally in G_{A} have new partial order relationships with operations originally in G_{H}, i.e., *$\forall o \in O_{\mathtt{HAI}} - O_{\mathsf{H}}$, $\exists o' \in O_{\mathtt{HAI}} - O_{\mathsf{A}}$ satisfying $o \prec_{G_{\mathtt{HAI}}} o'$ or $o' \prec_{G_{\mathtt{HAI}}} o$, and vice versa;*

Naturally, what we need is the *best* HAI-pipeline with the highest holdout test performance, *i.e.,*

$$G^*_{\mathtt{HAI}} = \underset{G_{\mathtt{HAI}} | G_{\mathsf{H}}, G_{\mathsf{A}}}{\mathtt{argmax}}\ \Phi_{test}(G_{\mathtt{HAI}}).$$

To this end, we divide the above problem into two sub-problems. The **first sub-problem** is to find all possible combinations.

Definition 3 (HAI-pipelines Enumeration). *Given an ML task $\mathcal{T}(D, M)$, HI-pipeline $G_{\mathsf{H}} = (O_{\mathsf{H}}, E_{\mathsf{H}})$ and AI-pipeline $G_{\mathsf{A}} = (O_{\mathsf{A}}, E_{\mathsf{A}})$, the problem aims to find all possible HAI-pipelines $G_{\mathtt{HAI}}$ defined in Definition 2.*

After finding candidate combined pipelines, the **second sub-problem** is to select the *best* one given a budget on the number of evaluated pipelines, as defined below.

Definition 4 (Best Pipeline Selection.). *Given an ML task $\mathcal{T}(D, M)$ and some candidate HAI-pipelines $C = \{G_{\mathtt{HAI}}\}$. Let K be the budget, which means that only K pipelines can be evaluated on validation dataset $D_{\mathtt{val}}$ to get the validation accuracy $\Phi_{\mathtt{val}}$. This problem aims to find the HAI-pipeline $G^*_{\mathtt{HAI}}$ in C with the highest validation accuracy $\Phi_{\mathtt{val}}$,* i.e., *$G^*_{\mathtt{HAI}} = \mathtt{argmax}_{G_{\mathtt{HAI}} | C}\ \Phi_{\mathtt{val}}(G_{\mathtt{HAI}})$.*

4.1 HAI-pipelines Enumeration

Enumeration Strategy. Because some operations in human-generated G_{H} may be redundant or have negative effects, we enumerate sub-graphs of G_{H}, denoted by $\{G^1_{\mathsf{H}}, G^2_{\mathsf{H}}, \ldots, G^m_{\mathsf{H}}\}$, by removing operations from G_{H}. Recall that HI operations need to preserve their partial-order relationships as mentioned in Definition 2. Thus, if we remove an operation from G_{H}, we also remove all the operations following the operation. Moreover, if the number of sub-graphs is large, we perform random sampling to save time.

Combining an HI-pipeline and an AI-pipeline. For each G^i_{H}, we combine it with an AI-pipeline G_{A} by progressively inserting operations in G_{A} to G^i_{H}, as illustrated in Figure 5. For ease of presentation, we use o^{H} and o^{A} to denote the operations in G^i_{H} and G_{A} respectively. Specifically, we first consider o^{A}_1 and enumerate possible HI operations, after which o^{A}_1 can be inserted. Taking Figure 5 as an example, we insert o^{A}_1 after o^{H}_1, o^{H}_2, o^{H}_3 and o^{H}_4, or simply omitting o^{A}_1 for insertion. We represent these enumeration choices at the first level of the tree in the right part of Figure 5. After that, we further consider o^{A}_2: due to the partial order relationship o^{A}_1, *i.e.,* $o^{\mathsf{A}}_1 \prec o^{\mathsf{A}}_2$, we can only insert o^{A}_2 after o^{A}_1. If we insert o^{A}_1 after o^{H}_3, due to the partial order relationship, we can only insert o^{A}_2 after o^{H}_3 and o^{H}_4, as shown in the second level of the tree. Following the above insertion process, we obtain a set of HAI-pipelines, *i.e.,* the leaf nodes. Note that, in our implementation, we do not limit operations to affect on the whole table or some columns/rows. For example, in G^i_{H}, users can operate on certain columns or the entire table. On the other hand, for $G_{\mathtt{HAI}}$, suppose that we insert o^{A} after an o^{H}. Then, the columns applied by o^{A} would depend on the output of o^{H}.

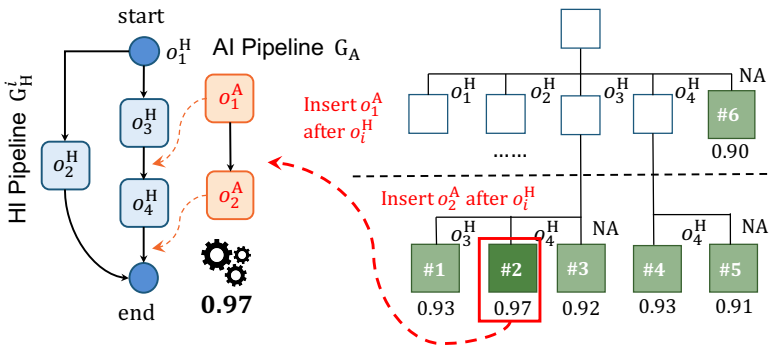


Fig. 5. Combining an HI-pipeline and an AI-pipeline.

4.2 Best Pipeline Selection

Ideally, we evaluate performance score $\Phi(G_{\texttt{HAI}})$ for each leaf $G_{\texttt{HAI}}$, as shown in Figure 5, and find the one with the highest score, *i.e.*, $G^*_{\texttt{HAI}} = \texttt{argmax}_{G_{\texttt{HAI}}} \Phi(G_{\texttt{HAI}})$, *e.g.*, the one with number #2. As discussed previously, such a processing is very time-consuming. Thus, we describe how to select the best HAI-pipeline from the candidate pipelines being enumerated, with the constraints that at most K pipelines can be evaluated.

The main difficulty is how to estimate the possible performance of each HAI-pipeline that is not evaluated. We propose a two-step method that learns to predict the performance of pipelines.

Step 1 proposes a neural network that learns the latent relationship between a pipeline and its performance. We model it as a regression problem and introduce an *Alternative Evaluation Network*, which takes the dataset features and pipeline features as input and outputs the predicted performance score. Figure 6 shows an overview of the *Alternative Evaluation Network*.

Step 2 uses *active learning* to select K HAI-pipelines as training data to train the above neural network, so as to find the samples that help model training at the most.

Finally, after the neural network has been well-trained, it can infer the performance of all HAI-pipelines and select the best one.

Network Architecture. The *Alternative Evaluation Network* f considers both dataset features and pipeline features as input, which are described as follows.

(1) Dataset features. The performance naturally depends on the characteristics of dataset D, such as distribution of features and number of records. Thus, we introduce a feature vector $\mathbf{\Psi}$ to represent *statistical* information of D. Specifically, for each column in D, we use a vector $\mathbf{\Psi}^{(i)}$ to summarize statistical information of the column, such as the ratio of missing values, column types, mean, median, standard variance, number of unique values, etc. Then, we concatenate $\mathbf{\Psi}^{(i)}$ of all columns to produce dataset features $\mathbf{\Psi}$. Note that, as neural networks need feature vectors to be fixed-sized, we use padding to handle datasets with various column numbers. We fed $\mathbf{\Psi}$ to a DNN with several dense hidden layers with `LeakyRelu` as activation, and finally obtain dataset features $\mathbf{\Psi}_{\texttt{emb}}$.

(2) Pipeline features. The performance of an HAI-pipeline can also be inferred from the operations and their orderings in the pipeline. For example, a `Feature Selection` operation may be necessary after `PolynomialFeatures` as the latter would bring useless interaction features and this combination may bring high performance. To capture this, we introduce an LSTM model [13] to encode the sequence of operations in G, and use the hidden layer of the LSTM model as the pipeline features $\mathbf{Y}$. It is worth noting that operations in HAI-pipelines can be divided into HI-operations

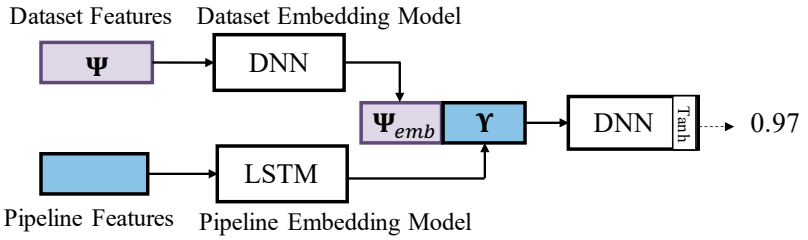


Fig. 6. Alternative Evaluation Network's architecture.

Algorithm 1: Best HAI-pipeline Selection

Require: Dataset D, candidate HAI-pipelines $C = \{G^1_{\texttt{HAI}}, G^2_{\texttt{HAI}}, ..., G^n_{\texttt{HAI}}\}$, and a number K.
Ensure: Best HAI-pipeline $G^*_{\texttt{HAI}}$

Initialize the maximum iteration number T.
Initialize the K-means parameter, *i.e.,* number of clusters τ.
Initialize an empty pipeline selection set U.
$k \leftarrow K/T$
Alternative Evaluation Network $f \leftarrow$ WarmupTraning()
for each $i \in [1, T]$ **do**
  Select HAI-pipelines $N = Kmeans(C - U, \tau)$ using Equation (1).
  Calculate $Score^{\texttt{rep}}(G_{\texttt{HAI}})$ for each $G_{\texttt{HAI}}$ in N for representative score using Equation (2).
  Calculate $Score^{\texttt{inf}}(G_{\texttt{HAI}})$ for each $G_{\texttt{HAI}}$ in N for informative score using Equation (3).
  Normalize $Score^{\texttt{rep}}(G_{\texttt{HAI}})$ and $Score^{\texttt{inf}}(G_{\texttt{HAI}})$ into $[0, 1]$.
  Calculate final score $Score(G_{\texttt{HAI}})$ for $G_{\texttt{HAI}}$ with Equation (4).
  Select the k HAI-pipelines as U_i with the highest $Score(G_{\texttt{HAI}})$ and evaluate them.
  Optimize our *Alternative Evaluation Network* f with U_i.
  $U = U \cup U_i$
end for
$G^*_{\texttt{HAI}} = \arg\max_{G_{\texttt{HAI}} \in C}(f(\mathbf{\Psi}_{G_{\texttt{HAI}}}, \mathbf{\Upsilon}_{G_{\texttt{HAI}}}))$

and ML-operations, which are originally from $G_{\texttt{H}}$ and $G_{\texttt{A}}$ respectively. Since HI-operations can be black-box operations, we just encode them uniformly.

Given the dataset and pipeline features, we concatenate them together and obtain a latent feature vector $\mathbf{l} = \texttt{concat}(\mathbf{\Psi}_{emb}, \mathbf{\Upsilon})$. Then, we fed the latent feature vector to a DNN with several dense hidden layers with `LeakyRelu` as activation, and finally output the predicted performance score of an HAI pipeline by using a `tanh` layer that normalizes the scores.

Active Learning Strategy for Training Data Selection. To train our *Alternative Evaluation Network* f, we need training data, which are HAI-pipelines with performance evaluated on the validation set. However, we can only afford to evaluate K HAI-pipelines as training data from all possible pipelines $C = \{G^1_{\texttt{HAI}}, G^2_{\texttt{HAI}}, ...\}$ to ensure efficiency. To this end, we introduce an active learning strategy to effectively select pipelines to generate training data.

As shown in Algorithm 1, we develop an iterative process with T iterations to generate training set U. Specifically, for the i^{th} iteration, we select a set of HAI-pipelines as U_i with size K/T, use U_i to optimize our *Alternative Evaluation Network* f, and add U_i into U. The key issue here is how to select U_i. To address this, we consider two factors to measure the *benefits* of HAI-pipelines.

(1) *Representativeness.* Intuitively, if an HAI-pipeline $G_{\texttt{HAI}}$ has many other similar HAI-pipelines, it can be selected as a *representative* one to train our *Alternative Evaluation Network* f.

Specifically, to find such representative HAI-pipelines, we first use a cluster algorithm (*e.g.,* K-Means) to find τ pipeline clusters. Then, in each cluster, we select the HAI-pipeline that is most close to the cluster centroid as the representative one.

(2) *Informativeness.* We prefer *informative* HAI-pipelines that are most helpful for model training, *i.e.,* leading to most significant changes to our *Alternative Evaluation Network* f. As directly computing such changes is very expensive, we use EMCM [5] to simulate the differences. More specifically, given an HAI-pipeline $G_{\texttt{HAI}}$, EMCM simulates the differences between the current model parameters and the parameters of the updated model trained with $G_{\texttt{HAI}}$.

Based on the above two factors, we introduce two scores $Score^{\texttt{rep}}$ and $Score^{\texttt{inf}}$ to measure the representativeness and informativeness of a HAI-pipeline $G_{\texttt{HAI}}$ respectively, and select K pipelines from a set C of candidate pipelines as follows.

For each candidate $G_{\texttt{HAI}}$ in $C-U$ that is not selected yet, we use the current *Alternative Evaluation Network* f to generate its latent features, *i.e.,* $\mathbf{l}_{G_{\texttt{HAI}}} = \texttt{concat}(\mathbf{\Psi}_{emb}, \mathrm{Y})$ described above. Then, we perform a K-Means algorithm on this latent feature space, and obtain τ clusters of HAI-pipelines. For each cluster c, we select the pipeline $G^c_{\texttt{HAI}}$ that is most close to the centroid of the cluster, *i.e.,*

$$G^c_{\texttt{HAI}} = \arg\min_{G_{\texttt{HAI}} \in c} \|\mathbf{l}_{G_{\texttt{HAI}}} - \frac{1}{|c|} \sum_{G'_{\texttt{HAI}} \in c} \mathbf{l}_{G'_{\texttt{HAI}}}\|, \tag{1}$$

where $\|\cdot\|$ denotes the Euclidean distance. In this way, from all the τ clusters, we can select a candidate set of HAI-pipelines, denoted as $N = \{G^{c_1}_{\texttt{HAI}}, G^{c_2}_{\texttt{HAI}}, \ldots, G^{c_\tau}_{\texttt{HAI}}\}$.

For each candidate pipeline $G_{\texttt{HAI}}$ in N, we examine whether $G_{\texttt{HAI}}$ is far from the pipelines U we already selected. If so, this pipeline would be more representative. To this end, we calculate the score $Score^{\texttt{rep}}$ for each $G_{\texttt{HAI}}$ in N as the average distance between $G_{\texttt{HAI}}$ and the pipelines in U, *i.e.,*

$$Score^{\texttt{rep}}(G_{\texttt{HAI}}) = \frac{1}{|U|} \sum_{G'_{\texttt{HAI}} \in U} \|\mathbf{l}_{G_{\texttt{HAI}}} - \mathbf{l}_{G'_{\texttt{HAI}}}\|. \tag{2}$$

On the other hand, to select pipelines that lead to the largest changes to our *Alternative Evaluation Network*, we introduce $Score^{\texttt{inf}}$ to measure how a selected pipeline changes the model parameters. Given a HAI-pipeline $G_{\texttt{HAI}}$, following the existing method [5], we use the model gradient to measure such change, which is denoted as $\Theta_{G_{\texttt{HAI}}}$. As gradient $\Theta_{G_{\texttt{HAI}}}$ cannot be calculated without the ground truth label $\Phi_{G_{\texttt{HAI}}}$, we utilize a bootstrap method to construct an ensemble B that contains κ estimate models to estimate the prediction distribution of $\Phi_{G_{\texttt{HAI}}}$. Then, we use the expected model change $\Theta^b_{G_{\texttt{HAI}}}$ of each estimate model b in B to approximate the true model change $\Theta_{G_{\texttt{HAI}}}$. The connection between bootstrap and prediction distribution has been studied in previous studies [5], and we implement their solutions. Therefore, the informativeness score of $G_{\texttt{HAI}}$ can be measured as

$$Score^{\texttt{inf}}(G_{\texttt{HAI}}) = \frac{1}{\kappa} \sum_{b \in B} (\Theta^b_{G_{\texttt{HAI}}}) \tag{3}$$

As the above representativeness and informativeness scores may not have the same scale, we further use normalization method to scale them into $[0, 1]$. Finally, we introduce a hyper-parameter λ for weighing these two scores, and obtain the final score of $G_{\texttt{HAI}}$ for pipeline selection, *i.e.,*

$$Score(G_{\texttt{HAI}}) = \lambda \cdot Score^{\texttt{rep}}(G_{\texttt{HAI}}) + (1-\lambda) \cdot Score^{\texttt{inf}}(G_{\texttt{HAI}}). \tag{4}$$

Warm-up Pre-training. Although the training samples are carefully selected, it is difficult to train the model from scratch with such a few samples, especially when the learning task is very difficult. As a result, before users train *Alternative Evaluation Network* on their own dataset, we first train it on many other datasets and pipelines, which is called warm-up pre-training in this paper. The

rationale is that, with good parameters as the premise, training (or fine-tuning) with a few samples can also produce good results. Intuitively, warm-up pre-training allows the model to learn general knowledge of operations in pipelines and the datasets. For example, some operations may be more effective than the others regardless of specific ML tasks, or datasets with certain characteristics (*e.g.,* categorical attributes) usually require some operations (*e.g.,* `Encoder`). With the general knowledge, the model would be easier to evaluate HAI-pipelines given a few training samples.

Discussion. Theoretically, an AI-pipeline can be generated by any AutoML tool. Practically, however, many AutoML tools are mainly designed to focus on optimizing model selection and parameters, not data prep, such as Auto-Sklearn [9]. Consequently, combining them with HI-pipelines has marginal improvement, as shown in Section 6.1. Thus, in the next section, we will propose our optimized AI-pipeline generation method, paying particular attention for data prep operations.

5 RL-BASED AI-PIPELINE GENERATION

AI-pipeline generation and optimization aim to find an optimized AI-pipeline G_{A} for a specific task $\mathcal{T}$. We introduce an *iterative* framework that selects operations from a candidate set in multiple iterations, as shown in Figure 7(a). Specifically, we first organize the candidate operations in multiple families, denoted by $\{O_1, O_2, \ldots, O_m\}$, based on their functionalities mentioned in Section 2. Then, in each iteration, we choose a family, say O_i, and further select an operation o_j^i in the family, which is then appended to the current pipeline G_{A}. The above process terminates when reaching a maximum iteration number T (*i.e.,* the maximum length of G_{A}). For simplicity, in this section, we directly use G by omitting the subscript of G_{A}.

Obviously, the key problem in the above process is how to select operations in each iteration, which is very challenging, because the search space for the selection of operations is *exponential* to the number of candidate operations.

To address this, we develop a reinforcement learning (RL) based approach, as shown in Figure 7(b). We formalize the selection process as a sequential decision process by an *agent*. Following a policy π_θ, the agent considers the current pipeline, denoted by G_t, and the dataset D_t produced by G_t as *state*, and performs an *action* that selects an operation to update G_t to G_{t+1}. When the entire pipeline is generated after T iterations, the agent gets a delayed *reward* from the *environment*, based on which the agent updates its policy π_θ. The environment contains essentially execution threads, which run the pipelines, re-train model M and evaluate the test performance $\Phi(G)$.

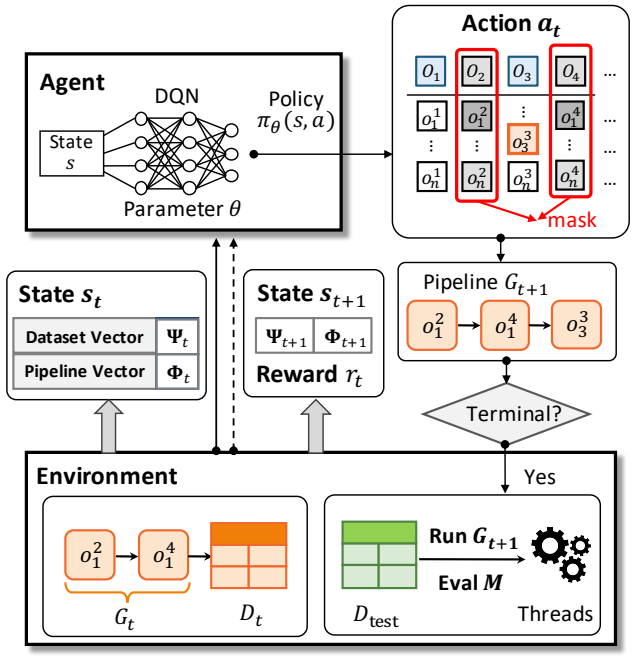
State. We consider the state consisting of dataset features $\mathbf{\Psi}$ and pipeline features $\mathbf{Y}$ for $\mathbf{s}_t$, which are the same as defined in *Alternative Evaluation Network* (see Section 4.2). And we represent the state by concatenating these two kinds of features as $\mathbf{s}_t = \texttt{concat}(\mathbf{\Psi}_t, \mathbf{Y}_t)$.

Action. We consider action a_t as choosing an operation o from a candidate set O, *i.e.,* defining $a_t \in \{o_1, o_2, \ldots, o_{|O|}\}$. To alleviate the aforementioned exponential exploration of the search space, we adopt a *masking* mechanism. The basic idea is that, in each pipeline, we only select the best operation in each family, as operations in the same family have very similar functionalities. Consider the example in Figure 7(b). Given a current pipeline G_t with operations o_1^2 and o_1^4, when deciding action a_t, we mask the families O_2 and O_4 as operations in these families have been already included. Then, the agent can determine a_t from the candidate operations in other families. Suppose that, following its policy π_θ, the agent chooses o_3^3 and then sends it to the environment.

Reward. The reward r_t is used as a signal of whether the actions performed are beneficial. As it may not be indicating to evaluate a partial pipeline due to dependency among operations, we introduce a *delayed* reward. To this end, we introduce a termination signal d_t to indicate whether a current pipeline G_t is complete. We set $d_t = true$ if all operations in O have been selected or

O_1: Imputer	O_2: Encoder	O_3: Scaler	O_4: Feature Transformer	O_5: Feature Selector
Simple Imputer	**OneHot Encoder**	MinMax Scaler	**Polynomial Features**	Select K Best
…	…	…	…	Select FPR
…	Get Dummies	**Standard Scaler**	PCA	…
Drop NA	Label Encoder	Max ABS Scaler	Kernel PCA	**Variance Threshold**

(a) An iterative framework for operation selection.



(b) A reinforcement learning (RL) based solution.

Fig. 7. Overview of AI-pipeline generation and optimization. The operation space is shown in (a), in which each column means an operation family and it includes many operations. Figure (b) shows one step of AI-pipeline optimization. At step t, environment generates s_t with D_t and G_t, which is the input of policy function π. Then agent selects an operation O_3^3 in $\{O_1, O_3, O_5\}$ because other two families are selected before. Then agent sends it to environment. Next, environment executes this operation, generates new state s_{t+1} and sends a reward r_t, which is then used to optimize the weight of π.

masked; otherwise $d_t = false$. Based on this, we define the reward as $r_t = 0$ if $d_t = false$; otherwise $r_t = \Phi(G_t)$ where $\Phi(G_t)$ is the performance of pipeline G_t.

Deep Q-Network (DQN). We adopt the Deep Q-Network (DQN) framework [19], where the agent iteratively learns to perform actions by interacting with the environment. Specifically, the agent takes the state $\mathbf{s}_t$ at each iteration, chooses an action a_t according to policy π_θ, and observes a

Algorithm 2: AI-Pipeline Generation and Optimization

Require: Environment $\mathbb{E}$
Ensure: action-value function Q
Initialize replay memory $\mathbb{D}$.
Initialize Q with random weights.
for each *episode* $\in [1, M]$ **do**
  Initialize $\mathbb{E}$ with a new task, and get dataset features $\mathbf{\Psi}_1$ and pipeline features $\mathbf{Y}_1$.
  Initialize an operation set O.
  $\mathbf{s}_1 = \texttt{concat}(\mathbf{\Psi}_1, \mathbf{Y}_1)$
  $d_1 = false$
  for each $i \in [1, T]$ **do**
    if $d_i = true$ **then**
      Initialize $\mathbb{E}$ with a new task, and get dataset features $\mathbf{\Psi}_t$ and pipeline features $\mathbf{Y}_t$.
      Initialize a operation set O.
    end if
    Select operation a_t with the $\epsilon-$greedy policy.
    Mask operations that is in a_t's family from O.
    if O is empty **then**
      $d_t = true$
    else
      $d_t = false$
    end if
    Environment $\mathbb{E}$ executes a_t and returns new features $\mathbf{\Psi}_{t+1}$ and $\mathbf{Y}_{t+1}$, and reward r_t.
    $\mathbf{s}_{t+1} = \texttt{concat}(\mathbf{\Psi}_{t+1}, \mathbf{Y}_{t+1})$
    Store transition $(\mathbf{s}_t, a_t, r_t, \mathbf{s}_{t+1}, d_t)$ in $\mathbb{D}$.
    Randomly sample minibatch of transitions $\{(\mathbf{s}_j, a_j, r_j, \mathbf{s}_{j+1}, d_j)\}$ from $\mathbb{D}$.
    Perform a gradient descent step on $(y_j - Q(\mathbf{s}_j, a_j; \theta))$ and update θ, where $y_j = r_j$ if $d_j = true$, otherwise $y_j = r_j + \gamma max_a Q(\mathbf{s}_{j+1}, a; \theta)$.
  end for
end for

reward r_t from the environment. In DQN, learning policy π_θ is to learn a value-function $Q(\mathbf{s}, a; \theta)$ that produces a value for a state $\mathbf{s}$ and an action a. Following the typical strategy in DQN, we adopt a neural network to approximate $Q(\mathbf{s}, a; \theta)$, where the neural network contains multiple fully connected layers with `LeakeyRelu` as activation function, and an output `tanh` layer.

DQN training and inference. We use the typical training algorithm [19] to train the DQN model. The pseudo-code of the training algorithm is shown in Algorithm 2. The algorithm uses an off-policy strategy that learns an ϵ-greedy policy in multiple episodes. In each episode, it selects the greedy action $a = \arg\max_a Q(\mathbf{s}, a; \theta)$ with a probability $1 - \epsilon$, and a random action with probability ϵ. Then, the algorithm uses a replay memory $\mathbb{D}$ to store the existing experiences of the agent, *i.e.,* $\{(\mathbf{s}_t, a_t, r_t, \mathbf{s}_{t+1}, d_t)\}$. After that, it samples a batch of $(\mathbf{s}_j, a_j, r_j, \mathbf{s}_{j+1}, d_j)$ from the replay memory and updates parameters θ in $Q(\mathbf{s}, a; \theta)$ using stochastic gradient descent.

For inference, given a new task $\mathcal{T}$, we extract the aforementioned features to represent states, and use the trained model (i.e. $Q(\mathbf{s}, a; \theta)$) to select the most appropriate actions ($a = \arg\max_a Q(\mathbf{s}, a; \theta)$) step by step without updating parameters θ. Our RL-based approach has good generalization ability, as demonstrated in our experiments.

Table 1. Dataset statistics, where SVM, KNN, LR (Logistic Regression) and RF (Random Forest) are downstreaming machine learning models used for evaluation.

ML model	# Task	# Pipeline	Avg # Col	Avg # Row
SVM	73	226	31.7	4632.5
KNN	109	235	15.0	7366.1
LR	135	477	14.7	9337.0
RF	145	480	38.2	12345.1
All	462	1418	24.9	9072.8

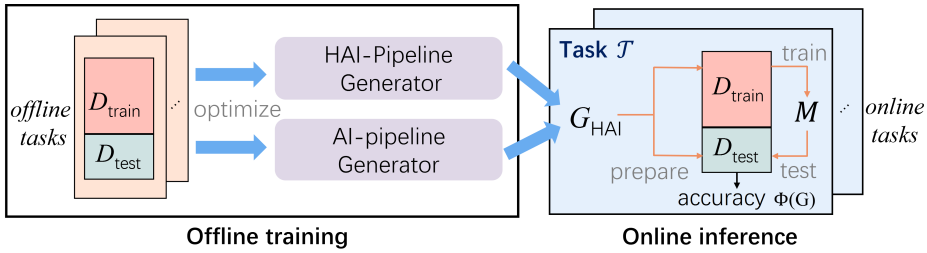


Fig. 8. An illustration of the evaluation method.

6 EXPERIMENTS

Dataset. We use real-world data prep pipelines written by data scientists in Kaggle [14]. Specifically, we extract the Jupyter notebooks written in Python collected by the KGTorrent Repository [25]. As these Jupyter notebooks may have a variety of purposes, we only use the ones that train *classification* models over *tabular* datasets. We also remove the Jupyter notebooks with small datasets, as their results may be insufficient for evaluation. Based on this process, we obtain 462 datasets with 1418 HI-pipelines. Table 1 provides the statistics of these tasks and pipelines by various ML models.

Evaluation method. We implement our HAIPipe framework and evaluate the performance by splitting the tasks, as shown in Table 1, into *offline tasks* and *online tasks* with the ratio of 2:1. The evaluation process is illustrated in Figure 8. We utilize the *offline tasks* to optimize our HAI-pipeline generator (Section 4) and the RL-based AI-pipeline generator (Section 5) in the offline stage. Then, we use the *online tasks* for online inference. In particular, each task $\mathcal{T}$ in *online tasks* contains a training dataset $D_{\texttt{train}}$, test dataset $D_{\texttt{test}}$ and an ML model M. Given the task $\mathcal{T}$, the online inference process first leverages the AI-pipeline generator to generate an optimized AI-pipeline G_{A} and then combines G_{A} with the HI-pipeline G_{H} using HAI-pipeline generator to generate an HAI-pipeline $G_{\texttt{HAI}}$. After $G_{\texttt{HAI}}$ is generated, it prepares $D_{\texttt{train}}$ and $D_{\texttt{test}}$. Finally, we train the ML model M on the prepared training data $D_{\texttt{train}}$ and use the test dataset $D_{\texttt{test}}$ to evaluate its holdout test performance $\Phi(G)$.

Evaluation metric. As mentioned in Section 2, we use the classification accuracy on the test dataset to evaluate the performance of an individual pipeline G, *i.e.,* $\texttt{accuracy} = \texttt{avg}_G(\Phi(G))$.

Hyper-parameter settings. (1) We implement all the operations in AI-pipelines using Scikit-Learn with their hyper-parameters set as default. (2) For implementing dataset features Ψ_t in Section 5, we extract a 19-dimensional vector for each column in the dataset, including the statistical information mentioned in Section 5. We consider a dataset D_t has 100 columns, and for the ones with less columns, we use the padding techniques to add zero values. (3) For implementing pipeline embedding

Table 2. Evaluation on the effect of combining HI- and AI-pipelines, where each column shows the average accuracy for a user-specific downstream ML model type.

Approach	ALL	SVM	KNN	LR	RF
HI-pipeline	0.847	0.813	0.812	0.858	0.865
AI-pipeline	0.816	0.837	0.846	0.820	0.791
HAI-pipeline	**0.862**	**0.862**	**0.851**	**0.864**	**0.865**

Table 3. Average runtime breakdown for generating HAI-pipelines in HAIPipe (in seconds). Note that the runtime of the fourth step, *i.e.,*HAI-program generation is much smaller than that of other steps and thus is omitted in the table.

Step in HAIPipe	ALL	SVM	KNN	LR	RF
HI-program Parsing	0.12	0.12	0.11	0.11	0.13
AI-pipeline Generation	18.38	19.23	16.03	18.20	19.12
HAI-pipeline Generation	12.18	11.59	13.17	11.61	12.58
End-to-End	30.68	30.94	29.31	29.92	31.83

Φ_t, we use a 30-dimensional embedding layer to encode operations, and the dimension of the LSTM hidden layer is 18. (4) For implementing the neural networks in DQN, we use 11 fully connected (FC) layers with `LeakyReLU` as activation function, except that the last FC layer is followed by activation function `tanh`. For training DQN, we set the learning rate as 0.00001, the minimum value of ϵ in $\epsilon-$greedy algorithm as 0.4, the max size of replay memory as 2000, and the batch size in each training iteration as 100. We train the agent with 56000 iterations. (5) For the XG-Boost classification model, which will be described in Section 6.2, we set the parameter `mean_child_weight` to 2, `gamma` to 0.1, `subsample` to 0.8 and `colsample_bytree` to 0.8. For training binary classification model in an downstream ML task, we set the parameter `n_estimators` to 100, `max_depth` to 10, and `learning_rate` to 0.05. For training multiple classification, we set parameter `n_estimators` to 200, `max_depth` to 5 and `learning_rate` to 0.1.

Our experiment environment is Python 3.8.12 on Unbuntu 20.04. The versions of Scikit-Learn and Pandas [24] are respectively 0.23.2 and 1.3.3. We train our model and execute inference with a 20-core Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz.

6.1 Evaluation on HAIPipe

This section first compares the HAI-pipelines produced by HAIPipe with HI-pipelines and AI-pipelines to evaluate the effect of combining HI- and AI-pipelines (**Exp-1**). Then, we investigate the effectiveness of our proposed RL-based AI-pipeline generation (**Exp-2**), and evaluate our best pipeline selection strategy for HAI-pipeline generation (**Exp-3**). In particular, for a fair comparison, we directly use the downstream ML models (*e.g.,* a classification model) specified by the data scientists in the Jupyter notebooks.

Exp-1: Whether the HAI-pipelines generated by HAIPipe outperform both HI- and AI-pipelines? In this experiment, given an ML task, we compare each **HAI-pipeline** generated by our HAIPipe with the following two alternatives: (1) **HI-pipeline** that is directly parsed from the corresponding Jupyter notebook, and (2) **AI-pipeline** that is generated by our RL-based approach from the corresponding dataset.

The result is reported in Table 2. We can clearly see that **HAI-pipeline** outperforms both **HI-pipeline** and **AI-pipeline** for all the user-specific downstream ML models. For example, for the SVM model, **HAI-pipeline** achieves 5% and 3% improvements on accuracy compared with **HI-pipeline** and **AI-pipeline** respectively. We can observe similar improvements for the other ML models, KNN and LR. We also find the improvement is less significant for the tree-based ML model RF. This is because RF has their own mechanisms for feature selection, *i.e.,* the feature importance in RF, which may mitigate the effect of some operations, *e.g.,* `FeatureSelection`.

Conclusion: Based on the results, we can conclude that combination of HI- and AI-pipelines for data preparation is very helpful.

Moreover, we report the average runtime breakdown for generating an HAI-pipeline in HAIPipe, as shown in Table 3. We have the following observations. First, the average end-to-end runtime used to generate an HAI-pipeline is in *seconds*, which means that HAIPipe is user-friendly to meet data prep requirements. Second, we find that AI-pipeline generation and HAI-pipeline generation take most of the time. This is because these two steps need to perform the DQN inference and the active learning strategy respectively.

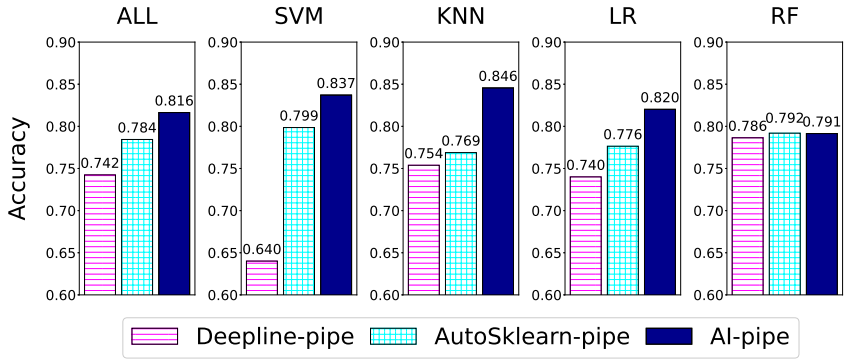
Exp-2: How effective is our AI-pipelines generated by RL? This experiment investigates whether our RL-based approach, denoted by **AI-pipe**, can effectively explore the space to produce high-quality AI-pipelines. We compare our RL model with the following alternative methods for AI-pipeline generation. (1) **Deepline-pipe** uses an alternative RL-based strategy from Deepline [12] that pre-defines the pipeline structure, which may narrow the search space to find optimized pipelines. Note that this baseline also considers the user-specific models, like our RL-based strategy. (2) **AutoSklearn-pipe** leverages the data prep pipelines generated by Auto-Sklearn [9], while using the user-specific models for fair comparison.

We first directly compare the AI-pipelines generated by the above approaches, and report the results in Figure 9(a). We can see that our RL-based approach **AI-pipe** outperforms **Deepline-pipe** and **AutoSklearn-pipe** in almost all the cases. For example, **AI-pipe** respectively improves the average accuracy across all models by 0.07 and 0.03 compared with **Deepline-pipe** and **AutoSklearn-pipe**. This is mainly attributed to our proposed AI-pipeline generation and optimization algorithm. In contrast, **Deepline-pipe** fixes the ordering of operations families, which may miss some opportunities for optimization. Auto-Sklearn pays more attention to model selection and parameter tuning, and thus the data prep pipelines generated by **AutoSklearn-pipe** are relatively simple. Also, we find that the improvement is more obvious in SVM, KNN and LR. For RF model, **AI-pipe** is almost the same as others because some data prep operations, such as scaling and feature transforming, have less significant effect on RF.

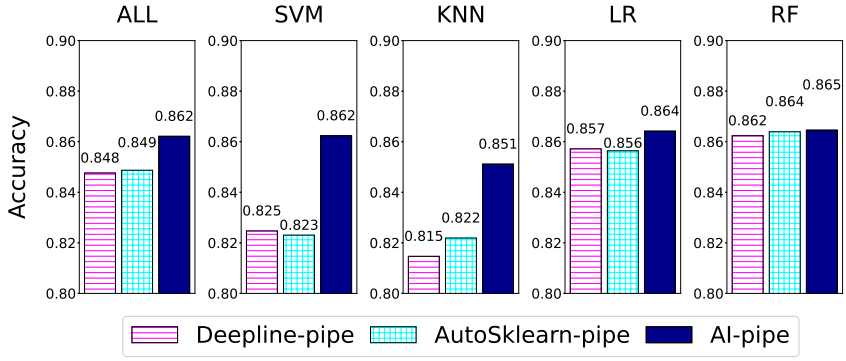
To make the comparison more comprehensive, we further indirectly compare various AI-pipelines by considering the HAI-pipelines generated by them. The purpose of the comparison is to investigate whether HAIPipe also performs well by combining AI-pipelines produced by alternative approaches **Deepline-pipe** and **AutoSklearn-pipe** with the same HI-pipelines. The experimental result is shown in Figure 9(b). We can see that **AI-pipe** still also outperforms the other two alternatives (0.862 vs. 0.848 and 0.849), which further validates the effectiveness of our RL-based approach.

Conclusion: These results show that our reinforcement learning based algorithm can effectively generate high-quality AI-pipelines.

Exp-3: How effective is our best pipeline selection? To analyze the effectiveness of our strategy for best pipeline selection (see Section 4.2), we compare our strategy with two baselines: (1) **Random** selects k pipelines to evaluate on validation dataset and selects the best as result. (2) **Heuristic** employs a set of manually defined rules to prune invalid pipelines to reduce the search space. Then, it randomly selects k pipelines from the reduced search space, evaluates them, and chooses the best



(a) Directly evaluating AI-pipelines generated by various approaches.



(b) Indirectly evaluating AI-pipelines by considering their generated HAI-pipelines.

Fig. 9. Evaluation on AI-pipeline generation.

one. These rules are specific to different data preparation operations. Specifically, users make some constraints according to operations characteristics to eliminate invalid or redundant pipelines. For example, `sklearn.StandardScaler` must exist after `pd.get_dummies` because scaler cannot deal with the categorical features with string type.

Figure 10 shows that HAIPipe is better than both **Random** and **Heuristic** strategies. **Random** is the worst because the strategy evaluates many invalid pipelines, which are unnecessary for selecting the best one. **Heuristic** can prune unnecessary pipelines, which leads to better results than **Random**. However, **Heuristic** may not be able to select the best pipeline in the pruned sets, as it is difficult for human to directly judge the performance of pipeline with pre-defined heuristic rules. Therefore, when k increases, **Heuristic** quickly reaches a value (0.850) and cannot be improved further.

Conclusion: Based on the results, we can conclude that the strategy of HAIPipe can effectively select pipelines with a limited budget, and performs well for selecting best HAI-pipelines.

6.2 Comparison with AutoML

This section compares HAIPipe with the existing AutoML approaches that both generate data prep process, select ML models and optimize hyper-parameters (**Exp-4**). Specifically, for a fair comparison, given an ML task $\mathcal{T}$ with a dataset D, we do not constrain the AutoML methods and

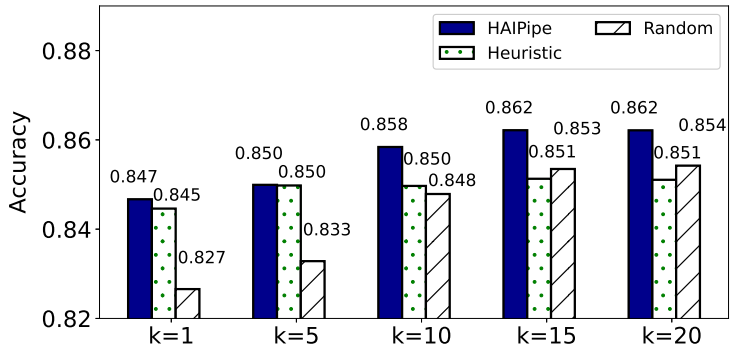


Fig. 10. Comparison of HAIPipe with alternative sampling strategies under different evaluation budget k.

let them freely conduct model construction, hyper-parameter selection, and data prep pipeline generation. Similarly, for HAIPipe, we do not constrain the downstream ML model to be the user-specific models from the corresponding Jupyter Notebooks. Instead, we set the downstream ML model of HAIPipe as a fixed model XG-Boost with pre-defined hyper-parameters (see more details in the "Hyper-parameter settings" above).

Exp-4: How does HAIPipe perform compared with AutoML? We compare HAIPipe with the following AutoML methods. (1) **AutoSklearn** utilizes the well-adopted Auto-Sklearn [9], which is the most popular AutoML Python package. In particular, we set the hyper-parameters of Auto-Sklearn `time_left_for_this_task` to 360, `per_run_time_limit` to 600 and `memory_limit` to 100000. Note that the time limit is set as 10 minutes, which is the same time constraint for HAIPipe, to fairly compare HAIPipe and **AutoSklearn**. (2) **Deepline** [12], except for the ability of automatic pipeline generation mentioned above, also supports model selection and hyper-parameter tuning, using reinforcement learning. We use the source code provided by the original paper of **Deepline** to implement the method, and set the hyper-parameters as default.

For better evaluation, we also investigate an "HI-pipeline + AutoML" setting, which applies an AutoML method in a human-generated pipelines (*i.e.,* HI-pipelines). Note that this setting reflects what many data scientists would use in practice, *i.e.,* writing an HI-pipeline first and directly applying an AutoML method. Based on this, we consider the following two baselines. (3) **HI+AutoSklearn** first applies an HI-pipeline to generate a prepared dataset, and then utilizes **AutoSklearn** for the remaining work. (4) **HI+Deepline** is similar to **HI+AutoSklearn**, except that it uses **Deepline** as the AutoML method.

Figure 11 shows the experimental result. The overall accuracy scores of HAIPipe, **AutoSklearn** and **Deepline** are respectively 0.876, 0.855 and 0.752, which shows that HAIPipe significantly outperforms **AutoSklearn** and **Deepline**. Specifically, compared with **AutoSklearn**, HAIPipe performs better than it on above half tasks. Compared with **Deepline**, HAIPipe performs better than it on most tasks. For those failed cases, the differences between HAIPipe and both approaches are not significant, *e.g.,* most of the red points lying near the diagonal line, which means that the approaches have nearly the same accuracy scores.

Not surprisingly, equipped with HI-pipelines, both AutoML methods gain improvements, *i.e.,* 0.855 (**AutoSklearn**) vs. 0.862 (**HI+AutoSklearn**) and 0.752 (**Deepline**) vs. 0.827 (**HI+Deepline**). This result shows that directly combining HI-pipelines and AutoML method is helpful to improve the overall performance. However, our proposed HAIPipe still achieves the highest accuracy (0.876).

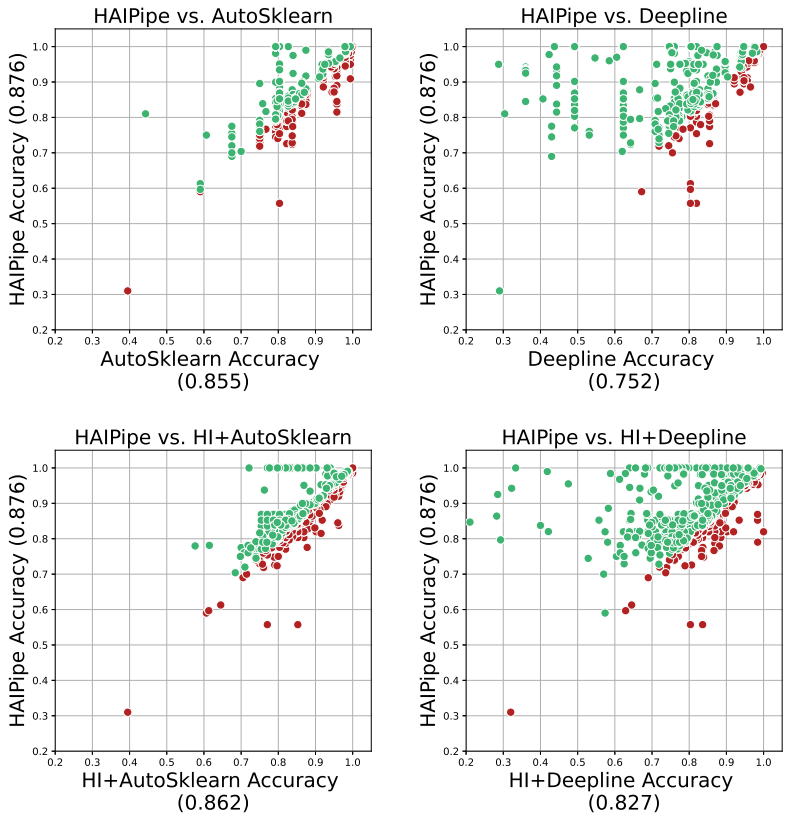


Fig. 11. Comparing HAIPipe with AutoML methods. Each sub-figure shows a pair-wise comparison of two approaches, where the green (or red) points means HAIPipe performs better (or worse) than a baseline.

This is mainly attributed to our AI-pipeline generation and best HAI-pipeline selection methods, which may further capture task-specific dependencies among data prep operations.

Conclusion: Based on the results, we can conclude that HAIPipe performs better than the existing AutoML methods on data prep pipeline generation. On the other hand, it is very likely that these two kinds of methods can complement each other.

6.3 Comparison with Interactive Method

Exp-5: How does HAIPipe perform compared with interactive recommendation-based methods, such as Copilot [6]?

This section compares HAIPipe with **Copilot** [6], a recent next-step code recommendation method from GitHub, which can also be applied to data prep pipeline generation. Given a dataset D, different from HAIPipe, **Copilot** requires online interaction of users. More specifically, it interacts with a user to suggest the next lines of code based on the context the users is working in, and the use can choose the code or not. Moreover, the user can control the recommendation process by writing natural language in comments, and **Copilot** can then suggest the code according to the comments. To fulfill the online interaction, we randomly select 10 tasks from our dataset (see Table 1), and employ 10 volunteers with data science programming experiences to use **Copilot** to complete the

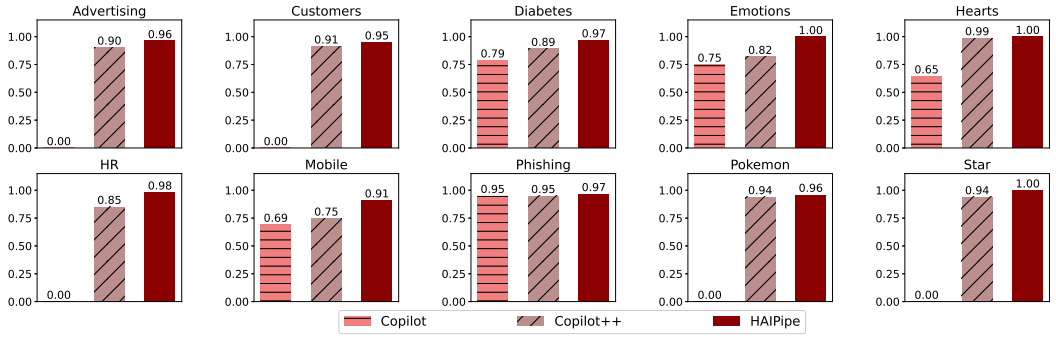


Fig. 12. Comparison of HAIPipe with interactive pipeline recommendation method Copilot.

```
# deal outliers
X = X.clip(lower=X.quantile(0.01), upper=X.quantile(0.99), axis=1)
```

(a) Outlier removal code suggested by **Copilot++**.

```
# deal outliers
columns_nozero_values = \
['Glucose','BloodPressure', 'SkinThickness','Insulin','BMI']
for n in columns_nozero_values:
    data[n] = data[n].replace(0,np.NaN)
    mean = int(data[n].mean())
    data[n] = data[n].replace(np.NaN,mean)
```

(b) Outlier removal code in HI-program written by users.

Fig. 13. In-depth analysis of Copilot for the `Diabetes` task.

tasks. We consider the following two settings. (1) **Copilot** iteratively suggests candidates for the next lines of code based on the context. In each iteration, the user can choose from the candidates and proceed to the next iteration, or decide to stop the recommendation process. (2) **Copilot++** enables the user to modify or rewrite the program after Copilot giving a suggestion.

As reported in Figure 12, the experimental result shows that HAIPipe outperforms **Copilot** and **Copilot++** in all tasks. To further analyze the reason, we provide an in-depth performance analysis for Copilot. We first examine the pairwise duplication of the code generated by **Copilot**, by using the well-known code checker MOSS [21], and find the average duplication rate is 67.5%. Moreover, when only considering the data prep code (*i.e.,* removing model training, dataset loading, comments, etc.), we find the duplication rate is nearly 100%. This result reveals that Copilot, which is trained from software documents and public code repositories, suggests the next-step operation that is *relevant* to the current code context, without considering the characteristics of dataset D in the task. Consequently, this would cause two key problems that affect the overall performance, namely *low success rate* and *low accuracy*, as described as below.

(1) Low success rate: We have an important observation that many pipelines generated by **Copilot** incur runtime errors during execution, and thus fail to generate the prepared datasets. For example, the average accuracy of **Copilot** is 0 for tasks `Advertising`, `Customers`, `HR`, `Pokemon` and `Star`, because all the pipelines generated by **Copilot** have failed. The main reason is that **Copilot** does

Table 4. Operation usage ratios in HI- and HAI-pipelines.

Operation	Usage in HI-pipelines	Usage in HAI-pipelines
StandardScaler	36.52%	37.01%
MinMaxScaler	6.99%	19.61%
RobustScaler	0.74%	7.11%
QuantileTransformer	0.00%	3.80%
Normalizer	0.12%	6.62%
MaxAbsScaler	0	39.98%
PowerTransformer	0.12%	10.42%
KBinsDiscretizer	0.12%	6.13%
PCA	3.43%	12.50%
PolynomialFeatures	0.61%	25.98%
IncrementalPCA	0	0
TruncatedSVD	0.12%	1.47%
KernelPCA	0	3.68%
RandomTreesEmbedding	0	0
VarianceThreshold	0.09%	22.06%

not suggest encoding schemes for the *categorical* columns in these datasets, leading to runtime errors in model training. In the other tasks, there may also be some failure pipelines caused by the ignorance of dataset characteristics.

(2) Low accuracy: By observing the tasks where Copilot's pipelines run successfully (*e.g.,*`Diabetes`, `Mobile` and `Phishing`), we find that HAIPipe still outperforms **Copilot** and **Copilot++**. The main reason is that Copilot cannot inject *dataset-specific* domain knowledge into data prep. Figure 13 shows an example to analyze **Copilot++** in the `Diabetes` task. As observed in Figure 13(a), **Copilot++** suggests a very common operation for outlier removal, *i.e.,* removing values outside specific quantiles, which may be learned from previous software documents and public code repositories. In contrast, as shown in Figure 13(b), HAIPipe can utilize *dataset-specific* outlier removal operations included in HI-program, *e.g.,* the appropriate range of medical features according to user's domain knowledge, which leads to better data prep performance.

Moreover, according to the feedback of the volunteers, they find difficulties in choosing the candidates suggested by Copilot. This is because, before the entire pipeline is completed and the model is trained, the volunteers do not know how the current intermediate operations will affect the final result. This makes the suggestions less relevant to the specific dataset and ML task.

Conclusion: Based on the results, we conclude that, although Copilot, the interactive code recommendation tool, has a wide range of applications and perform well in some scenarios (such as reproduction of common functions), in the scenario of data prep for ML tasks, the performance of Copilot is still limited, *e.g.,* having problems of low success rate and low accuracy.

6.4 A Case Study

Exp-6: How does the HAI-pipeline improve the overall performance of data prep? This section provides a case study to explain why HAIPipe can improve data prep by combining HI- and AI-pipelines.

We first analyze the usage of different operations in the HI-pipelines, which is reported in the "Usage in HI-pipelines" of Table 4. We have an interesting observation that most operations provided

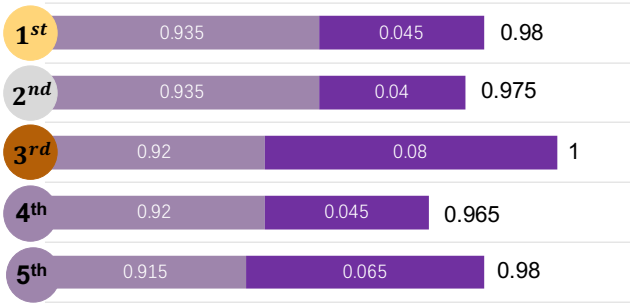


Fig. 14. An example task for advertising click prediction, where accuracy of top 5 notebooks is reported. The light part is the accuracy of HI-Pipeline, while the dark part is the improvement brought by HAIPipe.

by Sklearn [27] are not well acquainted by the data scientists in Kaggle. Specifically, only a few operations, such as `StandardScaler` and `get_dummies`, are commonly used (*i.e.,* the usage ratio is above 30%). Many sophisticated operations, such as `PolynomialFeatures` and `VarianceThreshold` are rarely considered. On the contrary, after smartly combined with AI-pipelines by HAIPipe, the usage of some sophisticated operations, such as `MaxAbsScaler`, `PolynomialFeatures` and `VarianceThreshold` are increased by 1-2 orders of magnitude. The result shows that combining HI- and AI-pipelines can complement with HI-pipelines by including more less popular yet very useful operations.

Next, we demonstrate the benefit of combining these sophisticated operations by using an example task for advertising click prediction. The goal of the task is to predict whether a user is likely to click on an advertisement based on one's browsing information, such as daily time spent on site, demographic information, etc. As shown in Figure 14, the light bars represent accuracy scores of the top-5 HI-pipelines in the task, which are sorted in descending order of accuracy, while the dark bars denote the improvements brought by HAIPipe. We have the following observations. First, every pipeline is improved by HAIPipe; most of them obtain higher ranks compared with the original HI-pipelines. For example, if we only apply HAIPipe to enhance the 5th HI-pipeline from 0.915 to 0.980 while keeping other HI-pipelines unchanged, this pipeline will become the best one. Second, we provide an in-depth analysis for the 3rd pipeline, which gains significant accuracy boosting (from 0.92 to 1.0). This HI-pipeline leverages the domain knowledge to divide the *Timestamp* into buckets according to the work and rest time of human, namely morning, noon, afternoon, evening and midnight. This makes continuous features become categorical features, while including more semantics. In summary, this example demonstrates the benefits of combining HI- and AI-pipelines.

7 RELATED WORK

Pipeline generation has been extensively studied [7, 8, 12, 18, 23, 28, 34] for different applications. We categorize them as follows.

Application-specific pipeline generation. Auto-Suggest [34] recommends the next data transformation operations, such as Join, Pivot, Groupby, and Relationalize JSON, which are mainly used for data/schema normalization. Auto-Pipeline [36] extends Auto-Suggest by generating the full pipeline that transforms an input table to a user-specified "target" table. This by-target paradigm is mainly for applications that the user knows the desired target schema, such as generating BI dashboards. React [18] is used to recommend interactive data analysis (IDA) pipelines. Besides these

three works, there are also some works focus on other operation types, such as entity matching and entity alignment [32, 33, 37].

None of the above methods is designed for machine learning data prep pipelines. That is, the data prep operations discussed in Section 2 for ML are not considered by the prior art.

Generic pipeline generation. This research direction learns from a large collection of code, such that it can assist users for "any" code generation. The state-of-the-art tool for this purpose is GitHub Copilot [6] from GitHub and OpenAI, which auto-generates the next lines of code based on what it has learned from public code repositories on GitHub. Also, Copilot is recommendation-based, which interacts with the user to complete the code.

A main shortcoming of Copilot for ML pipeline is that Copilot neither profiles the dataset nor sees the downstream ML model. In other words, it is extremely hard for Copilot to decide whether a feature is good or which data prep operations should be used, making the recommendation less relevant to the specific dataset and ML task. We have empirically verified this in Section 6.3.

AutoML for ML pipeline generation. The goal of AutoML is to automate the entire life cycle of ML. Most AutoML methods focus on model construction and hyper-parameter selection [1, 4, 10, 11, 15–17, 29, 30, 38]. Auto-Weka [31] and Auto-Sklearn [9] further consider data prep pipeline generation. Learn2clean [2] and its extension [3] are pioneering works of using reinforcement learning for generating data prep pipelines for ML. They use heuristic methods to constrain the orders of operations of pre-defined categories (for example, normalization should be executed before imputation), which considers a smaller search space than our approach (see Section 5). Deepline [12] is a recent work similar to Learn2clean, which is more effective and efficient, but it still considers a smaller search space than our approach. Alpine Meadow [28] proposes an exploitation-exploration solution for searching pipelines. TPOT [23] solves the pipeline optimization problem using genetic programming.

Different from them, the main goal of HAIPipe is to combine HI-pipelines and AI-pipelines to achieve better performance (see Section 6.1 **Exp-1**). Therefore, these AutoML techniques are complementary to the HAIPipe framework.

8 CONCLUSION

We have introduced a novel framework HAIPipe that combines a human-generated pipeline (HI-pipeline) and an AI-pipeline to maximize the performance for an ML task. We have also proposed a reinforcement learning based approach to search an optimized AI-pipeline and adopted an enumeration-sampling strategy to carefully select the best performing combined pipeline. We have conducted experiments on real-world data prep pipelines from Kaggle and the experimental results show that HAIPipe can significantly outperform both the HI-pipelines and AI-pipelines.

ACKNOWLEDGMENTS

This work was partly supported by the National Natural Science Foundation of China (62122090, 62072461, 62072458, 62232009, 61925205 and 62102215), National Key Research and Development Program of China (2020YFB2104101), Huawei, TAL education, Beijing National Research Center for Information Science and Technology (BNRist), the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China (23XNH001).

REFERENCES

[1] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.* 13 (2012), 281–305. http://dl.acm.org/citation.cfm?id=2188395

[2] Laure Berti-Équille. 2019. Learn2Clean: Optimizing the Sequence of Tasks for Web Data Preparation. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 2580–2586. https://doi.org/10.1145/3308558.3313602

[3] Laure Berti-Équille. 2019. Reinforcement Learning for Data Preparation with Active Reward Learning. In *Internet Science - 6th International Conference, INSCI 2019, Perpignan, France, December 2-5, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11938)*, Samira El Yacoubi, Franco Bagnoli, and Giovanna Pacini (Eds.). Springer, 121–132. https://doi.org/10.1007/978-3-030-34770-3_10

[4] Carsten Binnig, Benedetto Buratti, Yeounoh Chung, Cyrus Cousins, Tim Kraska, Zeyuan Shang, Eli Upfal, Robert C. Zeleznik, and Emanuel Zgraggen. 2018. Towards Interactive Curation & Automatic Tuning of ML Pipelines. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, Sebastian Schelter, Stephan Seufert, and Arun Kumar (Eds.). ACM, 1:1–1:4. https://doi.org/10.1145/3209889.3209891

[5] Wenbin Cai, Ya Zhang, and Jun Zhou. 2013. Maximizing Expected Model Change for Active Learning in Regression. In *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*, Hui Xiong, George Karypis, Bhavani Thuraisingham, Diane J. Cook, and Xindong Wu (Eds.). IEEE Computer Society, 51–60. https://doi.org/10.1109/ICDM.2013.104

[6] Copilot. [n. d.]. https://github.com/features/copilot.

[7] Iddo Drori, Yamuna Krishnamurthy, Rémi Rampin, Raoni de Paula Lourenço, Jorge Piazentin Ono, Kyunghyun Cho, Cláudio T. Silva, and Juliana Freire. 2021. AlphaD3M: Machine Learning Pipeline Synthesis. *CoRR* abs/2111.02508 (2021). arXiv:2111.02508 https://arxiv.org/abs/2111.02508

[8] Ori Bar El, Tova Milo, and Amit Somech. 2020. Automatically Generating Data Exploration Sessions Using Deep Reinforcement Learning. In *SIGMOD*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1527–1537. https://doi.org/10.1145/3318464.3389779

[9] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. Auto-sklearn: Efficient and Robust Automated Machine Learning. In *Automated Machine Learning - Methods, Systems, Challenges*, Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). Springer, 113–134. https://doi.org/10.1007/978-3-030-05318-5_6

[10] Nicoló Fusi, Rishit Sheth, and Melih Elibol. 2018. Probabilistic Matrix Factorization for Automated Machine Learning. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 3352–3361. https://proceedings.neurips.cc/paper/2018/hash/b59a51a3c0bf9c5228fde841714f523a-Abstract.html

[11] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 1487–1495. https://doi.org/10.1145/3097983.3098043

[12] Yuval Heffetz, Roman Vainshtein, Gilad Katz, and Lior Rokach. 2020. DeepLine: AutoML Tool for Pipelines Generation using Deep Reinforcement Learning and Hierarchical Actions Filtering. In *KDD*, Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash (Eds.). ACM, 2103–2113. https://dl.acm.org/doi/10.1145/3394486.3403261

[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[14] Kaggle. [n. d.]. https://www.kaggle.com/.

[15] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.* 18 (2017), 185:1–185:52. http://jmlr.org/papers/v18/16-558.html

[16] Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. 2018. Ease.ml: Towards Multi-tenant Resource Sharing for Machine Learning Workloads. *Proc. VLDB Endow.* 11, 5 (2018), 607–620. https://doi.org/10.1145/3187009.3177737

[17] Gang Luo. 2016. A review of automatic selection methods for machine learning algorithms and hyper-parameter values. *Netw. Model. Anal. Health Informatics Bioinform.* 5, 1 (2016), 18. https://doi.org/10.1007/s13721-016-0125-6

[18] Tova Milo and Amit Somech. 2018. Next-Step Suggestions for Modern Interactive Data Analysis Platforms. In *KDD*, Yike Guo and Faisal Farooq (Eds.). ACM, 576–585. https://doi.org/10.1145/3219819.3219848

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[20] Pyhon AST Module. [n. d.]. https://docs.python.org/3/library/ast.html.
[21] Moss. [n. d.]. http://theory.stanford.edu/ aiken/moss/.
[22] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. 2020. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *KDD*, Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash (Eds.). ACM, 1542–1551. https://doi.org/10.1145/3394486.3403205
[23] Randal S. Olson and Jason H. Moore. 2019. TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning. In *Automated Machine Learning - Methods, Systems, Challenges*, Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). Springer, 151–160. https://doi.org/10.1007/978-3-030-05318-5_8
[24] Pandas. [n. d.]. https://pandas.pydata.org/.
[25] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2021. KGTorrent: A Dataset of Python Jupyter Notebooks from Kaggle. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 550–554. https://doi.org/10.1109/MSR52588.2021.00072
[26] Shubhangi Vashisth Rita Sallam, Ehtisham Zaidi. 2017. Market guide for data preparation.
[27] Scikit-Learn. [n. d.]. https://scikit-learn.org/stable/.
[28] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing Data Science through Interactive Curation of ML Pipelines. In *SIGMOD*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1171–1188. https://doi.org/10.1145/3299869.3319863
[29] Evan R. Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. 2015. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, Shahram Ghandeharizadeh, Sumita Barahmand, Magdalena Balazinska, and Michael J. Freedman (Eds.). ACM, 368–380. https://doi.org/10.1145/2806777.2806945
[30] Thomas Swearingen, Will Drevo, Bennett Cyphers, Alfredo Cuesta-Infante, Arun Ross, and Kalyan Veeramachaneni. 2017. ATM: A distributed, collaborative, scalable system for automated machine learning. In *2017 IEEE International Conference on Big Data (IEEE BigData 2017), Boston, MA, USA, December 11-14, 2017*, Jian-Yun Nie, Zoran Obradovic, Toyotaro Suzumura, Rumi Ghosh, Raghunath Nambiar, Chonggang Wang, Hui Zang, Ricardo Baeza-Yates, Xiaohua Hu, Jeremy Kepner, Alfredo Cuzzocrea, Jian Tang, and Masashi Toyoda (Eds.). IEEE Computer Society, 151–162. https://doi.org/10.1109/BigData.2017.8257923
[31] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2012. Auto-WEKA: Automated Selection and Hyper-Parameter Optimization of Classification Algorithms. *CoRR* abs/1208.3719 (2012). arXiv:1208.3719
[32] Jianhong Tu, Ju Fan, Nan Tang, Peng Wang, Chengliang Chai, Guoliang Li, Ruixue Fan, and Xiaoyong Du. 2022. Domain Adaptation for Deep Entity Resolution. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 443–457. https://doi.org/10.1145/3514221.3517870
[33] Jianhong Tu, Xiaoyue Han, Ju Fan, Nan Tang, Chengliang Chai, Guoliang Li, and Xiaoyong Du. 2022. DADER: Hands-Off Entity Resolution with Domain Adaptation. *Proc. VLDB Endow.* 15, 12 (2022), 3666–3669. https://www.vldb.org/pvldb/vol15/p3666-fan.pdf
[34] Cong Yan and Yeye He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1539–1554. https://doi.org/10.1145/3318464.3389738
[35] Chengrun Yang, Jicong Fan, Ziyang Wu, and Madeleine Udell. 2020. AutoML Pipeline Selection: Efficiently Navigating the Combinatorial Space. In *KDD*, Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash (Eds.). ACM, 1446–1456. https://doi.org/10.1145/3394486.3403197
[36] Junwen Yang, Yeye He, and Surajit Chaudhuri. 2021. Auto-Pipeline: Synthesize Data Pipelines By-Target Using Reinforcement Learning and Search. *Proc. VLDB Endow.* 14, 11 (2021), 2563–2575. https://doi.org/10.14778/3476249.3476303
[37] Ziyue Zhong, Meihui Zhang, Ju Fan, and Chenxiao Dou. 2022. Semantics Driven Embedding Learning for Effective Entity Alignment. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2127–2140. https://doi.org/10.1109/ICDE53745.2022.00205
[38] Barret Zoph and Quoc V. Le. 2016. Neural Architecture Search with Reinforcement Learning. *CoRR* abs/1611.01578 (2016). arXiv:1611.01578 http://arxiv.org/abs/1611.01578

Received July 2022; revised October 2022; accepted November 2022