CtxPipe: Context-aware Data Preparation Pipeline Construction for Machine Learning

HAOTIAN GAO, National University of Singapore, Singapore and NUSRI Chongqing, China
SHAOFENG CAI, National University of Singapore, Singapore
TIEN TUAN ANH DINH, Deakin University, Australia
ZHIYONG HUANG, National University of Singapore, Singapore and NUSRI Chongqing, China
BENG CHIN OOI, National University of Singapore, Singapore

Machine learning models are only as good as their training data. Simple models trained on well-chosen features extracted from the raw data often outperform complex models trained directly on the raw data. Data preparation pipelines, which clean and derive features from the data, are therefore important for machine learning applications. However, constructing such pipelines is a resource-intensive process that involves deep human expertise.

Our goal is to design an efficient framework for automatically finding high-quality data preparation pipelines. The main challenge is how to explore a large search space of pipeline components with the objective of computing features that maximize the performance of the downstream models. Existing solutions are limited in terms of feature quality, which results in low accuracies of the downstream models, while incurring significant runtime overhead. We present CtxPipe, a novel framework that addresses the limitations of previous works by leveraging contextual information to improve the pipeline construction process. Specifically, it uses pre-trained embedding models to capture the data semantics, which are then used to guide the selection of pipeline components. We implement CtxPipe with deep reinforcement learning and evaluate it against state-of-the-art automated pipeline construction solutions. Our comprehensive experiments demonstrate that CtxPipe outperforms all of the baselines in both model performance and runtime cost.

CCS Concepts: • **Information systems** → **Data analytics**; **Data cleaning**; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: Data Analytics Pipeline, Data Preparation

ACM Reference Format:
Haotian Gao, Shaofeng Cai, Tien Tuan Anh Dinh, Zhiyong Huang, and Beng Chin Ooi. 2024. CtxPipe: Context-aware Data Preparation Pipeline Construction for Machine Learning. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 231 (December 2024), 27 pages. https://doi.org/10.1145/3698831

1 Introduction

The quality of input data determines the quality of Machine Learning (ML) models. For machine learning applications involving relational data, simple tree-based models trained with well-selected features derived from the raw data often have comparable, if not better model performance than that of state-of-the-art deep learning models trained directly on the raw data [19, 20, 40, 48]. The

Authors' Contact Information: Haotian Gao, National University of Singapore, Singapore, Singapore and NUSRI Chongqing, Chongqing, China, gaohaotian@comp.nus.edu.sg; Shaofeng Cai, National University of Singapore, Singapore, Singapore, shaofeng@comp.nus.edu.sg; Tien Tuan Anh Dinh, Deakin University, Melbourne, Victoria, Australia, anh.dinh@deakin.edu.au; Zhiyong Huang, National University of Singapore, Singapore, Singapore and NUSRI Chongqing, Chongqing, China, huangzy@comp.nus.edu.sg; Beng Chin Ooi, National University of Singapore, Singapore, Singapore, ooibc@comp.nus.edu.sg.

ACM 2836-6573/2024/12-ART231
https://doi.org/10.1145/3698831

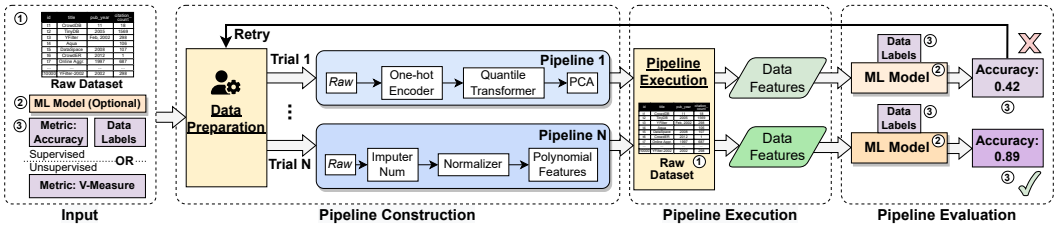

Fig. 1. Constructing a data preparation pipeline to maximize the performance of the generated data features.

tree-based models are not only more efficient to train but also have better interpretability [39, 46]. Extracting high-quality features from raw data requires a *data preparation pipeline*, which comprises multiple components that clean and transform raw data into features [1, 33]. These components either come from the ML frameworks, or are defined by the user.

The process of building a data preparation pipeline that generates high-quality features, or *data preparation pipeline construction*, is illustrated in Figure 1. It is performed by a data scientist, taking as input the raw dataset, an ML model, and its corresponding metric for evaluation. For supervised learning, the data labels are also provided. The process is manual and iterative. At each iteration, the scientist constructs a pipeline, executes it to derive a set of features, trains the model, and observes the model performance. The final output is the pipeline that achieves the best model performance.

Our goal is to design an efficient framework for automatically finding high-quality data preparation pipelines, that is, for automated data preparation pipeline construction. The main challenge is how to efficiently explore a large search space of pipeline components. In particular, the search space of possible component combinations grows exponentially with the number of possible components for different cleaning and transformation tasks [49]. Furthermore, the time cost of evaluating the pipelines is significant, because for each pipeline this cost includes the execution of the pipeline, as well as the model training and testing. With distribution shifts [32, 39, 47] requiring periodic pipeline rebuilding, this overhead accumulates and becomes significant over time.

Existing works on automated pipeline construction fall short of achieving our goal [8, 16, 21, 24, 26, 28–30, 33, 42, 46, 49, 54, 60]. Different techniques, e.g., genetic programming [42, 49], Bayesian optimization [16, 46, 54], reinforcement learning [8, 11, 21, 46], and differential formulation [24, 33, 60] are used to select and permute pipeline components. However, the search space still grows exponentially, resulting in significant runtime overhead. One reason for the large search space is that these works only use table schema and other statistical information to guide the exploration.

Our insight is that we can leverage *contextual information* to significantly reduce the search space. Intuitively, by capturing the *domain expertise* that data scientists use during the manual construction of data preparation pipelines, we can perform a more guided, efficient search. Such domain knowledge from the data, or the contextual information, can be derived from multiple sources (Figure 2). For example, the domain of the datasets (such as healthcare, finance, e-commerce, etc.) can be inferred from the column names and concrete string values in the table. The relationship across columns can be derived from columns with similar names. The high-level semantics of statistical information can also be derived from the type and range of numerical values in the columns. These column semantics, if captured automatically, could enable a more fine-grained separation of dataset scenarios to the pipeline construction solution, thereby improving its predictive performance.

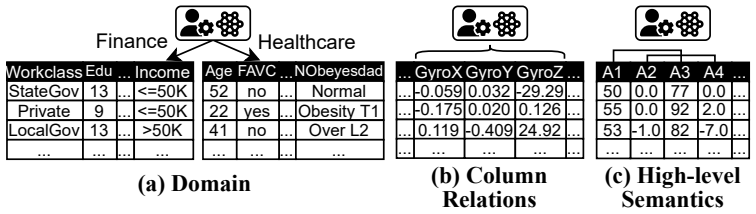

Fig. 2. Examples of contextual information.

We present CtxPipe[1], an efficient framework for automated data preparation pipeline construction. It addresses the limitations of previous works by leveraging contextual information from the data to improve the pipeline search. More specifically, CtxPipe employs state-of-the-art pre-trained embedding models to capture the semantics of raw and transformed data, then uses them to dynamically manipulate the training and inference of a pipeline construction model via a *gated context plug-in* module. CtxPipe can identify the pipeline that produces high-quality features, because pipeline search is guided within a specific context. In contrast, previous works use only table statistics to guide the search, which means that two tables with similar statistics but different semantics may produce similar pipelines and the corresponding features. CtxPipe implements the context plug-in based on deep reinforcement learning [11, 21]. It further improves the module robustness by introducing an optimized agent optimization algorithm called *Open-Closed-Gated (OCG) experience replay* that performs controlled backpropagations under marginal plug-in setups.

The most similar system of CtxPipe is HAIPipe [11], which leverages pre-built pipelines to complement the automatically generated pipelines. However, such pre-built pipelines are created manually, making the entire process less automated. In contrast, our system integrates external knowledge while enabling a fully automated pipeline construction workflow. To the best of our knowledge, we are the first to design and implement a fully automated, context-aware solution for data preparation pipeline construction.

In summary, we make the following contributions:

- We present CtxPipe, a novel and efficient framework for automated data preparation pipeline construction. It makes use of embedding models to automatically extract context into vectors, which are then integrated into the pipeline construction algorithm via a *gated context plug-in* module.
- We propose an agent optimization strategy called *Open-Closed-Gated (OCG) experience replay* that performs effective and stable joint optimization on the main deep reinforcement learning network and the context plug-in.
- We compare CtxPipe against state-of-the-art solutions in terms of the quality of transformed data on 18 commonly used datasets. Our extensive experimental results show that CtxPipe outperforms the baselines in all evaluation metrics, and achieves up to 170.99x speedup in pipeline construction compared to the second-best baseline.

The remainder of the paper is structured as follows. Section 2 reviews the related work. Section 3 presents the preliminaries and the problem definition. Section 4 overviews CtxPipe, focusing on its pipeline construction agent. Section 5 discusses the context plug-in in detail, including its context integration mechanism and agent optimization. Section 6 discusses the implementation, followed by the evaluation results in Section 7. Section 8 concludes the paper.

[1]The source code of CtxPipe is available at: https://github.com/ctxpipe.

Table 1. Systems for automated data preparation pipeline construction (A: Automated, FS: Flexible Structure, EI: Efficient Inference, C: Context-aware).

System	Optimization Method	A	FS	EI	C
WindTunnel [60]	Operator to Neural Network	✓			
DiffML [24]	Mixtures	✓			
DiffPrep [33]	Parameterization + Differential Relaxation	✓	✓		
Auto-WEKA [54]	Bayesian Optimization	✓			
Auto-Sklearn [16]	Bayesian Optimization	✓			
TPOT [42]	Genetic Programming	✓	✓		
Alpine Meadow [46]	Multi-Armed Bandit + Bayesian Optimization	✓	✓		
Learn2Clean [8]	Deep Reinforcement Learning	✓	✓	✓	
DeepLine [21]	Deep Reinforcement Learning + Hierarchical Step for Action	✓	✓	✓	
HAIPipe [11]	Deep Reinforcement Learning + Human-AI Pipeline Combination		✓	✓	✓
SAGA [49]	Genetic Programming + Pruning by Monotonicity	✓	✓	✓	
CtxPipe	Deep Reinforcement Learning + Data Context Integration	✓	✓	✓	✓

2 Related Work

2.1 Data Preparation Pipeline Construction

Automating data preparation pipeline construction is a type of Automated Machine Learning (AutoML) application. It is more challenging than other AutoML applications, such as hyperparameter tuning and neural architecture search, because the number of possible choices for pipeline *components* and their combinations results in vast search spaces [12, 32]. We summarize the related works in Table 1. We categorize them into two main approaches: *discrete construction* and *differential pipeline*.

Discrete Construction. This approach assembles components into pipelines through a discrete and iterative process. This process is optimized using different techniques, including genetic programming [42, 49], Bayesian optimization [16, 54], pruned tree search [30, 37], boosting [28], prediction uncertainty reduction [26], Reinforcement Learning (RL) [8, 11, 21], and some combination thereof [46]. These techniques leverage experience from automatic trials or pre-built human-written pipelines [11] to guide the search for high-quality pipelines. Our work automatically extracts context from data, thereby eliminating the need for pre-build pipelines, which makes the pipeline construction process more automated.

Differential Pipeline. This approach models the data preparation pipeline construction as a jointly optimized problem of the combination of components and their parameters. The key idea is to make the test performance *differentiable* with respect to the pipeline structure so that the pipeline can be optimized via gradient descent. WindTunnel [60] transforms non-differentiable components into neural networks to enable end-to-end joint optimizations. DiffML [24] connects components using weighted sums, resulting in mixtures of pipelines. DiffPrep [33] proposes changes to the problem definition to allow parameterized and optimized component selections with differential relaxations. Although this can find better pipelines than discrete construction, it is slow to converge, as each step requires executing model training and evaluation on the entire dataset. Our work achieves the same pipeline quality as this approach, with a similar efficiency of discrete construction.

2.2 Profiling and Learning

Data Profiling. Pipeline components can be learned using features extracted from the dataset. *Data profiling* [2, 3] is an active research area focusing on extracting structural properties and relations within a dataset [38, 57]. It targets both specific [22, 27, 43, 44, 51] and generalized

Table 2. Notations.

Symbol	Description
$\mathcal{T} = \{t_i\}$	The set of candidate data preparation component types
$\mathcal{F} = \{f_j\}$	The set of candidate data preparation components
$\mathbf{t} \subseteq \mathcal{F}$	The set of data preparation components that have type t
E	The data flow between components/types
$O = \{T, E\}$	A logical data preparation pipeline (in DAG)
$G = \{F, E\}$	A physical data preparation pipeline (in DAG)
$\mathbf{s}$	The state of the pipeline construction agent
$\boldsymbol{\psi}, \boldsymbol{\rho}, \mathbf{e}$	The statistical, pipeline and contextual features
θ_g	The parameters of a learnable computation g
Q	The main DQN network (without context plug-in)
$Q^{(k)}$	The k-th layer of main DQN network's MLP
ϕ	The evaluation metric

properties [9, 15, 36] of the dataset. Existing data profiling techniques can be leveraged for better decisions on transforming the dataset, which is similar to finding and applying data preparation pipelines. In particular, assuming the availability of the data profiles, data transformations can be suggested for fixing the target dataset given contrasting scenarios [17].

Our problem is orthogonal to data profiling, as it focuses on generating components without prior knowledge of quality. We use the agent to perform automatic profiling learned from experience via RL techniques, and leverage contextual information to tune the agent's action. We note that profiling hypotheses could serve as valuable prior knowledge for a fine-grained selection of specific columns and their combinations.

Deep Learning for Tabular Data. The ML models in Figure 1 can be replaced by deep learning models that perform the desired task directly using the raw data. Such models can be Transformer-based neural networks pre-trained with masked language tokens [23, 52, 59] or table entities [4, 13, 25], which are then fine-tuned for downstream tasks, such as entity resolution, question answering, text-to-SQL, entity population, etc. Our work focuses on data preparation pipelines, which preprocess data to derive high-quality features for training simple, yet highly accurate and interpretable models.

3 Preliminary

In this section, we introduce the preliminaries of automated data preparation pipeline generation and outline the problem formulation. Table 2 summarizes frequently used notations in this paper.

3.1 Constructing Data Preparation Pipeline

Data Preparation Components. We adopt the definition of data preparation components, i.e., primitives, from various studies on data preparation and analytics pipelines [11, 33, 39]. Specifically, a component $f \in \mathcal{F}$ encapsulates a data transformation process and exposes the same interface that can be applied to a dataset X, i.e., $X' = f_\theta(X)$, where θ denotes the parameters of f. Each component corresponds to a type $t \in \mathcal{T}$, where $\mathcal{T}$ is the set of component types. For each type t, the corresponding component set $\mathbf{t} = \{f_1, f_2, ...\}$ represents those sharing a transformation purpose, for instance, the feature scalers `Min Max Scaler` and `Max Absolute Scaler` are defined as components typed feature preprocessing.

Without loss of generality, we use fixed sets of candidate $\mathcal{F}$ and $\mathcal{T}$ in this paper to illustrate the framework design of CtxPipe.

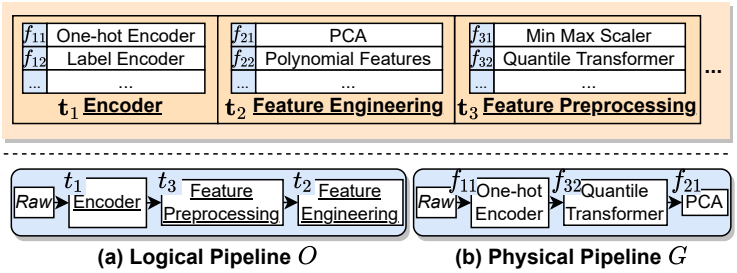

Fig. 3. Logical and physical pipelines.

Data Preparation Pipelines. A data preparation pipeline connects a series of components via data flows to support iterative transformations of a specific dataset X. The output of one component can become the input of another. Components are executed sequentially, transforming the data from the original form to the final output. A pipeline can be linear, where components have at most one input and one output, or non-linear, where components with multiple inputs or outputs form a directed acyclic graph (DAG).

Logical and Physical Pipelines. To reduce the number of manual trials in pipeline construction, data scientists typically adopt a two-stage optimization strategy [33, 45, 46]. In the first stage, a *logical pipeline* O is created by fixing the pipeline structure with the component types and their order. We make the assumption, similar to existing works [18, 33], that components in O have unique types. Formally, a logical pipeline is defined as $O = \{T, E\}$, where $T = \{t_1, t_2, ...\}$ denotes the involved component types in the pipeline, and $E = \{(t_i, t_j) \mid t_i, t_j \in T\}$ denotes the data flow between components. The component type uniqueness requires that $\forall t_i, t_j \in T, t_i \neq t_j$. In the second stage, each component type t is instantiated to a specific component $f \in \mathbf{t}$, forming the *physical pipeline* G for O, i.e., $G = \{F, E\}$, where $F = \{f_i \mid f_i \in \mathbf{t}_i\}$ and $E = \{(f_m, f_n) \mid f_m \in \mathbf{t}_i, f_n \in \mathbf{t}_j\}$. Figure 3 shows examples of logical and physical pipelines, where the components f_{11}, f_{32} and f_{21} are selected from the component sets $\mathbf{t}_1$ to $\mathbf{t}_3$.

Unique component types. The assumption of unique component types in a logical pipeline is made to simplify the formulation. This is reasonable since existing work suggests that multiple components of the same type are rare in practice, and that they contribute little to the overall pipeline quality [33, 45, 46]. Furthermore, we can relax this assumption by adding i new types with duplicated available components to support $i + 1$ repeated component types [33].

3.2 Problem Formulation

Given a raw dataset $X \in \mathcal{X}$, a specific ML model $M \in \mathcal{M}$, and an evaluation metric $\phi \in \mathbf{\Phi}$, the objective of the automated data preparation pipeline construction problem is to optimize an algorithm $A : \mathcal{X} \times \mathcal{M} \times \mathbf{\Phi} \rightarrow \mathcal{G} \times \mathbf{\Theta}^{\mathcal{G}}$ that outputs (1) a physical pipeline $G = \{F, E\}$, and (2) a set of component parameters $\{\theta_f | f \in F\}$. The output satisfies that executing G on X, i.e., $G(X)$, yields data features that maximize the evaluation performance under metric ϕ. More specifically, the input dataset can be split into a tuple of training and test data, i.e., $X = (X_{train}, X_{test})$, where in unsupervised learning $X_{train} = \emptyset$. Given $M_\theta(X)$ representing inference performed on X with the model M parameterized by weights θ, the algorithm A finds the pipeline structure G^* and corresponding weights θ^* such that

$$G^*, \theta^* = \arg\max_{G,\theta} \phi(M_\theta(G(X_{test}))).$$

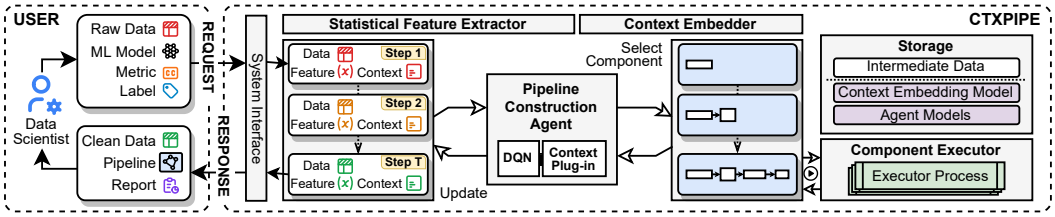

Fig. 4. Architecture of CtxPipe framework.

Typically, in supervised learning tasks, the requirement of model weights in this optimization objective is relaxed to a *greedy weight* setup, formulated as follows:

$$G^* = \arg\max_{G} \phi(M_{\theta^*}(G(X_{test}))),$$
$$\text{s.t.} \quad \theta^* = \arg\max_{\theta} \phi(M_{\theta}(G(X_{train}))).$$

The optimization derives the best pipeline G^* by considering the logical pipeline O, the components F that fit O, and the data flow E between F.

To simplify the discussion of the design choices, in this paper, we focus on (1) supervised learning tasks for the ML model, and (2) linear pipelines for the construction. Linear pipelines are defined such that, given a physical pipeline $G = \{F, E\}$, each component has at most one input and one output data flow:

$$\forall f \in F, \ \left[\left(\sum\nolimits_{(a,b)\in E,\, a=f} 1\right) \leq 1\right] \wedge \left[\left(\sum\nolimits_{(a,b)\in E,\, b=f} 1\right) \leq 1\right].$$

We note that all pipeline-related features in our design are represented in a high-level abstraction and thus can be seamlessly generalized to the DAG data structure.

Pipeline Evaluation and Optimization. The process of evaluating and optimizing a learned pipeline construction algorithm is similar to how data scientists use their knowledge and feedback from evaluation results to iteratively test data preparation pipelines, as discussed in Section 1. The steps are as follows: First, *initial pipeline generation*: given the raw dataset X, the algorithm generates a data preparation pipeline G based on its existing knowledge. Second, *pipeline execution*: G is executed on X to obtain the cleaned and transformed data features $G(X)$. Third, *model training and testing*: G is evaluated by splitting the X into the training set X_{train} and test set X_{test}, and then the model M is trained and tested to evaluate the performance of the pipeline. Finally, *feedback and refinement*: based on the evaluation results, the pipeline construction algorithm is refined and updated. This process of execution, evaluation, and refinement is typically iterated until the desired level of performance is achieved.

4 CtxPipe

In this section, we describe the high-level framework of CtxPipe, focusing on the selection process of data preparation pipeline construction. We also introduce how contextual information is seamlessly integrated to guide the selection and ordering of components.

4.1 Framework

Figure 4 illustrates the overview of the CtxPipe framework, which consists of four major modules: the Statistical Feature Extractor and Context Embedder, the Pipeline Construction Agent, the Component Executor, and the Storage Layer. Specifically, the *statistical feature extractor* and *context*

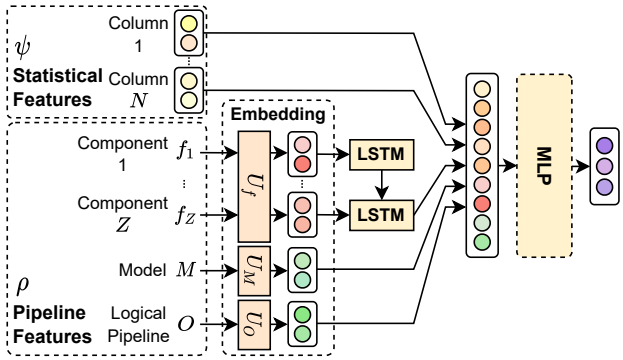

Fig. 5. Architecture of the main DQN network with the statistical and pipeline features as input.

embedder extract statistical and contextual information respectively from the input. The *pipeline construction agent* iteratively selects new components for the pipeline. The *component executor* receives and executes the selected component on the current transformed dataset. The *storage layer* handles the storage requirements for other modules and intermediate results.

Workflow. To initiate and execute the data preparation pipeline, users first prepare the raw dataset for either a supervised or unsupervised task, along with other optional inputs, and then submit a pipeline generation request through the programming interface. Upon receiving the request, the system iteratively processes the parsed input through a core loop of the following steps:

(1) *Statistical feature extraction and context embedding:* the dataset X_i (where i denotes the pipeline construction step) is processed by the *statistical feature extractor* to compute necessary statistical features, and the *context embedder* to generate embeddings for contextual information.
(2) *Pipeline construction:* the generated statistical features $\boldsymbol{\psi}_i$ and context embeddings $\mathbf{e}_i$ serve as input to the *pipeline construction agent*, which determines the logical pipeline O or the next component f_i in the physical pipeline G.
(3) *Component execution:* The component f_i newly added to the physical pipeline is then executed on the current dataset by the *component executor*, which produces a new dataset $X_{i+1} = f_i(X_i)$ that is recorded in the history.
(4) *Iterative update:* steps 1-3 are repeated for X_{i+1} to iteratively update the dataset until all components are finalized for the physical pipeline.
(5) *Output and reporting*: the final cleaned dataset X^* and the pipeline G used to produce X^* are returned to the user. Additionally, a pipeline construction report of the structure and the data transformation process is provided for further analysis.

CtxPipe employs deep reinforcement learning, specifically the Deep Q-Network (DQN), to build the pipeline construction agent. This allows the algorithm to easily extract and integrate contextual information by fusing neural network architectures. Moreover, it is enhanced by the *context integration* mechanism via a novel *context plug-in*, which will be elaborated on in the following sections. Notably, the logical and physical pipeline basically follow the same selection and integration process. Hence, for simplicity, in the following sections, we mainly introduce the context integration mechanism for selecting physical components.

4.2 Pipeline Construction Agent

The Reinforcement Learning (RL) agent takes as input the current state $\mathbf{s}$ and relies on its learned state-action function $Q(\mathbf{s}, f; \theta)$ to select the component f that maximizes the estimated reward. The RL policy π_θ is used to identify the *action* of selecting the component f^* that maximizes the Q-value, i.e., $f^* = \arg\max_f Q(\mathbf{s}, f; \theta)$. The Q-values are computed using the Bellman equation, which evaluates the expected sum of discounted multi-step rewards: $Q(\mathbf{s}, f; \theta) = \mathbb{E}_{\pi_\theta}[r + \gamma \max_{f'} Q_i(\mathbf{s}', f') | \mathbf{s}, f]$, where r is the reward of selecting f for the current step, and $\gamma \in (0, 1]$ is the discount factor.

For deep RL, the agent employs a neural network to model $Q(\mathbf{s}, f; \theta)$ We refer to this as the main DQN network. The network takes *statistical features* $\boldsymbol{\psi}$ and *pipeline features* $\boldsymbol{\rho}$ of the current dataset X_i as input, and processes them via a series of embedding layers. The output vectors are then concatenated and passed to a *multi-layer perceptron* (MLP) to produce the Q-values. Figure 5 illustrates the architecture of the main DQN network. We adopt the same architecture as that in existing works using deep RL to construct pipelines [11], in order to isolate the effect of the context integration mechanism.

Statistical Features. For each intermediate dataset X_i, its statistical features $\boldsymbol{\psi}_i$ consists of column-wise features, i.e., $\boldsymbol{\psi}_i = [\boldsymbol{\psi}_i^{(1)} \boldsymbol{\psi}_i^{(2)} ... \boldsymbol{\psi}_i^{(N)}]$, where $\boldsymbol{\psi}_i^{(k)}$ corresponds to features of the k-th column, and N is the maximal number of column features that $\boldsymbol{\psi}_i$ are truncated or padded to, which ensures a uniform statistical feature size across tasks. The column features $\boldsymbol{\psi}_i^{(k)}$ comprises a list of feature values representing critical information that can be exploited for determining components, e.g., column types, the ratio of missing values, the number of unique values, etc. Particularly, for numeral columns, additional statistics such as mean, standard deviation, and median can also be calculated as statistical features.

Notably, many numerical features of $\boldsymbol{\psi}_i$ span a real number range $\mathbb{R}$, and direct normalizing these features without considering the distribution could lead to precision issues, such as outlier values diminishing the distinction between other values. To address this issue, we instead represent each numerical feature ψ as $(\text{frac}(\psi), \exp(\psi))$, a combination of its fraction $\text{frac}(\psi)$ and exponent $\exp(\psi)$, where $\psi = \text{frac}(\psi) \times 2^{\exp(\psi)}$. This representation preserves the scaling of numerical features, enables the learning model to adapt its focus based on the magnitude of the values, and effectively stabilizes the training process.

Pipeline Features. The pipeline features $\boldsymbol{\rho}_i$ for the dataset X_i contains information related to the current state of the constructed pipeline. Specifically, this includes the selected logical pipeline structure O, a specific ML model M to evaluate the pipeline for the reward, and the component selection history up to f_{i-1}. These categorical features are processed via dedicated embedding layers U_O, U_M, and U_f, respectively. Particularly, the embedded component selection history $[U_f(f_1), U_f(f_2), ..., U_f(f_Z)]$ are processed by an LSTM model, whose outputs are then concatenated with embeddings $U_M(M)$ and $U_O(O)$ of M and O respectively to constitute the pipeline feature $\boldsymbol{\rho}_i$. The statistical features $\boldsymbol{\psi}_i$ and the pipeline feature $\boldsymbol{\rho}_i$ are then fed into the MLP to produce Q-values.

Learning the Global Policy. In the training phase, a *global policy* is learned iteratively as follows: (1) the RL agent randomly samples a dataset from a large pool of training datasets and exploits its current statistical and pipeline features to generate a pipeline for this dataset. (2) The pipeline is evaluated by executing it on the dataset. (3) The pipeline performance is used as the reward to optimize the agent to generate better-performing pipelines on datasets. As the datasets are from different domains, with diverse characteristics and distributions, the learned global policy can generalize across datasets with different semantics and characteristics.

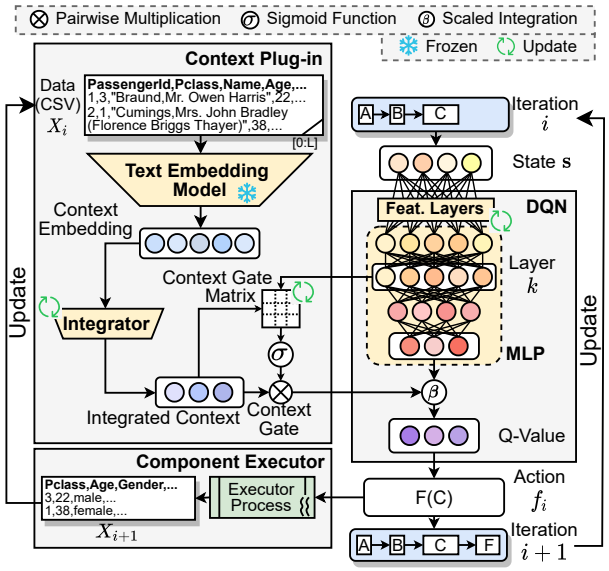

Fig. 6. Design of context plug-in and its interaction with the main DQN network. Feature layers contain embedding models and the LSTM for component embeddings in Figure 5.

5 Context Plug-in

In conventional automated pipeline construction processes [11, 21, 46], the selection of pipeline structures and components is typically based solely on the statistical features of the dataset. However, this approach can be significantly enhanced by integrating semantics and contextual information of the dataset, which are critical to the decision-making process of the agent by providing information beyond the statistical features.

To support such integration, we develop a *context plug-in* that operates alongside the main DQN network. This plug-in interacts with the main network at specific layers of DQN activations. By functioning independently from DQN, the context plug-in enhances the adaptability and applicability of our framework.

5.1 Overview

The architecture of the context plug-in to be integrated into the agent is illustrated in Figure 6. This integration takes a five-step iterative process: (1) the data table of the dataset is converted into the textual format and processed by a *text embedding model* to generate a *context embedding* $\mathbf{e}$. (2) The context embedding is then transformed by an *integrator model* C to produce the integrated context $C(\mathbf{e})$ that aligns its dimensionality with the output of DQN. (3) The transformed integrated context $C(\mathbf{e})$ is further manipulated by a learned *context gating module* to filter out less contributive contextual information. (4) Next, this gated context is fused with the DQN output using a *scaled integration technique* to produce the final Q-values $\mathbf{q}$. (5) Finally, the action, which corresponds to selecting a specific logical pipeline O or a physical component f, is determined based on $\mathbf{q}$. Particularly, for the selected physical component, f is then executed to transform the current dataset.

Algorithm 1: Action(): Select component with ϵ-greedy and context plug-in.

Input: Q: Action-value function; $\mathbf{s}$: Agent state; $\mathbf{e}$: Context embedding; $\mathbf{t} = \{f_1, f_2, ...\}$: Candidate components of type t.
Output: f^*: Selected component.

1 $\tau \sim U(0, 1)$
2 **if** $\tau < \epsilon$ **then**
3 | **return** randomly selected component
4 **else**
5 | Compute integrated context $C(\mathbf{e})$ through integrator C
6 | $\boldsymbol{\alpha} \leftarrow \sigma\left(\mathbf{W}_c \left[C(\mathbf{e}); Q^{(k)}(...(Q^{(2)}(Q^{(1)}(\mathbf{s}, \mathbf{t})))\right] + \mathbf{W}_b\right)$
7 | $\boldsymbol{\beta} \leftarrow \boldsymbol{\alpha} \circ (C(\mathbf{e}) - \boldsymbol{J}) + \boldsymbol{J}$ `// J is all-ones matrix`
8 | $\mathbf{q} \leftarrow \boldsymbol{\beta} \circ Q(\mathbf{s}, \mathbf{t})$
9 | **return** $f^* = \arg\max_f q_f$

The context plug-in involves four key modules: the context embedding, the context integrator, the learned context gate, and the scaled integration technique, which are introduced to enhance the selection of the next component f. Algorithm 1 describes the `Action` function that computes Q-values with the context plug-in to select a component f from all possible candidate components of type t, which adopts the ϵ-greedy policy for the component selection (Line 1-3) and replaces the process of computing Q-values $Q(\mathbf{s}, \mathbf{t})$ with the context-integrated version (Line 4-9).

5.2 Context Plug-in Modules

Context Embedding. For effective context extraction and integration, we employ vector embedding techniques. We consider two main techniques for transforming dataset contextual information into vector embeddings. One is to use specialized *table representation models* to encode the entire table into a vector, which is effective in capturing dedicated features of tabular data, such as relationships between cells. However, these models often underperform the latest text-based models in comprehending and embedding context due to limited training on specialized datasets. The other technique converts the data table into a formatted sequence of text, and then uses pre-trained *text embedding models* to transform the data text into embedding vectors. This technique benefits from extensive pre-training on diverse text corpora, and thus can provide rich semantic and context embeddings. However, these embeddings may not be sensitive to the positional and structural relationships among cells as they are not trained over table-specific tasks [14, 59].

Given these considerations, we find text-based embeddings more appropriate for our pipeline construction context. The reason is that during component exploration, the semantics of schemas and entities are more informative than their positions and cell relations. In particular, data tables often undergo transformations, where columns are frequently added, removed, or mutated, making cell values highly dynamic. However, for context comprehension, these actions do not fundamentally change the semantics of the dataset. Text-based embedding models handle these dynamics effectively, focusing on the underlying semantics without being confounded by the physical changes to the table. In contrast, table representation models typically generate embeddings obtained from predicting cell values rather than understanding these dataset semantics.

Due to the token limit of the text embedding model, we need to preprocess the dataset X_i before obtaining its context embeddings. First, we randomly sample rows from X_i's data table to allow the embedding model to capture both the schema and the distribution of columns. Its number is determined based on the token limit, the number of columns, and the value patterns of each

column. Next, these sampled rows are compiled into a new table with the same schema as X_i, which is then converted to the CSV format with the table header preserved. This format allows the embedding model to comprehend the header and row values in a uniform and structured textual input. The textual input is then truncated at L tokens defined by the model specification, and fed into the embedding model, which can thus prioritize header information to ensure effective context extraction.

Context Integrator. The output of the text embedding model, $\mathbf{e}$, is then processed by a *context integrator* model C to align the dimension of $\mathbf{e}$ to that of the DQN network output, i.e., $|C(\mathbf{e})| = |\mathbf{t}|$, where $\mathbf{t} = \{f_1, f_2, ...\}$ is the set of candidate components of type t. The resulting $C(\mathbf{e})$, namely the *integrated context*, is used to manipulate the result of the main DQN network, after being filtered by the *context gating module* introduced below.

Learned Context Gate. The context embeddings produced by the text embedding model may sometimes misinterpret semantics or extract noisy contextual information. To address this issue, we further introduce a component-wise gating module to obtain the *gating factor* $\boldsymbol{\alpha}$, specifically, $\mathbf{s} \times \mathbf{e} \times \mathbf{t} \mapsto \boldsymbol{\alpha}$, where $\mathbf{s}$ is the state vector of the agent, $\mathbf{e}$ is the context embedding vector, $\mathbf{t}$ is the set of candidate components with type t, and $\boldsymbol{\alpha} \in [0, 1]^{|\mathbf{t}|}$. The gating factor $\boldsymbol{\alpha}$ will be used to dynamically rescale the integrated context via the scaled integration technique, and this gating module is learned end-to-end together with other modules.

Due to the training instability of reinforcement learning [41, 55], we introduce a lightweight structure for computing gating factor $\boldsymbol{\alpha}$ to prevent overfitting. Specifically, the gating module extracts the following two key features as input: (1) the integrated context $C(\mathbf{e})$ based on the context embedding $\mathbf{e}$, and (2) activations of the k-th layer of the main DQN's MLP. Choosing a smaller k allows the layer output to preserve key statistical information in a compact embedding. Specifically, given $Q^{(k)}$ representing the k-th transformation layer of the main DQN's MLP, $\boldsymbol{\alpha}$ is computed by fusing the concatenated input features with a learned matrix, and normalizing the output values to fit within the target value range:

$$\boldsymbol{\alpha} = \sigma\left(\mathbf{W}_c\left[C(\mathbf{e}); Q^{(k)}(...(Q^{(2)}(Q^{(1)}(\mathbf{s}, \mathbf{t}))))\right] + \mathbf{W}_b\right), \tag{1}$$

where (1) $\mathbf{W}_c \in \mathbb{R}^{|\mathbf{t}| \times (|\mathbf{t}| + |Q^{(k)}|)}$ is a learnable weight matrix, with $|Q^{(k)}|$ denoting the dimension of $Q^{(k)}$ activations, (2) $\mathbf{W}_b \in \mathbb{R}^{|\mathbf{t}| \times 1}$ is a learnable bias matrix, and (3) σ is the sigmoid function to ensure $\boldsymbol{\alpha}$ values bounded between 0 and 1.

Scaled Integration. The final integration of contextual information into DQN is conducted via the scaled integration technique, where the integrated context $C(\mathbf{e})$ will be rescaled by the gating factor $\boldsymbol{\alpha}$ to inject the contextual knowledge into the main DQN output $Q(\mathbf{s}, \mathbf{t})$ via $\boldsymbol{\beta}$. The intuition is to make the context vector a *scaler* of the original state-action function, which (1) makes the integrated context more pertinent to the main DQN output, and (2) fine-tunes the output based on contextual information. Specifically, for candidate components $\mathbf{t} = \{f_1, f_2, ...\}$ with type t, the final Q-values $\mathbf{q} = \{q_1, q_2, ...\}$ is produced by:

$$\mathbf{q} = \boldsymbol{\beta} \circ Q(\mathbf{s}, \mathbf{t}),$$

where $\circ$ denotes Hadamard product, and $\boldsymbol{\beta}$ is the recalibrated scaling factor computed from $\boldsymbol{\alpha}$ and $C(\mathbf{e})$ as below. In particular, to support the experience replay optimized for context plug-in (introduced in Section 5.3), $\boldsymbol{\beta}$ is introduced to ensure the following properties for each $\alpha_i \in \boldsymbol{\alpha} = [\alpha_1\ \alpha_2\ ...\ \alpha_{|\mathbf{t}|}]^\top$:

$$\alpha_i = 0 \Rightarrow q_i = Q(\mathbf{s}, f_i),$$
$$\alpha_i = 1 \Rightarrow q_i = C(e_i)Q(\mathbf{s}, f_i).$$

Algorithm 2: Optimization for pipeline construction.

Input: $\mathcal{E}$: Environment, ϕ: Evaluation metric.
Output: $\mathcal{Q}$: Action-value functions to select logical pipeline and components.

Initialize replay memory $\mathcal{D}$
Initialize Q_0 with random weights θ_0
for $\mathbf{t} \in \mathcal{T}$ **do**
    Initialize $Q_{\mathbf{t}}$ with random weights $\theta_{\mathbf{t}}$
$\mathcal{Q} \leftarrow \{Q_0, Q_{\mathbf{t}}\}$
for *episode* $\leftarrow 1$ **to** M **do**
    Initialize $\mathcal{E}$ with a new dataset X
    Get statistical feature $\boldsymbol{\psi}_0$ and context embedding $\mathbf{e}_0$ from X
    $O = \{T, E\} \leftarrow \texttt{Action}(Q_0, \boldsymbol{\psi}_0, \mathbf{e}_0)$
    for $i, \mathbf{t}_i \in \texttt{enumerate}(T)$ **do**
        Get statistical feature $\boldsymbol{\psi}_i$ and context embedding $\mathbf{e}_i$ from X_i
        Get pipeline feature $\boldsymbol{\rho}_i$ from O and G
        $\mathbf{s}_i \leftarrow [\boldsymbol{\psi}_i; \boldsymbol{\rho}_i]$
        $f_i \leftarrow \texttt{Action}(Q_t, \mathbf{s}_i, \mathbf{e}_i, \mathbf{t}_i)$
        $X_{i+1}, r_i \leftarrow \mathcal{E}(X_i, f_i; \phi)$ `// New data and reward`
        Add f_i to G
        Preprocess $\mathbf{s}_{i+1}$ from X_{i+1}, O and G
        Store transition $(\mathbf{s}_i, f_i, r_i, \mathbf{s}_{i+1})$ in $\mathcal{D}$
        for $Q \in \mathcal{Q}$ **do**
            $\texttt{LearnQ}(Q, \mathcal{D}, i = \texttt{len}(O))$
Procedure $\texttt{LearnQ}(Q, \mathcal{D}, b)$**:**
    Sample random minibatch of transitions $\{(\mathbf{s}_j, f_j, r_j, \mathbf{s}_{j+1})\}$ from $\mathcal{D}$
    $y_j = \begin{cases} r_j & b = \texttt{true} \\ r_j + \gamma \max_f Q(\mathbf{s}_{j+1}, f; \theta) & \text{otherwise} \end{cases}$
    Perform an Open-Closed-Gated gradient descent step on $(y_j - Q(\mathbf{s}_j, f_j; \theta))$

In essence, for component set $\mathbf{t}$, the gating factor $\boldsymbol{\alpha}$ modulates how the contextual information $C(\mathbf{e})$ scales the original reward estimations $Q(\mathbf{s}, \mathbf{t})$. To achieve this, $\boldsymbol{\beta}$ is defined as follows:

$$\boldsymbol{\beta} = \boldsymbol{\alpha} \circ (C(\mathbf{e}) - \boldsymbol{J}) + \boldsymbol{J},$$

where $\boldsymbol{J} = [1\,1\,...\,1]^\top$. This linear transformation ensures that $\boldsymbol{\beta}$ impartially modifies the gradient flow with an even-handed influence across all component reward adjustments during backpropagation.

5.3 Agent Optimization

The design of DQN with context plug-in supports end-to-end joint optimization of the main network and the context integration mechanism. The detailed optimization of the agent with context plug-in is described in Algorithm 2. The agent optimization generally follows the original DQN training and inference process, using an off-policy strategy performing ϵ-greedy actions across episodes [41]. Meanwhile, to fit the pipeline construction scenario, the process is enhanced with an optimized experience replay strategy to stabilize the learning of the context plug-in.

In each episode, the agent performs a two-phase strategy, constructing the logical pipeline first and the physical pipeline afterward. It first determines the logical pipeline O by feeding the

Algorithm 3: Open-Closed-Gated gradient descent step.

Input: $\mathbf{q}$: Target value, η: Learning rate.

/* Open ($\boldsymbol{O}$ is all-zeros matrix) */

$L_1 \leftarrow \|$(Inference with gate valued $\boldsymbol{O}$) - $\mathbf{q}\|$

Update θ_Q: $\theta_Q = \theta_Q - \eta \nabla_{\theta_Q} L_1$

/* Gated */

$L_2 \leftarrow \|$(Inference with gate valued $\boldsymbol{\beta}$) - $\mathbf{q}\|$

Update $\mathbf{W}_c$: $\mathbf{W}_c = \mathbf{W}_c - \eta L_2 (C(\mathbf{e}) - \boldsymbol{J})^\top \nabla_{\mathbf{W}_c} \boldsymbol{\alpha}$

Update $\mathbf{W}_b$: $\mathbf{W}_b = \mathbf{W}_b - \eta L_2 (C(\mathbf{e}) - \boldsymbol{J})^\top \nabla_{\mathbf{W}_b} \boldsymbol{\alpha}$

Update θ_C: $\theta_C = \theta_C - \eta L_2 \left((C(\mathbf{e}) - \boldsymbol{J})^\top \nabla_{\theta_C} \boldsymbol{\alpha} + \boldsymbol{\alpha}^\top \nabla_{\theta_C} C(e) \right)$

$|Q| \leftarrow$ The number of layers in Q

for $l \in [|Q|, |Q| - 1, ..., k + 1]$ **do**

  Update $\theta_{Q^{(l)}}$: $\theta_{Q^{(l)}} = \theta_{Q^{(l)}} - \eta \nabla_{\theta_{Q^{(l)}}} L_2$

for $l \in [k, k - 1, ..., 1]$ **do**

  Update $\theta_{Q^{(l)}}$: $\theta_{Q^{(l)}} = \theta_{Q^{(l)}} - \eta L_2 \left((C(\mathbf{e}) - \boldsymbol{J})^\top \nabla_{\theta_{Q^{(l)}}} \boldsymbol{\alpha} \right) + \nabla_{\theta_{Q^{(l)}}} L_2$

/* Closed ($\boldsymbol{J}$ is all-ones matrix) */

$L_3 \leftarrow \|$(Inference with gate valued $\boldsymbol{J}$) - $\mathbf{q}\|$

Update θ_C: $\theta_C = \theta_C - \eta \, \mathrm{diag}(L_3) \nabla_{\theta_C} C(e)$

Update θ_Q: $\theta_Q = \theta_Q - \eta \nabla_{\theta_Q} L_3$

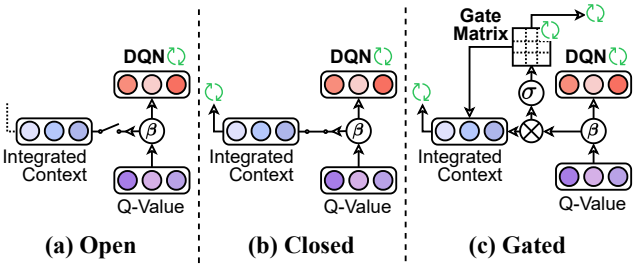

Fig. 7. Backpropagation paths of Open-Closed-Gated Experience Replay.

statistical features $\boldsymbol{\psi}_0$ and contextual embedding $\mathbf{e}_0$ of the raw dataset X into agent Q_0 specialized for generating logical pipelines (Line 7-9). This results in the subset of involved component types $T \subseteq \mathcal{T}$ and edges E for O. Then, for each component type t_i enumerated in the topological order, the component is selected from the candidate components $\mathbf{t}_i$ (Line 11-14). Both logical pipeline and component selection perform the `Action` procedure defined in Algorithm 1, while the feature differs by including the feature vector representing current pipeline structure $\boldsymbol{\rho}_i$ for selecting the next component f_i. After selection, the dataset gets transformed by f_i, and reward r_i is computed as defined in the evaluation metric ϕ, with the state change recorded in replay buffer $\mathcal{D}$ (Line 15-18). Finally, the `LearnQ` procedure is called to perform agent optimization via our proposed *Open-Closed-Gated experience replay*, an optimized network learning strategy for context integration mechanism.

Open-Closed-Gated Experience Replay. In order to preserve the Reinforcement Learning (RL) disciplines, the context plug-in should participate across the whole RL agent optimization process.

However, simply performing experience replay on the joint main DQN and context plug-in networks would destabilize the agent optimization. This is because the different controlling semantics of both networks are connected with a branching and merging architecture. Hence, a single backpropagation is prone to handle the joint optimization with different knowledge uniformly, making the plug-in less functional.

To avoid such degradation for the context plug-in, we propose an optimized joint network training strategy that enforces the separated networks to equip them with the corresponding expected functionality. The strategy, namely *Open-Closed-Gated (OCG) experience replay*, performs a multi-stage backpropagation and controls the gated network to ensure the functionality of different modules within the joint networks.

Figure 7 illustrates its basic backpropagation paths. The detailed backpropagation procedure is listed in Algorithm 3. From a high-level perspective, it contains three stages where the context gate is periodically fully open (Line 1-2), passed with gate vector $\boldsymbol{\beta}$ (Line 3-8), and fully closed (Line 9-11). In each stage, the loss is computed on the controlled joint networks and is used to guide the backpropagation of the corresponding flow with different updating strategies for model weights. The intuition is to (1) force the main DQN to learn with and without external contextual information, and (2) let the context integrator understand the usage of such information to manipulate the better-learned state-action function. Afterward, the context gate could (3) determine the passing and filtering of the contextual knowledge and be less interfered with by low-quality and unstable manipulations. Some gradient computations, involving the vector-matrix ones (e.g., $\nabla_{\mathbf{W}_c}\boldsymbol{\alpha}$ when backpropagating through Equation 1) and those with uncertain model structure (e.g., C and Q), are not expanded in the algorithm. We note that the agent optimization, without complex custom logic, could be fully handled by the `autograd` module of deep learning frameworks.

6 Implementation

In this section, we describe in detail the specifications of CtxPipe's pipeline construction agent used in the experiments.

Component and User-Defined Functions. We adopt scikit-learn APIs for the component interface. Specifically, a component's transformation is done by calling the `fit()` and `transform()` methods of the operator's class. This allows the user to define a new component by defining a class with these two methods and performing the corresponding logic by User-Defined Functions (UDFs). The new component can then be registered to CtxPipe, which will finally learn when to select it via deep reinforcement learning.

Main DQN. The statistical features consist of embeddings for at most 100 input columns, i.e., $N = 100$. The pipeline features comprise a 16-dimensional embedding for the model M, an 8-dimensional embedding for the logical pipeline structure O, and component embeddings fed into an LSTM with a 96-dimensional hidden state. These feature representations are concatenated and passed through the Multi-Layer Perceptron (MLP), which incorporates 5 linear layers and `LeakyReLU` activations between each. The output of the last layer is transformed by a `tanh` function. Moreover, the output of the first layer is sent to the context plug-in for context integration.

Context Plug-in. The models in the context plug-in involve context embedding and integrator. For context embedding, we use GTE-large [35] as the text embedding model, and randomly sample 50 rows in the data table of the dataset as its input. The model produces a 1,024-dimensional context embedding $\mathbf{e}$. For context integrator C, we use a small two-layer MLP with a 128-dimensional intermediate feature, `LeakyReLU` activation, and transform the output of the second layer with a `tanh` function to produce $C(\mathbf{e})$.

Hyperparameters. Most aforementioned hyperparameters for the main DQN are selected following the model structure of existing works [11]. Other hyperparameters, including those for the

Table 3. Selected components and their types.

Type	Component
Imputer	Mean, Median, Most Frequent
Encoder	Numeric Data, Label Encoder, One-hot Encoder
Feature Preprocessing	Min Max Scaler, Max Absolute Scaler, Robust Scaler, Standard Scaler, Quantile Transformer, Normalizer, Power Transformer, K-bins Discretizer
Feature Engineering	Polynomial Features, Interaction Features, PCA, Kernel PCA, Incremental PCA, Truncated SVD, Random Trees Embedding
Feature Selection	Variance Threshold

context plug-in, are set empirically based on our own experiments. We evaluate the effect of these hyperparameters in Section 7.7.

Agent Optimization. All variants of CtxPipe's agents, including ablations, are trained for 32,000 steps on the training dataset used in [11]. Notably, we exclude from the training datasets those sharing a similar schema to any test datasets to prevent data leakage.

7 Evaluation

In this section, we evaluate the performance of CtxPipe.

7.1 Baselines

We compare CtxPipe to the state-of-the-art solutions with both discrete construction and differential pipeline discussed in Section 2. In particular, the following setups are included as competitors:

- **Default (DEF)** from DiffPrep [33]. It implements the pipeline structure commonly used in AutoML frameworks like Azure AutoML [7] and H2O [31]. The pipeline imputes missing values with the mean value for numerical columns and the most frequent value for categorical columns, then normalizes each feature using standardization.
- **DeepLine (DL)**, with the same configuration listed in [21]. We use the provided source code and train the model for 150,000 steps, as recommended in its code documentation.
- **AI-pipe of HAIPipe (HAI-AI)**, which applies the deep reinforcement learning-based AI-pipeline generation algorithm proposed by HAIPipe [11]. We use the model provided by the authors, which is trained for 56,000 steps.
- **DiffPrep-Fix (DP-Fix)** and **DiffPrep-Flex (DP-Flex)** from DiffPrep, with the same configuration listed in [33].
- **RandomSearch (RS)** from DiffPrep, which uses the same logical pipeline of **DP-Fix** and randomly samples components for each type to generate 20 pipelines, then selects the one with the best validation accuracy.
- **SAGA**, with the same configuration listed in [49].

Notably, we do not consider the combination of HI-pipe and AI-pipe from HAIPipe as a competitor, since its pipeline generation algorithm requires a human-written pipeline structure as input, which does not exist in the test datasets.

7.2 Experimental Setup

Hardware and OS. We run our experiments on a server with 2 Intel Xeon Silver 4214R CPUs @ 2.4GHz, with 48 cores and 128GB of memory in total. All deep reinforcement learning-based setups

Table 4. Characteristics of the DiffPrep datasets and comparison of test accuracy.

	Data Characteristics					Test Accuracy								
Dataset	**#Inst.**	**#Feat.**	**#Lbl.**	**#Miss.**	**#Out.**	**(ES*)**	**DEF**	**RS**	**DP-Fix**	**DP-Flex**	**DL**	**HAI-AI**	**SAGA**	**CtxPipe**
abalone	4177	9	28	0	200	0.287	0.240	0.243	0.238	0.271	0.157	0.260	0.255	**0.287**
ada_prior	4562	15	2	88	423	0.857	0.848	0.844	**0.853**	0.846	0.803	0.801	0.833	0.818
avila	20867	11	12	0	4458	0.916	0.553	0.598	0.652	0.633	0.593	0.630	0.636	**0.759**
connect-4	67557	43	3	0	45873	0.792	0.659	0.671	0.726	0.702	0.683	**0.775**	0.758	0.763
eeg	14980	15	2	0	209	0.861	0.589	0.658	0.675	0.683	0.607	0.556	0.658	**0.740**
google	9367	9	2	1639	109	0.672	0.586	0.627	0.631	**0.661**	0.553	0.550	0.596	0.590
house	1460	81	2	6965	617	0.955	0.928	0.938	0.932	**0.952**	0.771	0.928	0.913	0.818
jungle_chess	44819	7	3	0	0	0.861	0.668	0.669	0.680	0.687	0.717	0.760	0.745	**0.861**
micro	20000	21	5	0	8122	0.634	0.564	0.579	0.595	0.593	0.613	**0.633**	0.556	0.605
mozilla4	15545	6	2	0	290	0.940	0.855	0.922	0.924	0.927	0.747	0.870	0.932	**0.940**
obesity	2111	17	7	0	25	0.927	0.775	0.841	**0.891**	0.874	0.590	0.768	0.751	0.868
page-blocks	5473	11	5	0	1011	0.973	0.942	0.959	0.959	**0.973**	0.940	0.935	0.849	0.965
pbcseq	1945	19	2	1445	99	0.866	0.710	0.730	0.728	0.725	0.680	0.733	**0.866**	0.805
pol	15000	49	2	0	8754	0.977	0.884	0.879	0.903	0.916	0.873	0.916	0.888	**0.949**
run_or_walk	88588	7	2	0	8548	0.990	0.719	0.829	0.903	0.912	0.820	0.915	0.832	**0.956**
shuttle	58000	10	7	0	5341	1.000	0.964	0.996	0.998	0.999	0.790	0.951	0.405	**1.000**
uscensus	32561	15	2	4262	2812	0.854	0.848	0.840	**0.854**	0.852	0.813	0.807	0.835	0.845
wall-robot-nav	5456	25	4	0	1871	0.962	0.697	0.872	0.905	0.913	0.927	0.896	0.841	**0.946**

are run on NVIDIA GeForce RTX 3090 GPUs with CUDA 11.2. The OS is Ubuntu 20.04 with Linux kernel 5.4.0-72. For SAGA, the maximum heap size of Java runtime is set to 80GB.

Datasets. We employ the same set of 18 real-world datasets from the OpenML data repository [56] used by DiffPrep [33] and refer to it as the *DiffPrep datasets*. These datasets are frequently utilized in both AutoML and data preparation research endeavors [8, 31, 33, 34]. Salient data characteristics of the selected datasets are presented in the Data Characteristics column group in Table 4. We further combine the DiffPrep datasets with 56 datasets from Deepline [21] excluding 8 where any baseline fails to generate the result. This results in 66 datasets, which we refer to as the *extended datasets*.

For baselines not including any dataset in their experimental studies, we adapt their codebase to accommodate the dataset as valid inputs. This involves generating necessary metadata for their parsing and inference processes, whenever required.

Components. Table 3 enumerates the candidate components and their respective types that are utilized in this work. These components are consistent with those employed in the recent state-of-the-art work by Chen et al. [11]. For each component type, the proposed pipeline architecture allows for either including a single component or omitting that specific type. The combination of components within the pipeline follows a flexible linear structure, wherein the order of the component types is not fixed.

Model. For the downstream ML model to train and evaluate, we use Logistic Regression as it is studied by all the baselines and is supported by all frameworks used by existing works, including PyTorch, TensorFlow, scikit-learn, and Apache SystemDS [10].

Evaluation Metrics. For the metric ϕ, we evaluate all generated pipelines using the accuracy of classification on the test dataset. This aligns with experimental studies of all the baselines. We also consider other metrics, including precision, recall, and F1-score. Since all labels are assigned a class in our test setup, these metrics are identical to accuracy with micro-averaging. Therefore, we use macro-averaging to ensure they offer distinct insights.

A typical data analytics lifecycle involves more than one pipeline. In particular, distribution shifts [32, 39, 47] can degrade the performance of the current pipeline, requiring constructing a new pipeline to adapt to the changes. As a result, we use pipeline runtime to evaluate the efficiency of an evolving data analytics process.

Table 5. Average results on the DiffPrep and extended datasets. Metrics include (1) test accuracy, its ranking, and running time (in seconds) (2) macro-averaged precision, recall, and F1-score per dataset (3) p-values for t-test (t) and sign test (sign).

	DiffPrep Datasets						Extended Datasets	
Setup	**Test Accuracy ↑**	**Ranking ↓**	**Running Time ↓**	**Precision ↑**	**Recall ↑**	**F1-score ↑**	**P-value (t) ↓**	**P-value (sign) ↓**
DEF	0.724	6.111	36.237	0.605	0.629	0.604	0.002	0.042
RS	0.761	4.889	803.888	0.647	0.673	0.650	0.001	0.003
DP-Fix	0.780	3.389	1024.845	0.681	0.727	0.687	0.035	0.013
DP-Flex	0.784	2.889	11148.911	0.697	0.722	0.695	0.044	0.042
DL	0.704	6.444	**9.291**	0.616	0.584	0.570	<0.001	<0.001
HAI-AI	0.760	4.722	22.195	0.705	0.619	0.613	0.007	0.024
SAGA	0.731	4.944	1384.824	0.647	0.640	0.618	0.015	0.003
CtxPipe	**0.806**	**2.389**	65.203	**0.774**	**0.736**	**0.745**	-	-

7.3 Performance Comparison

We first evaluate the performance of constructed pipelines on test datasets for all setups. This is measured by the evaluation metric ϕ and its ranking across setups per dataset.

Test Accuracy. Table 4 lists the test accuracy of the ML models trained on the preprocessed datasets via the generated pipelines from different setups. From the result, we can observe that CtxPipe achieves the best test accuracy on 9 out of 18 datasets, which outperforms all other setups on the consistency of the superiority, i.e., the number of datasets that obtain the best test accuracy is the highest. The overall average test accuracy increases from 2.2% to 10.2% across datasets, which is considered significant in the context of data preparation [21, 33]. Moreover, by integrating contextual information, under the same component search space, 15 out of 18 datasets get improved accuracy compared with HAI-AI, with 4 of them increased by over 10%, and the largest increased by 18.4%, which is significant. Notably, certain datasets such as `avila` and `eeg` receive significant improvements from CtxPipe. These datasets share the same pattern that differs from others, involving real values with small variances with different means across all columns. Also, the table header line suggests that each column in the table is a sub-feature of a complete measurement. Hence, this type of semantic can be considered as meaningful contextual information for the whole dataset, which is captured by the context plug-in to guide the agent to use the optimal components for this pattern.

Optimality Gap. Apart from the baselines, we also study how close different techniques are to the optimum. However, since the components' combinatorial space is prohibitively large, we thus approximate it for a dataset by extending the RS setup and running it 10,000 times, performing an *approximated Exhaustive Search* (ES*).

The values are recorded in Table 4. For most samples, we observe that the methods achieving the best performance are also close to the approximated optima. However, for those where CtxPipe significantly outperforms the baselines, e.g., `avila` and `eeg`, there are larger gaps toward the potential optimal solution. In other words, CtxPipe enables pipeline construction to balance more intelligently in the exploration-exploitation tradeoff, while the baselines are more likely to be stuck in the local optima without the ability to leverage contextual semantics to separate different cases.

Setup Ranking. Since the efficacy of data preparation varies across the datasets, we further conduct a granular analysis by ranking the competitors' effectiveness at the dataset level. Specifically, for each dataset, the test accuracy of all setups is ranked from one to eight. The average rank of each setup and the proportion of datasets the setup within top-K accuracy are calculated afterward. Our analysis concentrates on those ranked in the top 50% of all setups.

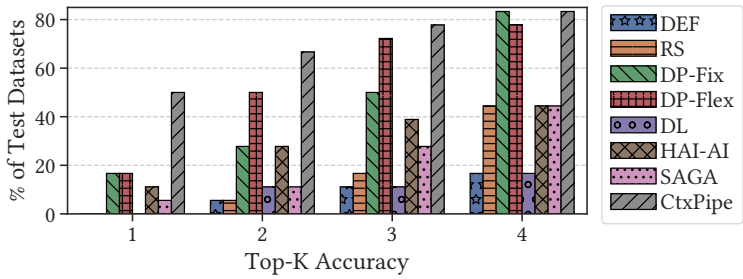

Fig. 8. Proportions of setups achieving top-K accuracy.

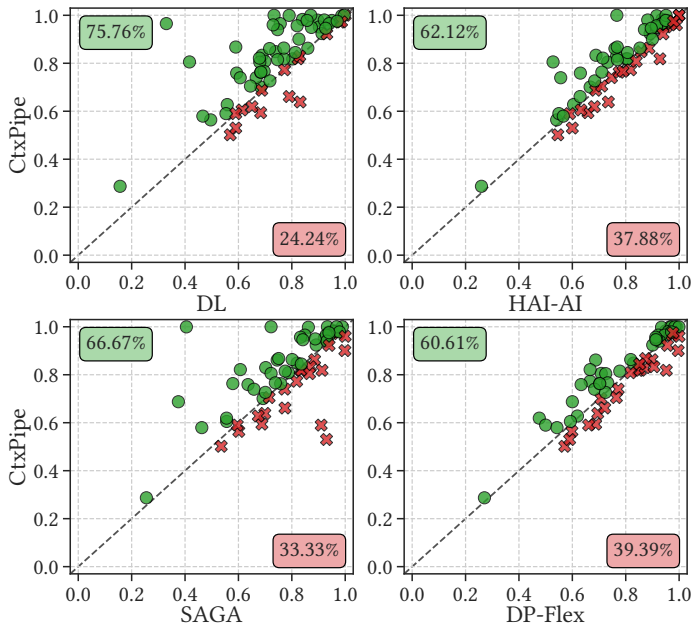

Fig. 9. Pair-wise performance comparison of CtxPipe vs. baselines among the extended datasets.

The average rank values are presented in the second column of Table 5. From the results, we see a significant improvement by CtxPipe from HAI-AI (4.722 -> 2.389), while CtxPipe outperforms all other setups on this criterion. Meanwhile, the ranking proportion results are plotted in Figure 8. From the plot, CtxPipe achieves a significantly higher top-1 ranking and continuously outperforms baselines for all top 50% K values. Moreover, despite sharing similar component candidates and main model architecture with HAI-AI, CtxPipe successfully increases the ranking proportion from the second tier (HAI-AI, SAGA, RS) to the first tier (CtxPipe, DP-Fix, DP-Flex), while incurring only modest overheads.

Other Metrics. To measure the quality of generated features more thoroughly, we report the performance on other metrics. In particular, we compute macro-averaged precision, recall, and F1-scores for each dataset, then report the average. The results in Table 5 show consistent improvements similar to that of test accuracy in all metrics. This further demonstrates the higher quality of the generated features over the baselines.

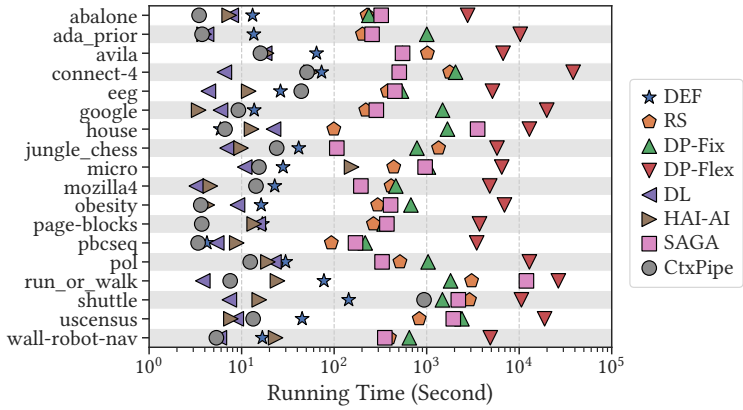

Fig. 10. Comparison of running time.

Statistical Significance. To study the statistical significance, we perform both qualitative and quantitative tests. First, we select four baseline setups (DL, HAI-AI, SAGA, DP-Flex) and perform pairwise comparisons on extended datasets. The results are depicted in Figure 9, where the test accuracy of the baseline setup and CtxPipe are plotted for each dataset. CtxPipe demonstrates superior performance on the majority of data distributions, outperforming the baselines on 60.61% to 75.76% of the datasets. The trend of improvements in both the average accuracy and rank are consistent with those on Diffprep datasets.

We conduct two quantitative metrics to verify the significance of the result. Following DeepLine [21], we use paired t-tests on the extended datasets. Moreover, to relax the assumption of normal distributions on the test results, we use the differences in the accuracy values to perform sign tests. The results, presented in Table 5, show that all p-values in both sign tests and t-tests are below 0.05. This indicates that CtxPipe has stable and significant accuracy improvements over all baselines.

7.4 Running Time and Cost-effectiveness

In addition to the overall quality of the generated data representations, the efficiency of the pipeline construction and inference is an important consideration. This is particularly crucial in production settings, where a continuous stream of datasets from diverse distributions must be processed and modeled expediently and efficiently [58]. To evaluate this, we further assess the cost-effectiveness tradeoff of the setups, that is, their data quality as well as the computational resources required to attain those quality levels. Specifically, we measure the end-to-end running time for constructing the pipeline and generating the data features. Notably, we do not separate these two phases as they alternate and are hence tightly coupled in the differentiable pipeline-based solutions.

Figure 10 plots for all setups the running time on each test dataset. Across all test datasets, CtxPipe shows low execution overheads compared with baselines. Moreover, the last column of Table 5 reports the average running time incurred by each setup when evaluated on the test datasets. Combining both results, we observe that the running time can be summarized by three clusters:

(1) Less than 10^2 seconds, including DEF, DL, HAI-AI, and CtxPipe. Except for DEF using a pre-defined physical pipeline, the deep reinforcement learning algorithms of other setups spend constant time selecting components. We note that although DL achieves very fast inference, its constructed pipelines perform the worst considering both accuracy and rank metrics.

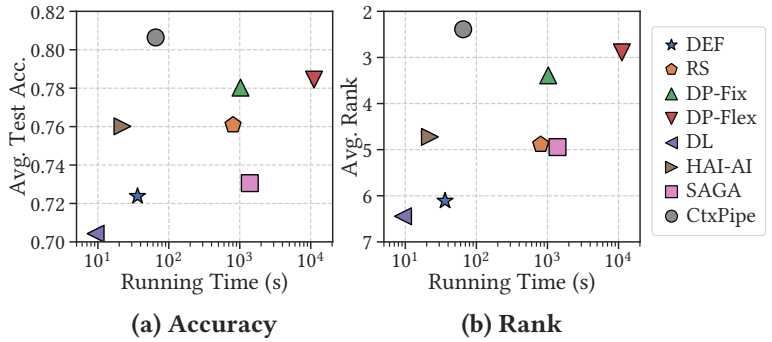

Fig. 11. Comparison of cost-effectiveness.

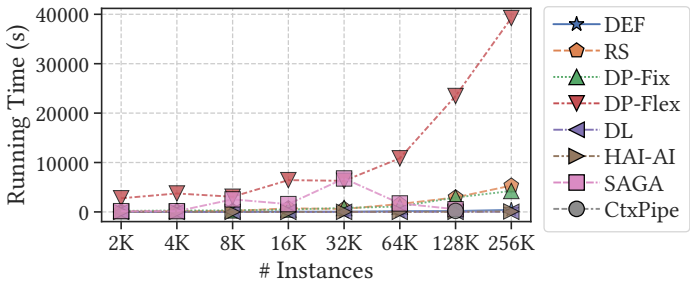

Fig. 12. Scalability under different dataset sizes.

(2) Closer to 10^3 seconds, including RS, DP-Fix, and SAGA. These setups rely on multiple pipeline executions per dataset and thus are slower by several times.
(3) Greater than 10^4 seconds, including DP-Flex. It explores a large component search space and computes complex gradients to update the optimization loss, greatly increasing the overhead.

Overall, the result demonstrates that CtxPipe is efficient in generating and executing the pipeline.

Cost-effectiveness. To further compare the cost-effectiveness, we visualize the gains in predictive performance, measured by the dual criteria of average test accuracy and ranking with respect to the increase in computational overhead. Figure 11 depicts the result. The scatter plots represent efficiency on the X-axis and performance on the Y-axis. An optimal method should generate high-quality feature representations with minimal running time overhead, i.e., mapping its output to the desirable upper left quadrant of the plot.

From the plots, we can observe that CtxPipe achieves the best performance on both criteria while incurring only relatively small increases in running time. Concretely, based on the worst performing setup, CtxPipe outperforms DP-Flex, the second-most performant setup, attaining 1.34x average test accuracy and 1.33x average ranking with 170.99x speedup in running time. Moreover, compared with HAI-AI, CtxPipe increases average accuracy by 82.14% and ranking by 135.48%, while the overhead remains within the same order of magnitude. This justifies the efficiency of the context integration mechanism and optimizations.

Combining the performance and running time, we can conclude that CtxPipe is cost-effective compared with baselines.

Table 6. Impact of the Context Plug-in (CP) and OCG experience replay (OCG) on pipeline construction performance. Metrics include test accuracy (Acc) and its ranking (Rank), macro-averaged precision (P), recall (R), and F1-score (F1).

Setup	Acc ↑	Rank ↓	P ↑	R ↑	F1 ↑
CtxPipe	**0.806**	**2.389**	**0.774**	**0.736**	**0.745**
CtxPipe (w/o CP & OCG)	0.750	4.278	0.725	0.645	0.654
CtxPipe (w/o OCG)	0.789	3.444	0.754	0.708	0.709

7.5 Scalability Test

To further measure the system's efficiency under different scales of user input datasets, we test the scalability of all setups with respect to the dataset sizes. Specifically, we chose 9 verified datasets[2] from OpenML data repository sizes ranging from 2K to 256K, where the sizes double between each two. For each size 2^iK where $i \in \{1, 2, ..., 8\}$, the actual number of instances is in range $[2^i\text{K}, 2^{i+1}\text{K}]$. All chosen datasets have column numbers between 7 and 12, rendering the scale mainly from the number of instances.

The result is shown in Figure 12. From the result, we can see that the dataset sizes have a stably increasing impact on RS, DP-Fix, and DP-Flex. For RS, this is because the random search requires a fixed number of pipeline executions. For DP-Fix and DP-Flex this is because each gradient descent step requires one pipeline execution. These reasons make the setups encounter the effect of slower execution when the number of dataset instances exceeds 32K. Notably, SAGA involves randomization in the grid search actions for component selection, resulting in a more unstable efficiency measurement. Our proposed CtxPipe uses iterative step execution that generates the pipeline directly, effectively reducing the influence of the dataset size on the execution.

7.6 Ablation Study

We conduct an ablation study on CtxPipe to test the isolated and quantitative efficacy of our contributions involving the context integration mechanism and the agent optimization strategy. We consider two ablated variants: (1) removing the context plug-in from the pipeline construction agent, coupled with disabling our proposed Open-Closed-Gated (OCG) experience replay by replacing it with normal experience replay (**w/o CP & OCG**), and (2) retaining the context plug-in while disabling the OCG experience (**w/o OCG**).

Context Plug-in. To ablate the context plug-in, we fix its gating value to 0 during inference and preclude gradient updates while performing agent optimization. This ensures that the randomly initialized agent model weights remain unchanged across both setups with and without the context plug-in. The results, presented in Table 6, show stable improvements in the prepared data features across all the metrics. In particular, incorporating the context plug-in yields a substantial improvement in the average rank of downstream model accuracy obtained using the generated features, decreasing from 4.278 to 2.389. This suggests that the performance gains facilitated by contextual awareness are uniformly distributed across diverse datasets, regardless of their data distributions.

OCG Experience Replay. We also record CtxPipe without OCG experience replay strategy to the training process of the agent including both the main DQN and the context plug-in. The result demonstrates the performance gain of this optimization strategy. The reason is that OCG experience replay forces the DQN model to make decisions with and without contextual information, helping the agent to scale the Q-values more intelligently. Moreover, we observe from the training losses among iterations that without OCG experience replay, the context plug-in is easier to stuck in

[2]The IDs of selected datasets are: 44091, 45039, 310, 45578, 42733, 45547, 133, 41752.

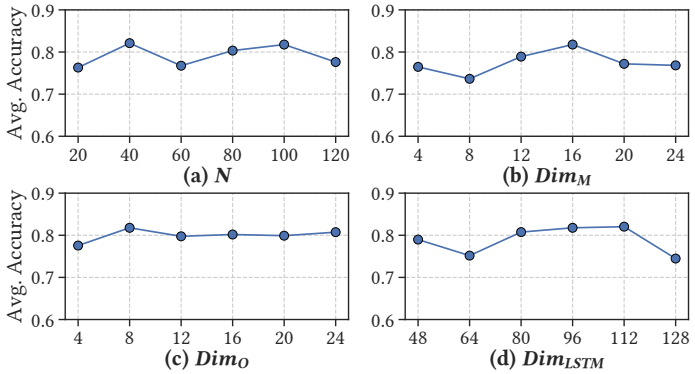

Fig. 13. Result of hyperparameter search.

Table 7. Comparison of non-linear pipeline construction setups and exploration-exploitation tradeoff policies.

Setup	Acc ↑	Rank ↓	P ↑	R ↑	F1 ↑
CtxPipe	0.806	**2.389**	**0.774**	0.736	**0.745**
CtxPipe (Non-linear, Row=2)	**0.807**	3.056	0.720	0.680	0.676
CtxPipe (Non-linear, Row=3)	0.776	3.500	0.720	0.691	0.677
CtxPipe (Non-linear, Row=4)	0.746	4.056	0.698	**0.764**	0.662
CtxPipe (Thompson Sampling)	0.747	4.312	0.559	0.530	0.582
CtxPipe (UCB)	0.746	4.778	0.678	0.595	0.586

local optima without accuracy changes from further training. This suggests that by iteratively controlling the passing of contextual information, OCG experience replay allows the agent to be more proactive in optimizing the main DQN network and could explore the global state-action approximation more thoroughly.

7.7 Discussion

Hyperparameter Sensitivity. CtxPipe uses several hyperparameters, similar to existing work [11]. To understand their impact on performance, we conduct a search over these hyperparameters using the extended datasets. The hyperparameters include the maximal number of columns N, the dimension of the model feature Dim_M, the dimension of the logical pipeline feature Dim_O, and the dimension of LSTM encoding Dim_{LSTM}. For each hyperparameter, we fix the other hyperparameters while varying the selected one in a constant sequence, and then measure the resulting accuracy.

The results shown in Figure 13 suggest that accuracy is stable across the hyperparameters, achieving the best performance at N=40, Dim_M=16, Dim_O=8, and Dim_{LSTM}=112, respectively. As a result, minimal tuning effort is required to achieve optimal performance. Notably, Dim_M is more critical to the performance, because M represents the ML model used to evaluate the pipeline, and thus its learning capacity has a larger effect on and tends to learn more from the accuracy value. We note that it is possible to optimize these hyperparameters for the main DQN model. However, it is orthogonal to our work which focuses on leveraging contextual information for higher-quality pipeline generation.

Non-linear Pipelines. The current design focuses on a linear pipeline structure with flexible component types. It can be extended to non-linear pipelines using the grid-based method proposed by DeepLine [21]. However, as previously noted, the learning capacity of DQN may be affected due

to the increased complexity which could degrade the model performance. To understand this, we extend CtxPipe with support for non-linear pipeline generation by adopting the grid-based method. The pipeline consists of a fixed number of *rows*, each of which is a flexible linear pipeline. Next, an ensemble component is selected by the model to merge the output of each row. We use the same architecture and vary the number of rows to test its performance. Table 7 shows the results. We observe that more rows lead to lower performance in terms of accuracy, ranking, and F1-score. This suggests that the model overfits the dataset features by repeatedly predicting similar components across rows. We leave the efficient solutions for non-linear pipeline construction for future works.

Exploration-exploitation Tradeoff. We compare our ϵ-greedy policy with methods that balance exploration and exploitation during training. In particular, we implement Thompson Sampling [6, 53] and Upper Confidence Bounds (UCB) [5, 50] as replacements for the ϵ-greedy policy. The result (Table 7) shows that these methods are less effective in our setup. We attribute this to the increased complexity of the learning algorithm that makes the model more difficult to train. In conclusion, common techniques on deep RL for pipeline construction are highly effective for CtxPipe.

Other Feature Types. To ensure fair comparisons with existing works, the current main DQN model only considers numerical and categorical features. We note that extended support for other types, such as text, can be implemented by adding text vectorizer components like `CountVectorizer` and `TfidfVectorizer` from scikit-learn into the candidates, allowing CtxPipe to learn to select these components for specific fields.

8 Conclusion

In this work, we present CtxPipe, a novel system supporting fully automated context-aware construction of data preparation pipelines. We design the framework and workflow that seamlessly integrates contextual information into the decision-making process of deep reinforcement learning via a context plug-in. It extracts contextual information from the transformed data with the text embedding model and uses it to manipulate the state-action inference. Furthermore, we optimize the workflow with our proposed Open-Closed-Gated experience replay, an enhanced reinforcement learning optimization strategy combined with the context plug-in. The experimental results on commonly adopted test datasets show that CtxPipe generates high-quality data features with significantly lower overhead than similar-performed state-of-the-art solutions.

Acknowledgments

This research is supported by the National Research Foundation, Singapore under its Emerging Areas Research Projects (EARP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting Data Errors: Where are we and what needs to be done? *Proc. VLDB Endow.* 9, 12 (2016), 993–1004.

[2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *VLDB J.* 24, 4 (2015), 557–581.

[3] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2017. Data Profiling: A Tutorial. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017.* ACM, Chicago, IL, USA, 1747–1751.

[4] Sercan Ö. Arik and Tomas Pfister. 2021. TabNet: Attentive Interpretable Tabular Learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021.* AAAI Press, Virtual Event, 6679–6687.

[5] Peter Auer. 2002. Using Confidence Bounds for Exploitation-Exploration Trade-offs. *J. Mach. Learn. Res.* 3 (2002), 397–422.

[6] Kamyar Azizzadenesheli, Emma Brunskill, and Animashree Anandkumar. 2018. Efficient Exploration through Bayesian Deep Q-Networks. *CoRR* abs/1802.04412 (2018), 40 pages.

[7] Microsoft Azure. 2024. *Azure Automated Machine Learning - AutoML.* Microsoft. Retrieved 2024-03-19 from https://azure.microsoft.com/en-us/products/machine-learning/automatedml/

[8] Laure Berti-Équille. 2019. Learn2Clean: Optimizing the Sequence of Tasks for Web Data Preparation. In *The World Wide Web Conference, WWW 2019.* ACM, San Francisco, CA, USA, 2580–2586.

[9] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient Denial Constraint Discovery with Hydra. *Proc. VLDB Endow.* 11, 3 (2017), 311–323.

[10] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *10th Conference on Innovative Data Systems Research, CIDR 2020.* www.cidrdb.org, Amsterdam, The Netherlands, 8 pages.

[11] 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 (2023), 91:1–91:26.

[12] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data Cleaning: Overview and Emerging Challenges. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016.* ACM, San Francisco, CA, USA, 2201–2206.

[13] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2020. TURL: Table Understanding through Representation Learning. *Proc. VLDB Endow.* 14, 3 (2020), 307–319.

[14] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2022. TURL: Table Understanding through Representation Learning. *SIGMOD Rec.* 51, 1 (2022), 33–40.

[15] Anna Fariha, Ashish Tiwari, Arjun Radhakrishna, Sumit Gulwani, and Alexandra Meliou. 2021. Conformance Constraint Discovery: Measuring Trust in Data-Driven Systems. In *SIGMOD '21: International Conference on Management of Data.* ACM, Virtual Event, 499–512.

[16] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015.* Curran Associates Inc., Montreal, Quebec, Canada, 2962–2970.

[17] Sainyam Galhotra, Anna Fariha, Raoni Lourenço, Juliana Freire, Alexandra Meliou, and Divesh Srivastava. 2022. DataPrism: Exposing Disconnect between Data and Systems. In *SIGMOD '22: International Conference on Management of Data.* ACM, Philadelphia, PA, USA, 217–231.

[18] Joseph Giovanelli, Besim Bilalli, and Alberto Abelló. 2021. Effective data pre-processing for AutoML. In *Proceedings of the 23rd International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP) co-located with the 24th International Conference on Extending Database Technology and the 24th International Conference on Database Theory (EDBT/ICDT 2021) (CEUR Workshop Proceedings, Vol. 2840).* CEUR-WS.org, Nicosia, Cyprus, 1–10.

[19] Yury Gorishniy, Ivan Rubachev, Valentin Khrulkov, and Artem Babenko. 2021. Revisiting Deep Learning Models for Tabular Data. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021.* Curran Associates, virtual, 18932–18943.

[20] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. 2022. Why do tree-based models still outperform deep learning on typical tabular data?. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022.* Curran Associates, New Orleans, LA, USA, 48 pages.

[21] 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 '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining.* ACM, Virtual Event, CA, USA, 2103–2113.

[22] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. 2013. Scalable Discovery of Unique Column Combinations. *Proc. VLDB Endow.* 7, 4 (2013), 301–312.

[23] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. TaPas: Weakly Supervised Table Parsing via Pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020.* Association for Computational Linguistics, Online, 4320–4333.

[24] Benjamin Hilprecht, Christian Hammacher, Eduardo Souza dos Reis, Mohamed Abdelaal, and Carsten Binnig. 2023. DiffML: End-to-end Differentiable ML Pipelines. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning, DEEM 2023.* ACM, Seattle, WA, USA, 7:1–7:7.

[25] Hiroshi Iida, Dung Thai, Varun Manjunatha, and Mohit Iyyer. 2021. TABBIE: Pretrained Representations of Tabular Data. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021.* Association for Computational Linguistics, Online, 3446–3456.

[26] Bojan Karlas, Peng Li, Renzhi Wu, Nezihe Merve Gürel, Xu Chu, Wentao Wu, and Ce Zhang. 2020. Nearest Neighbor Classifiers over Incomplete Information: From Certain Answers to Certain Predictions. *Proc. VLDB Endow.* 14, 3 (2020), 255–267.

[27] Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. 2009. Metric Functional Dependencies. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009*. IEEE Computer Society, Shanghai, China, 1275–1278.

[28] Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, and Eugene Wu. 2017. BoostClean: Automated Error Detection and Repair for Machine Learning. *CoRR* abs/1711.01299 (2017), 15 pages.

[29] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. 2016. ActiveClean: Interactive Data Cleaning For Statistical Modeling. *Proc. VLDB Endow.* 9, 12 (2016), 948–959.

[30] Sanjay Krishnan and Eugene Wu. 2019. AlphaClean: Automatic Generation of Data Cleaning Pipelines. *CoRR* abs/1904.11827 (2019), 15 pages.

[31] Erin LeDell and Sebastien Poirier. 2020. H2o automl: Scalable automatic machine learning. In *Proceedings of the AutoML Workshop at ICML*, Vol. 2020. ICML, PMLR, San Diego, CA, USA, 16 pages.

[32] Chonho Lee, Zhaojing Luo, Kee Yuan Ngiam, Meihui Zhang, Kaiping Zheng, Gang Chen, Beng Chin Ooi, and James Wei Luen Yip. 2017. Big Healthcare Data Analytics: Challenges and Applications. In *Handbook of Large-Scale Distributed Computing in Smart Healthcare*. Springer, Cham, Switzerland, 11–41.

[33] Peng Li, Zhiyi Chen, Xu Chu, and Kexin Rong. 2023. DiffPrep: Differentiable Data Preprocessing Pipeline Search for Learning over Tabular Data. *Proc. ACM Manag. Data* 1, 2 (2023), 183:1–183:26.

[34] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2021. CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks. In *37th IEEE International Conference on Data Engineering, ICDE 2021*. IEEE, Chania, Greece, 13–24.

[35] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards General Text Embeddings with Multi-stage Contrastive Learning. *CoRR* abs/2308.03281 (2023), 18 pages.

[36] Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. 2020. Approximate Denial Constraints. *Proc. VLDB Endow.* 13, 10 (2020), 1682–1695.

[37] Roque Lopez, Raoni Lourenço, Rémi Rampin, Sonia Castelo, Aécio S. R. Santos, Jorge Henrique Piazentin Ono, Cláudio T. Silva, and Juliana Freire. 2023. AlphaD3M: An Open-Source AutoML Library for Multiple ML Tasks. In *International Conference on Automated Machine Learning (Proceedings of Machine Learning Research, Vol. 224)*. PMLR, Hasso Plattner Institute, Potsdam, Germany, 22/1–22.

[38] Raoni Lourenço, Juliana Freire, and Dennis E. Shasha. 2020. BugDoc: Algorithms to Debug Computational Processes. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020*. ACM, Portland, OR, USA, 463–478.

[39] Zhaojing Luo, Sai Ho Yeung, Meihui Zhang, Kaiping Zheng, Lei Zhu, Gang Chen, Feiyi Fan, Qian Lin, Kee Yuan Ngiam, and Beng Chin Ooi. 2021. MLCask: Efficient Management of Component Evolution in Collaborative Data Analytics Pipelines. In *37th IEEE International Conference on Data Engineering, ICDE 2021*. IEEE, Chania, Greece, 1655–1666.

[40] Duncan C. McElfresh, Sujay Khandagale, Jonathan Valverde, Vishak Prasad C., Ganesh Ramakrishnan, Micah Goldblum, and Colin White. 2023. When Do Neural Nets Outperform Boosted Trees on Tabular Data?. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023*. Curran Associates, New Orleans, LA, USA, 34 pages.

[41] 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), 9 pages.

[42] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. 2016. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. ACM, Denver, CO, USA, 485–492.

[43] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *Proc. VLDB Endow.* 8, 10 (2015), 1082–1093.

[44] Abdulhakim Ali Qahtan, Nan Tang, Mourad Ouzzani, Yang Cao, and Michael Stonebraker. 2020. Pattern Functional Dependencies for Data Cleaning. *Proc. VLDB Endow.* 13, 5 (2020), 684–697.

[45] Alexandre Quemy. 2020. Two-stage optimization for machine learning workflow. *Inf. Syst.* 92 (2020), 101483.

[46] 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 *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019*. ACM, Amsterdam, The Netherlands, 1171–1188.

[47] Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. 2024. "We Have No Idea How Models will Behave in Production until Production": How Engineers Operationalize Machine Learning. *Proc. ACM*

Hum.-Comput. Interact. 8, CSCW1, Article 206 (apr 2024), 34 pages. https://doi.org/10.1145/3653697

[48] Ravid Shwartz-Ziv and Amitai Armon. 2022. Tabular data: Deep learning is not all you need. *Inf. Fusion* 81 (2022), 84–90.

[49] Shafaq Siddiqi, Roman Kern, and Matthias Boehm. 2023. SAGA: A Scalable Framework for Optimizing Data Cleaning Pipelines for Machine Learning Applications. *Proc. ACM Manag. Data* 1, 3 (2023), 218:1–218:26.

[50] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement learning - an introduction.* MIT Press, Massachusetts, USA.

[51] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. 2012. Fundamentals of Order Dependencies. *Proc. VLDB Endow.* 5, 11 (2012), 1220–1231.

[52] Nan Tang, Ju Fan, Fangyi Li, Jianhong Tu, Xiaoyong Du, Guoliang Li, Samuel Madden, and Mourad Ouzzani. 2021. RPT: Relational Pre-trained Transformer Is Almost All You Need towards Democratizing Data Preparation. *Proc. VLDB Endow.* 14, 8 (2021), 1254–1261.

[53] William R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3-4 (1933), 285–294.

[54] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013.* ACM, Chicago, IL, USA, 847–855.

[55] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence.* AAAI Press, Phoenix, Arizona, USA, 2094–2100.

[56] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luís Torgo. 2013. OpenML: networked science in machine learning. *SIGKDD Explor.* 15, 2 (2013), 49–60.

[57] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data X-Ray: A Diagnostic Tool for Data Errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* ACM, Melbourne, Victoria, Australia, 1231–1245.

[58] Doris Xin, Hui Miao, Aditya G. Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *SIGMOD '21: International Conference on Management of Data.* ACM, Virtual Event, China, 2639–2652.

[59] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020.* Association for Computational Linguistics, Online, 8413–8426.

[60] Gyeong-In Yu, Saeed Amizadeh, Sehoon Kim, Artidoro Pagnoni, Ce Zhang, Byung-Gon Chun, Markus Weimer, and Matteo Interlandi. 2021. WindTunnel: Towards Differentiable ML Pipelines Beyond a Single Model. *Proc. VLDB Endow.* 15, 1 (2021), 11–20.

Received April 2024; revised July 2024; accepted August 2024