Auto-Pipeline: Synthesizing Complex Data Pipelines By-Target Using Reinforcement Learning and Search

Junwen Yang
University of Chicago
junwen@uchicago.edu

Yeye He
Microsoft Research
yeyehe@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Recent work has made significant progress in helping users to automate *single* data preparation steps, such as string-transformations and table-manipulation operators (e.g., Join, GroupBy, Pivot, etc.). We in this work propose to automate *multiple* such steps end-to-end, by synthesizing complex data-pipelines with both string-transformations and table-manipulation operators.

We propose a novel *by-target* paradigm that allows users to easily specify the desired pipeline, which is a significant departure from the traditional *by-example* paradigm. Using by-target, users would provide input tables (e.g., csv or json files), and point us to a "target table" (e.g., an existing database table or BI dashboard) to demonstrate how the output from the desired pipeline would schematically "look like". While the problem is seemingly under-specified, our unique insight is that implicit table constraints such as FDs and keys can be exploited to significantly constrain the space and make the problem tractable. We develop an Auto-Pipeline system that learns to synthesize pipelines using deep reinforcement-learning (DRL) and search. Experiments using a benchmark of 700 real pipelines crawled from GitHub and commercial vendors suggest that Auto-Pipeline can successfully synthesize around 70% of complex pipelines with up to 10 steps.

PVLDB Reference Format:
Junwen Yang, Yeye He, and Surajit Chaudhuri. Auto-Pipeline: Synthesizing Complex Data Pipelines By-Target Using Reinforcement Learning and Search. PVLDB, 14(11): 2563 - 2575, 2021.
doi:10.14778/3476249.3476303

1 INTRODUCTION

Data preparation, sometimes also known as data wrangling, refers to the process of building sequences of table-manipulation steps (e.g., Transform, Join, Pivot, etc.), to bring raw data into a form that is ready for downstream applications (e.g., BI or ML). The end-result of data preparation is often a *workflow* or *data-pipeline* with a sequence of these steps, which are often then operationalized as recurring jobs in production.

It has been widely reported that business analysts and data scientists spend a significant fraction of their time on data preparation tasks (some report numbers as high as 80% [23, 24]). Accordingly, Gartner calls data preparation "the most time-consuming step in analytics" [45]. This is particularly challenging for less-technical users, who increasingly need to prepare data themselves today.

In response, significant progress has been made in the research community toward helping users author *individual* data preparation steps in data-pipelines. Notable efforts include automated data transformations (e.g., [17, 27, 30, 32]), table-joins (e.g., [38, 53]), and table-restructuring (e.g., [18, 34, 52]), etc.

In commercial systems, while pipelines are traditionally built manually (e.g., using drag-and-drop tools to build ETL pipelines), leading vendors have adopted recent advances in research and released features that make it really easy for users to build key steps in pipelines (e.g., automated transformation-by-example has been used in Excel [12], Power Query [6], and Trifacta [7]; automated join has been used in Tableau [15] and Trifacta [16], etc.).

Automating multi-step pipeline-building. While assisting users to build *single* data-prep steps (e.g., Transform, Join, etc.) is great progress, not much attention has been given to the more ambitious goal of automating *multi-step* pipeline-building end-to-end. We argue that building on top of recent success in automating single-steps such as [52], synthesizing *multi-step* pipelines has become feasible and will be an area that warrants more attention.

The key challenge in multi-step pipeline-synthesis is to allow users to easily specify the desired pipelines. Existing methods use the "by-example" paradigm (e.g., SQL-by-example [51] and Query-by-output [50]), which unfortunately requires a *matching* pair of input/output tables to be provided in order for the desired program (e.g., in SQL) to be synthesized. While by-example is easy-to-use for *row-to-row* string transformation [27, 30] (because users only need to type 2-3 example values), for *table-to-table* transformations this paradigm would unfortunately require users to manually enter an entire output table, which is not only significant overhead, but can also be infeasible for users to provide in many cases (e.g., when complex aggregations are required on large tables).

Furthermore, existing by-example approaches largely resort to some forms of exhaustive search, which unfortunately limits the richness of the operators they can support, also making these approaches frequently fail or time-out when synthesizing real pipelines with large amounts of data.

New paradigm: "by-target" pipeline-synthesis. In this work, we propose a new paradigm for multi-step pipeline-synthesis called *by-target*. We show that a "target" is easy for users to provide, yet it still provides a sufficient specification for desired pipelines to be synthesized. We emphasize that this novel paradigm is not studied before, and is a significant departure from the by-example approach.

Our key observation here is that a common usage pattern in pipeline-building (e.g., ETL) is to onboard new data files, such as sales data from a new store/region/time-period, etc., that are often formatted differently. In such scenarios, users typically have a

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476303

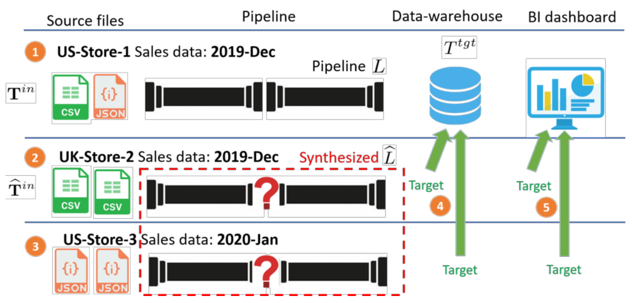


Figure 1: An example of pipeline-by-target. (1-3): Input tables from different time-periods/store-locations often have different formats and schema. (1): A pipeline previously built on one chunk of the input to produce database tables or BI dashboards. (2, 3): Instead of manually building pipelines for new chunks of input, we try to synthesize these pipelines by asking users to point us to a fuzzy "target" that can be (4) an existing table or (5) an existing visualization.

precise "target" in mind, such as an existing data-warehouse table, where the goal is to bring the new data into a form that "looks like" the existing target table (so that the new data can be integrated). Similarly, in building visualizations and dashboards for data analytics (e.g., in Tableau or Power BI), users can be inspired by an existing visualization, and want to turn their raw data into a visualization that "looks like" the given target visualization (in which case we can target the underlying table of the visualization).

Figure 1 illustrates the process visually. In this example, a large retailer has sales data coming from stores in different geographical regions and across different time-periods. Some version of the desired pipeline has been built previously – the top row of the figure shows a chunk of data for "`US-Store-1`" and "`2019-Dec`", and for this chunk there may already be a legacy script/pipeline from IT that produces a database table or a dashboard. However, as is often the case, new chunks of data for subsequent time-periods or new stores need to be brought on-board, which however may have different formats/schema (e.g., JSON vs. CSV, pivoted vs. relational, missing/extra columns, etc.), because they are from different point-of-sales systems or sales channels. Building a new pipeline manually for each such "chunk" (shown in the second/third row in the figure) is laborious, and especially challenging for less-technical users who may not have the skills to build such pipelines from scratch. Today these less-technical users often have to submit a ticket, and wait until IT has the bandwidth to serve their needs.

The aspirational question we ask, is whether pipelines can be synthesized automatically in such settings – if users could point us to a "target" that schematically demonstrates how the output should "look like", as shown with green arrows in Figure 1 that point to existing database tables or visualizations. Concretely, "targets" can be specified like shown in Figure 2, where users could right-click an existing database table and select the option to "`append data to the table`", or right-click an existing visualization and select "`create a dashboard like this`", to easily trigger a pipeline synthesis process.

Unlike by-example synthesis, "target" used in this new paradigm is only as a fuzzy illustration of user intent. Surprisingly, we show that this seemingly imprecise specification is in fact often sufficient to uniquely determine the desired pipeline – our insight is that implicit constraints such as FDs and Keys discovered from the target table are often sufficient to constrain the space of possible pipelines. This is a key property overlooked thus-far by existing work, which

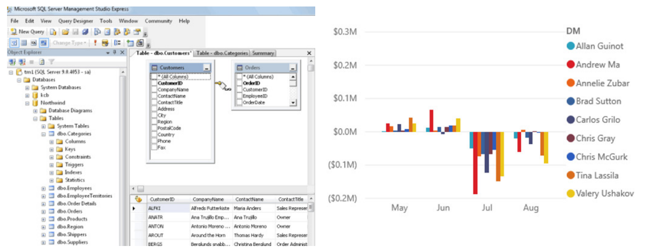


Figure 2: To trigger by-target synthesis, users only need to (Left): pick an existing database table, right click and select "Append data to this table", or (Right): point to a visualization, right click and select "Create a dashboard like this".

we argue can be the key to make pipeline-synthesis practical (because fuzzy "targets" are a lot easier for users to provide).

Search and RL-based Synthesis. The problem of synthesizing multi-step pipelines is clearly challenging, as the number of candidate pipelines grows exponentially in the number of steps, which is prohibitively large very quickly (reaching 10^{20} within 5 steps on typical tables having 10 columns).

In order to make synthesis tractable, we formalize the end-to-end synthesis as an optimization problem, and develop a search-based algorithm AUTO-PIPELINE-SEARCH that considers a diverse array of factors to best prioritize search over the most promising candidates.

We also design a deep reinforcement-learning (DRL) based synthesis algorithm AUTO-PIPELINE-RL, which "learns" to synthesize pipelines using large collections of real pipelines. Drawing inspiration from the success of using "self-play" to train game-playing agents like AlphaGo [47] and Atari [40], we use "self-synthesis" to train an agent by asking it to try to synthesize real pipelines, and rewarding it when it succeeds. It turns out that the RL-based synthesis can learn to synthesize fairly quickly, and slightly outperforms hand-crafted search using AUTO-PIPELINE-SEARCH.

2 MULTI-STEP BY-TARGET SYNTHESIS

We describe the by-target synthesis problem in this section, and we will start with preliminaries.

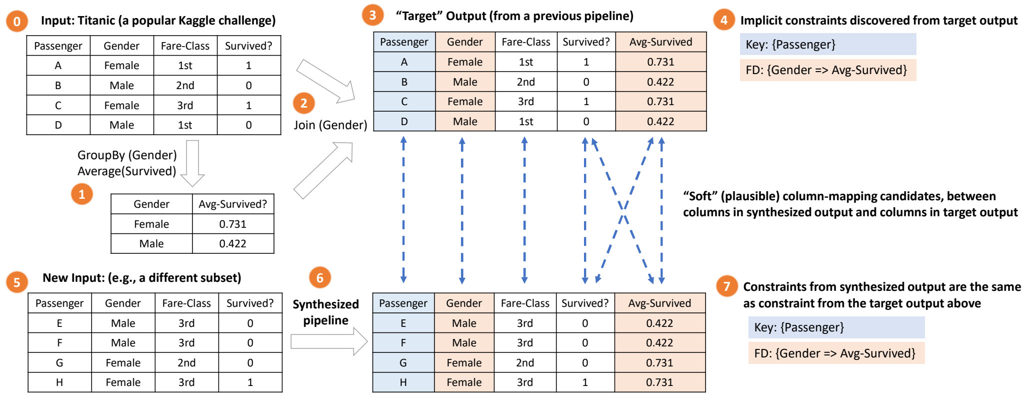
2.1 Preliminary: Pipelines and Operators

Data-pipelines. Data pipelines are ubiquitous today, to transform raw data into suitable formats for downstream processing. Step (0)-(3) of Figure 3 shows a conceptual pipeline using the `Titanic` table as input, which is a popular Kaggle task to predict which passengers survived [4]. The pipeline in this case performs (1) a GroupBy on the `Gender` column to compute `Avg-Survived` by `Gender`, and then (2) a Join of the result with the input table on `Gender`, so that in (3) `Avg-Survived` becomes a useful feature for predictions.

Today pipelines like this are built by both experts (e.g., developers and data-scientists) and less-technical users (e.g., end-users in tools like Power Query and Tableau Prep).

Expert users typically build pipelines using code/script, with Pandas [14] in Python being particularly popular for table manipulation. Figure 4(a) shows an example pipeline written in Pandas that corresponds to the same steps of Figure 3. Today a lot of these pipelines are written in Jupyter Notebooks [13] and are publicly available online. We crawled over 4M such notebooks on GitHub [52], from which we can extract large quantities of real data pipelines.

Less-technical users also increasingly need to build pipelines themselves today, typically using drag-and-drop tools (e.g., Power-Query, Informatica, Azure Data Factory, etc.) to manually specify


0
Input: Titanic (a popular Kaggle challenge)
GroupBy (Gender)
Average(Survived)
1
2
Join (Gender)
3
"Target" Output (from a previous pipeline)
4
Implicit constraints discovered from target output
Key: {Passenger}
FD: {Gender => Avg-Survived}
"Soft" (plausible) column-mapping candidates, between columns in synthesized output and columns in target output
5
New Input: (e.g., a different subset)
6
Synthesized pipeline
7
Constraints from synthesized output are the same as constraint from the target output above
Key: {Passenger}
FD: {Gender => Avg-Survived}

Passenger	Gender	Fare-Class	Survived?
A	Female	1st	1
B	Male	2nd	0
C	Female	3rd	1
D	Male	1st	0

Gender	Avg-Survived?
Female	0.731
Male	0.422

Passenger	Gender	Fare-Class	Survived?	Avg-Survived
A	Female	1st	1	0.731
B	Male	2nd	0	0.422
C	Female	3rd	1	0.731
D	Male	1st	0	0.422

Passenger	Gender	Fare-Class	Survived?
E	Male	3rd	0
F	Male	3rd	0
G	Female	2nd	0
H	Female	3rd	1

Passenger	Gender	Fare-Class	Survived?	Avg-Survived
E	Male	3rd	0	0.422
F	Male	3rd	0	0.422
G	Female	2nd	0	0.731
H	Female	3rd	1	0.731

Figure 3: An example pipeline to show why "by-target" provides a sufficient specification. Given (0) an input Titanic table from Kaggle to predict passenger survivals, a manually-authored pipeline performs (1) a GroupBy on "`Gender`" to compute "`Avg-Survived`" by "`Gender`", and then (2) Joins it back on "`Gender`". Imagine that users give the output table (3) as the "target", we can discover constraints such as (4) Key:{"`Passenger`"} and FD: {"`Gender`"$\rightarrow$ "`Avg-Survived`"}. (5) For a new input table (with a different set of passengers) and given (3) as the "target", intuitively a correctly synthesized pipeline in (6) should have the same FD/Key constraints that match the ones from the target table (3), like shown in (7).

pipelines step-by-step. Figure 4(b) shows an example pipeline with the same steps as Figure 3, but built in a visual drag-and-drop tool, which are more accessible to less-technical non-programmer users.

We note that the two pipelines in Figure 4 are equivalent, because they invoke the same sequence of *operators* (a GroupBy followed by Join). We introduce the notion of operators below.

Operators. Conceptually, data-pipelines invoke sequences of *operators* that broadly fall into two categories:

(1) *Table-level operators*: e.g., Join, Union, GroupBy, Pivot, Unpivot, etc. that manipulate tables. A subset of these operators are considered in SQL-by-example [50, 51].

(2) *String-level operators*: e.g., Split, Substring, Concatenate, etc., that perform string-to-string transformations. These operators are traditionally considered in transformation-by-example [29, 30].

In this work, we consider both classes of operators, since both are common in pipelines. Figure 5 shows the operators we consider, henceforth referred to as $\mathbf{O}$. Because these operators are fairly standard (e.g., the automation of individual operators are studied in depth in a prior work [52]), we defer descriptions of these operators to a full version of the paper [1].

Limitations. We note that expert users can write ad-hoc *user-defined functions* (e.g., any python code) in their pipelines, which are unfortunately intractable for program-synthesis even in simple cases (e.g., PSPACE-hard for arithmetic functions) [22, 25], and are thus not considered in our work. Similarly, we do not consider row-level filtering because it is also intractable in general [50].

2.2 Problem: Multi-step By-target Synthesis

As illustrated in Figure 1, in our pipeline synthesis problem, we are given as "target" an existing table T^{tgt} (e.g., a database table or a dashboard), generated from a pipeline L on a previous batch of input tables $\mathbf{T}^{in} = \{T_1, T_2, \ldots\}$, written as $T^{tgt} = L(\mathbf{T}^{in})$.

As is often the case, new data files, denoted by $\widehat{\mathbf{T}}^{in} = \{\widehat{T}_1, \widehat{T}_2, \ldots\}$, have similar content but may have different schema and representations (e.g., because they come from a different store/region/time-period, etc.). Users would want to bring $\widehat{\mathbf{T}}^{in}$ onboard, but L is no longer applicable, and often also not accessible[1].

In this work, we ask the aspirational question of whether new pipelines can be automatically synthesized, if users can point us to the new input files $\widehat{\mathbf{T}}^{in}$ and the target T^{tgt}, to schematically demonstrate what output from a desired pipeline should "look like". This by-target synthesis problem is defined as follows:

Definition 1. In *by-target pipeline-synthesis*, given input data $\widehat{\mathbf{T}}^{in}$, and a target table T^{tgt} generated from related input $\mathbf{T}^{in}$ that schematically demonstrates the desired output, we need to synthesize a pipeline $\widehat{L}$ using a predefined set of operators $\mathbf{O}$, such that $\widehat{T}^{o} = \widehat{L}(\widehat{\mathbf{T}}^{in})$ produces the desired output.

Evaluate synthesized pipelines from by-target. Since one may worry that a target-table T^{tgt} only provides a fuzzy specification of the synthesis problem, we will start by discussing how a by-target synthesis system can be systematically evaluated.

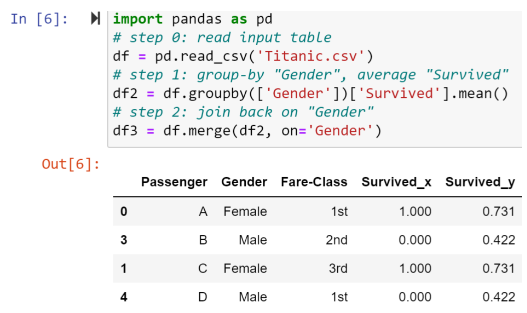
In traditional by-example synthesis (e.g., SQL-by-example [50, 51]), a pair of *matching* input/output tables $(\widehat{\mathbf{T}}^{in}, \widehat{T}^{o})$ is provided as input to synthesis algorithms (even though in practice $\widehat{T}^{o}$ is hard to come by). In such a setting, evaluating a synthesized program $\widehat{L}$ often reduces to a simple check of whether the synthesized output $\widehat{L}(\widehat{\mathbf{T}}^{in})$ is the same as $\widehat{T}^{o}$.

In by-target synthesis, we are given as input a pair of *non-matching* tables $(\widehat{\mathbf{T}}^{in}, T^{tgt})$, for which the same evaluation does not apply. It turns out, however, that evaluation by-target synthesis can be performed similarly, using what is analogous to "testing"/"training" in Machine Learning.

Specifically, as illustrated in Figure 6, for each real pipeline L authored by humans, we split the input tables used by L 50%/50% into "testing" and "training"[2]. We treat the the first 50% as if they are $\mathbf{T}^{in}$ in by-target synthesis, and use the ground-truth pipeline L to generate the target output $T^{tgt} = L(\mathbf{T}^{in})$. We then use the remaining 50% as if they are $\widehat{\mathbf{T}}^{in}$, and feed the non-matching pair $(\widehat{\mathbf{T}}^{in}, T^{tgt})$ as input to by-target synthesis (circled in dash in Figure 6), so

[1]End-users wanting to build a "similar" pipeline targeting an existing database-table/dashboard often do not have access to the original legacy pipelines L built by IT, due to discover-ability and permission issues. As such, to ensure generality, in this work we do not assume the original L to be available as reference to synthesize new pipelines (though we are clearly more likely to succeed if the original L is available).

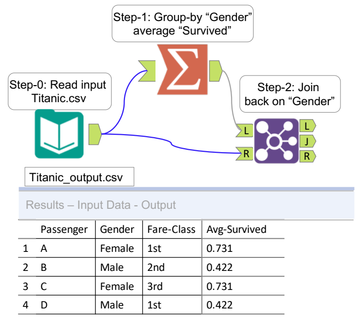
[2]When there are multiple tables in a pipeline and Joins are required, we split the largest input table (which is fact-table-like) to ensure that Joins do not produce empty results.



```
In [6]:  import pandas as pd
         # step 0: read input table
         df = pd.read_csv('Titanic.csv')
         # step 1: group-by "Gender", average "Survived"
         df2 = df.groupby(['Gender'])['Survived'].mean()
         # step 2: join back on "Gender"
         df3 = df.merge(df2, on='Gender')
Out[6]:
```

	Passenger	Gender	Fare-Class	Survived_x	Survived_y
0	A	Female	1st	1.000	0.731
3	B	Male	2nd	0.000	0.422
1	C	Female	3rd	1.000	0.731
4	D	Male	1st	0.000	0.422

(a) A pipeline authored using Python Pandas


Step-1: Group-by "Gender" average "Survived"
Step-0: Read input Titanic.csv
Step-2: Join back on "Gender"
L
R
L
J
R
Titanic_output.csv
Results – Input Data - Output

	Passenger	Gender	Fare-Class	Avg-Survived
1	A	Female	1st	0.731
2	B	Male	2nd	0.422
3	C	Female	3rd	0.731
4	D	Male	1st	0.422

(b) A pipeline authored in visual drag-and-drop tool

Figure 4: Example pipelines corresponding to the same steps in Figure 3. (a): A pipeline built by data scientists using Python Pandas in a Jupyter Notebook. (b): The same pipeline built by less-technical users using visual drag-and-drop tools.

Table-reshaping operators	Join, Union, GroupBy, Agg, Pivot, Unpivot, Explode
String-transformation operators	Split, Substring, Concatenate, Casing, Index

Figure 5: Operators considered in by-target synthesis.

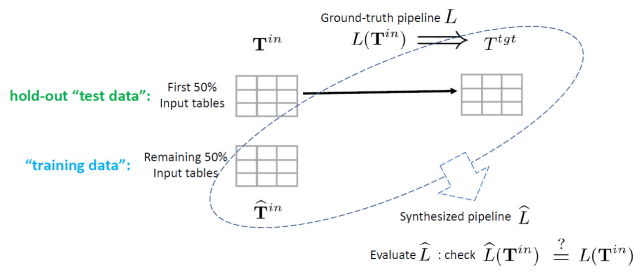

Ground-truth pipeline L
$\mathbf{T}^{in}$
$L(\mathbf{T}^{in}) \Longrightarrow T^{tgt}$
hold-out "test data":
First 50% Input tables
"training data":
Remaining 50% Input tables
$\widehat{\mathbf{T}}^{in}$
Synthesized pipeline $\widehat{L}$
Evaluate $\widehat{L}$: check $\widehat{L}(\mathbf{T}^{in}) \overset{?}{=} L(\mathbf{T}^{in})$

Figure 6: Evaluate by-target synthesis: Given a human-authored pipeline L, we treat the first 50% of input data for L as $\mathbf{T}^{in}$, to generate the target table $T^{tgt} = L(\mathbf{T}^{in})$. We then use the remaining 50% of input as $\widehat{\mathbf{T}}^{in}$, which together with T^{tgt}, is fed into by-target synthesis to synthesize a new pipeline $\widehat{L}$. The correctness of $\widehat{L}$ can be verified on $\mathbf{T}^{in}$ (held-out during synthesis), by checking whether $\widehat{L}(\mathbf{T}^{in}) \overset{?}{=} L(\mathbf{T}^{in})$.

that a new pipeline $\widehat{L}$ can be synthesized. The correctness of the synthesized $\widehat{L}$ can be verified on the first 50% data ($\mathbf{T}^{in}$), which is held-out during synthesis, by checking whether $\widehat{L}(\mathbf{T}^{in}) \overset{?}{=} L(\mathbf{T}^{in})$. Note that because $\mathbf{T}^{in}$ is held-out during synthesis (analogous to hold-out test-data in ML), and the original pipeline L is also held-out, the fact that we can "reproduce" a synthesized $\widehat{L}$ that has the same effect as L on the hold-out data $\mathbf{T}^{in}$ ensures that the synthesized $\widehat{L}$ from by-target is indeed what users want.[3]

Is by-target a sufficient specification? Even though by-target synthesis can be systematically evaluated using a procedure analogous to train/test in ML, one may still wonder whether a non-matching pair $(\widehat{\mathbf{T}}^{in}, T^{tgt})$ in by-target synthesis provides a sufficient specification for a desired pipeline to be synthesized. We show that this seemingly imprecise specification is in fact sufficient in most cases, by leveraging *implicit constraints* that we can discover from T^{tgt}. We illustrate this using the following example.

Example 1. Figure 3 shows the conceptual steps of a simple pipeline for the Titanic challenge [5]. Like we discussed in Section 2.1, this particular pipeline computes (1) a GroupBy on the `Gender` column to compute `Avg-Survived` by `Gender`, and then (2) a Join on `Gender` to bring `Avg-Survived` as an additional feature into the original input, like shown in (3).

In our setting of by-target synthesis, a different user is now given a similar input table with a different set of passengers like shown in (5). Without having access to the original pipeline, she points to (3) as the target table to as a fuzzy demonstration of her desired output, in order for by-target synthesis to produce the desired pipeline.

Our key insight is that in such cases, the desired pipeline can be uniquely determined, by leveraging *implicit constraints* discovered from the output table (3). Specifically, we can apply standard constraint-discovery techniques (e.g., [42]) to uncover two constraints shown in (4): Key-column:{"`Passenger`"}, Functional-dependency (FD): {"`Gender`" $\longrightarrow$ "`Avg-Survived`"}.

When table (5) is used as the new input and table (3) is used as the target, implicitly we want a synthesized pipeline (6) to follow the same set of transformations in the pipeline that produces (3), and as such the new output using table (5) as input should naturally satisfy the same set of constraints. Namely, if we perform a column-mapping between the table (3) and table (6), we can see that the constraints discovered from these two tables, as shown in (4) and (7), have direct one-to-one correspondence. If we need to recreate these implicit constraints in table (3) in a synthesized pipeline, it can be shown that the only pipeline with the fewest steps to satisfy all these constraints is the aforementioned pipeline. (Others would either miss one constraint, or require more steps, which are less likely to be desired according to MDL and Occam's Razor [26]). [4]

In summary, our key insight is that leveraging implicit constraints can sufficiently constrain the synthesis problem. Our large-scale evaluation on real pipelines (Section 5) confirms that most can indeed be successfully synthesized using the by-target paradigm.

2.3 Synthesis Algorithm: Intuitive Sketch

We now give a sketch of how a synthesis algorithm may look like before we formalize the problem.

[3]Note that we do not require $\widehat{L}$ and L to be identical at a syntactical-level, because there are often semantically equivalent ways to rewrite a pipeline (e.g., change of operator orders, or rewrite using an equivalent sequence).

[4]We note that while the synthesized pipeline in the example of Figure 3 is the same as the original, there are many cases where synthesized pipelines are different from the original, while still being semantically equivalent. We defer this to Section 5.4.

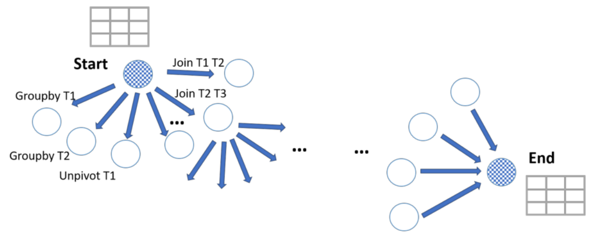


Figure 7: A search graph for synthesis: from the start-node (an empty pipeline) to the end-node (a synthesized pipeline), each intermediate node represents a partial pipeline, and each edge represents the act of adding one operator, which leads to a new pipeline with one more operator.

Figure 7 gives an intuitive illustration of the synthesis process. Each node here represents an intermediate state in the synthesis process, which corresponds to a "partial pipeline". The starting state (shown with a checkerboard pattern at the top-left) corresponds to an empty pipeline $\widehat{L} = \{\}$, and the ending state (at bottom-right) corresponds to a final synthesized pipeline $\widehat{L} = \{O_1, O_2, \dots O_n\}$.

From each state representing a partial pipeline, we can extend the partial pipeline by one additional "step" using some operator $O \in \mathbf{O}$ in Figure 5, to move to a subsequent state. For example, from the starting state $\widehat{L} = \{\}$, we can add different instantiations of operators in **O** (e.g., different ways to apply GroupBy/Join/Pivot, etc., on given input tables), which lead to different one-step pipelines (e.g., $\widehat{L}$ = {GroupBy(`table-1`, `column-1`)}). This synthesis process can then be visualized as traversing the search graph, until a satisfactory end-state is reached (e.g., satisfying all implicit constraints).

It is clear from this intuitive sketch, however, that the search space of possible pipelines is prohibitively large, because (1) the number of possible pipelines grows exponentially with the number of steps; and (2) even one individual step can be parameterized in numerous ways – e.g., a Join between two tables with $|C|$ columns each can in theory use any of the $|C|^2$ column-pairs as the Join key (the same is true for GroupBy/Pivot, etc.).

While we will defer a description of our solution to (1) above, solving (2) is relatively straightforward because for each operator (e.g., Join), we can leverage existing work (e.g., [52]) to accurately predict the most likely way to parameterize the operator given input tables (e.g., which columns to Join/GroupBy/Pivot, etc.).

Predict Single-Operator Parameters. Conceptually, for each operator $O \in \mathbf{O}$, and given input tables T, we need to predict the likelihood of using parameter p for O in the context of T, written as $P_T(O(p))$. For instance, for a Join between two given tables, we need consider the characteristics of the tables to estimate which columns will likely join (which is a Join parameter); similarly for Unpivot, we need to consider input tables and predict which subset of columns should Unpivot (also a parameter), etc.

For this reason, we build upon a prior technique called Auto-Suggest [52], which learns from real data pipelines to predict the likelihood of using parameters p for each operator O given input tables T, which is exactly $P_T(O(p))$. In this work, we leverage [52] and treat these $P_T(O(p))$ as given, to better focus on the end-to-end pipeline synthesis problem. We refer readers to [52] for details of these single-operator predictions in the interest of space.

Optimization-based formulation. Given the probabilistic estimates of operator parameters $P(O(p))$, and the fact that we want to synthesize a pipeline that can satisfy all implicit constraints (FD/Key), we formulate the synthesis as an optimization problem. Specifically, we want to find the "most likely" pipeline $\widehat{L}$ consisting of a sequence of suitably parameterized operators $\widehat{L} = \{O_1(p_1), O_2(p_2), \dots\}$[5], by maximizing the joint probabilities of these operators $O_i(p_i)$, under the constraints that output from $\widehat{L}$ should satisfy all implicit constraints. This problem, henceforth referred to as PMPS (probability-maximizing pipeline synthesis), can be written as follows:

$$\text{(PMPS)} \quad \underset{\widehat{L}}{\arg\max} \prod_{O_i(p_i) \in \widehat{L}} P(O_i(p_i)) \tag{1}$$

$$\text{s.t. } \mathrm{FD}(\widehat{L}(\widehat{\mathbf{T}}^{in})) = \mathrm{FD}(T^{tgt}) \tag{2}$$

$$\mathrm{Key}(\widehat{L}(\widehat{\mathbf{T}}^{in})) = \mathrm{Key}(T^{tgt}) \tag{3}$$

$$\text{Col-Map}(\widehat{L}(\widehat{\mathbf{T}}^{in}), T^{tgt}) \tag{4}$$

The objective function in Equation (1) states that we want to find the most likely pipeline $\widehat{L}$, or the one whose joint probability of all single-step operator invocations is maximized. Equation (2) and (3) state that when running the synthesized pipeline $\widehat{L}$ on the given input $\widehat{\mathbf{T}}^{in}$ to get $\widehat{L}(\widehat{\mathbf{T}}^{in})$, the FD/Key constraints discovered from T^{tgt} should also be satisfied on $\widehat{L}(\widehat{\mathbf{T}}^{in})$. Finally Equation (4) states that we should be able to "map" columns from $\widehat{L}(\widehat{\mathbf{T}}^{in})$ to T^{tgt}, with standard schema-mapping [44].

Example 2. We revisit Figure 3. Using [52], we estimate the probabilities $P(O(p))$ of the two steps in the pipeline (GroupBy and Join) to be 0.4 and 0.8, respectively. Among all other possible pipelines, this two-step pipeline maximizes the joint probability (0.32) in Equation (1), while satisfying all FD/Key/column-mapping constraints in Equation (2)-(4), which is thus the solution to PMPS.

3 SEARCH-BASED AUTO-PIPELINE

This section describes our synthesis using Auto-Pipeline-Search.

3.1 A High-level Overview

As discussed in Section 2.3, at a high level the synthesis process can be seen as traversing a large search graph shown in Figure 7. Because each node corresponds to a partial-pipeline, and each edge corresponds to the act of adding one operator, each node that is *depth*-steps away from the start-node would naturally correspond to a partial-pipeline with *depth* number of operators/steps.

Given the large search graph, it is natural to explore only "promising" parts of the graph. We first describe such a strategy in a meta-level synthesis algorithm shown in Algorithm 1 below, which uses a form of beam search [41].

Algorithm 1 starts by initializing $depth = 0$ to indicate that we are at the start-node in Figure 7. The variable *candidates* stores all "valid" pipelines satisfying the constraints in PMPS (Equation (2)-(4)), and is initialized as an empty set. The variable S_{depth} corresponds to all pipelines with *depth*-steps that are actively explored

[5] While pipelines are in general directed acyclic graphs (DAGs), they can be serialized into sequences of invocations, thus the simplified notation.

Algorithm 1 Synthesis: A meta-level synthesis algorithm

```
1:  procedure Synthesis($\widehat{\mathrm{T}}^{in}, T^{tgt}$, O)
2:      $depth \leftarrow 0, candidates \leftarrow \emptyset$
3:      $S_{depth} \leftarrow \{empty()\}$                ▷ #initialize an empty pipeline
4:      while $depth < maxDepth$ do
5:          $depth \leftarrow depth + 1$
6:          for each ($L \in S_{depth-1}, O \in \mathbf{O}$) do
7:              $S_{depth} \leftarrow S_{depth} \cup$ AddOneStep($L, O$)
8:          $S_{depth} \leftarrow$ GetPromisingTopK($S_{depth}, T^{tgt}$)
9:          $candidates \leftarrow candidates \cup$ VerifyCands($S_{depth}, T^{tgt}$)
10:     return GetFinalTopK($candidates$)
```

in one loop iteration, and at line 3 we initialize it to a single placeholder empty-pipeline, because it is the only 0-step pipeline and we are still at the start-node of the search graph.

From line 4, we iteratively visit nodes that are $depth = \{1, 2, ...\}$ steps away from the start-node, which is equivalent to exploring all pipelines with $\{1, 2, ...\}$ operators. As we increment $depth$ in the loop, we take all active pipelines from the previous iteration with ($depth - 1$) steps, denoted by $S_{depth-1}$, and "extend" each partial pipeline $L \in S_{depth-1}$ using one additional operator $O \in \mathbf{O}$, by invoking AddOneStep(L, O), which is shown at line 7. These resulting pipelines with $depth$-steps are saved as S_{depth}. Because we cannot exhaustively explore all pipelines in S_{depth}, at line 8, we select top-K most promising ones from S_{depth} by invoking GetPromisingTopK(). Among these top-K promising partial pipelines, we check whether any of them already satisfy PMPS constraints using VerifyCand(), and save the feasible solutions separately into $candidates$ (line 9). This marks the end of one iteration.

We continue with the loop and go back to line 4, where we increment $depth$ by 1 and explore longer pipelines, until we find enough number of valid candidates, or we reach the maximum depth, at which point we return the final top-K candidate pipelines by invoking GetFinalTopK() (line 10).

Discussion. While the key steps in our synthesis are sketched out in Algorithm 1, we have yet to describe the sub-routines below:

- AddOneStep() extends a partial pipeline L using one additional operator $O \in \mathbf{O}$;
- VerifyCands() checks whether pipelines satisfy PMPS constraints, and if so marks them as final candidates;
- GetPromisingTopK() selects the most promising K pipelines from all explored pipelines with $depth$-steps;
- GetFinalTopK() re-ranks and returns final pipelines.

The first two sub-routines, AddOneStep() and VerifyCands(), are reasonably straightforward – AddOneStep() adds one additional step into partial pipelines by leveraging Auto-Suggest [52] to find most likely parameters for each operator, while VerifyCands() checks for PMPS constraint using standard FD/key-discovery [20, 42] and column-mapping [44]. We will describe these two sub-routines in Section 3.2 and 3.3, respectively.

The last two sub-routines, GetPromisingTopK() and GetFinalTopK(), are at the core of Auto-Pipeline, where a good design ensures that we can efficiently search promising parts of the graph and synthesize successfully. In Section 3.4, we will describe a search-based strategy to instantiate these two sub-routines, and later in Section 4, we will describe a learning-based alternative using RL.

3.2 Extend pipelines by one step

We describe the AddOneStep() subroutine in this section. AddOneStep(L, O) takes as input a $depth$-step partial pipeline $L = \{O_1(p_1), \ldots, O_{depth}(p_{depth})\}$, and some operator O (enumerated from all possible operators **O**) that we want to add into L. We leverages [52], which considers the characteristics of intermediate tables in the partial pipeline L, to predict the best parameter $p = \arg\max_{p \in \mathbf{p}} P(O(p)|L)$ to use. We use this predicted parameter p to instantiate the new operator O, and use the resulting $O(p)$ to extend L by one additional step, producing $L' = \{O_1(p_1), \ldots O_{depth}(p_{depth}), O(p)\}$.

Note that in general, for each operator O, there may be more than one good way to parameterize O (e.g., there may be more than one plausible GroupBy column, and more than one good Join column, etc.). So instead of using only top-1 predicted parameter, for each O we keep top-M most likely parameters, which would produce M possible pipelines after invoking AddOneStep(L, O) for a given L and O.

We use the following example to illustrate the process.

Example 3. We revisit the pipeline in Figure 3. At step (0), we have one input table and an empty pipeline $L = \{\}$. We enumerate all possible operators $O \in \mathbf{O}$ to extend L.

Suppose we first pick O to be GroupBy. Intuitively we can see that `Gender` and `Fare-Class` columns are the most likely for GroupBy (because among other things these two columns have categorical values with low cardinality). We leverage single-operator predictors from [52] – in this case we use the GroupBy predictor (Section 4.2 of [52]), which may predict that $P(GroupBy(\texttt{Fare-Class})|L) = 0.5$ and $P(GroupBy(\texttt{Gender})|L) = 0.4$ to be the most likely. If we use $M = 2$ or keep top-2 parameters for each operator, this leads to two new 1-step pipelines $L'_1 = \{GroupBy(\texttt{Fare-Class})\}$ and $L'_2 = \{GroupBy(\texttt{Gender})\}$.

The same process continues for other $O \in \mathbf{O}$. For instance when we pick O to be "Pivot", we may predict that `Gender` and `Fare-Class` to be likely Pivot keys, so we get $L'_3 = \{Pivot(\texttt{Gender})\}$, $L'_4 = \{Pivot(\texttt{Fare-Class})\}$.

However, when we pick O to be Join/Union, the probabilities of all possible parameters are 0, because no parameter is valid with only one input table in L. This changes when we have more intermediate tables – e.g., in a subsequent step marked as (1) in Figure 3, a new intermediate table is generated from GroupBy. At that point, using [52] we may predict a Join using `Gender` to be likely, while a Union is unlikely (because of the schema difference).

3.3 Verify constraint satisfaction

We now describe VerifyCands() in this section. Recall that VerifyCands(S_{depth}, T^{tgt}) takes as input a collection of pipelines S_{depth} (the set of synthesized pipelines with $depth$ steps), and check if any $\widehat{L} \in S_{depth}$ satisfy all constraints listed in Equation (2)-(4) for Key/FD/column-mapping, in relation to the target table T^{tgt}.

Column-mapping. For column-mapping, we apply standard schema-mapping techniques [44] to find possible column-mapping between the target table T^{tgt}, and the output table from a synthesized pipeline $\widehat{L}(\widehat{\mathbf{T}}^{in})$, using a combination of signals from column-names and column-values/patterns. In the interest of space, we defer details of this to a full version of the paper [1], but we give an example below for illustration.

Example 4. Consider a synthesized pipeline $\widehat{L}$ that produces an output table shown at step (6) of Figure 3. Recall that our target table T^{tgt} is shown in step (3) – for a synthesized $\widehat{L}$ to be successful, its output $\widehat{L}(\widehat{\mathbf{T}}^{in})$ should "cover" all columns in the target T^{tgt}, as required in Equation (4). As such, we need to establish a column-mapping between the table in (3) and the table in (6).

Using standard schema-mapping techniques, we find column-to-column mapping shown in Figure 3, using a combination of signals from column-values, column-patterns, and column-names.

It should be noted that our mapping is *not* required to be hard 1:1 mapping, but can be *"soft"* 1:N mapping. Like shown in Figure 3, the `Avg-Survived` column may be mapped to both `Survived` and `Avg-Survived` in the other table, since both share similar column-names and values, and can be plausible mapping candidates. In the end, so long as columns in T^{tgt} can be "covered" by some plausible mapping candidates in the synthesized result $\widehat{L}(\widehat{\mathbf{T}}^{in})$, this synthesized pipeline $\widehat{L}$ is deemed to satisfy the column-mapping constraint in Equation (4).

FD/Key constraints. For FD/Key constraints, we again apply standard-constraint discovery techniques [20, 42], to discover FD/Key constraints from both the target table T^{tgt}, and the output table $\widehat{L}(\widehat{\mathbf{T}}^{in})$ from a synthesized pipeline $\widehat{L}$, in order to see if all FD/Key constraints from T^{tgt} can be satisfied by $\widehat{L}$. We use the example below to illustrate this.

Example 5. Given a synthesized pipeline $\widehat{L}$ that produces an output table shown at step (6) of Figure 3, and a target table T^{tgt} shown in step (3), we use constraint-discovery to discover Key/FD constraints, which are shown in (7) and (4) for these two tables, respectively. Given the soft column-mapping candidates shown in Figure 3, we can see that there exists one column-mapping (with `Survived` $\leftrightarrow$ `Survived` and `Avg-Survived` $\leftrightarrow$ `Avg-Survived`), under which all Key/FD constraints from the target table (3) can be satisfied by the ones in table (6), thus satisfying Equation (2) and (3).

Given that Key/FD/column-mapping have all been satisfied for the output in (6), in VerifyCands() we can mark the corresponding pipeline $\widehat{L}$ as a feasible solution to PMPS (and saved in the variable *candidate* in Algorithm 1).

3.4 A diversity-based search strategy

We now describe GetPromisingTopK() and GetFinalTopK(), which are at the core of Auto-Pipeline-Search.

Recall that our goal is to solve the optimization problem PMPS in Section 2.3, which requires us to find a pipeline that can (1) maximize overall joint operator probabilities in the synthesized pipeline (the objective function in Equation (1)), and (2) satisfy constraints in Equation (2)-(4).

Because each candidate pipeline has already been checked for constraint satisfaction (Equation (2)-(4)) in VerifyCands(), this makes GetFinalTopK() easy as we only need to pick candidate pipelines that maximize joint operator probabilities. That is, for a synthesized pipeline $\widehat{L} = \{O_1(p_1), O_2(p_2), \ldots\}$, we can calculate its joint operator probabilities as $P(\widehat{L}) = \prod_{O_i(p_i) \in \widehat{L}} P(O_i(p_i))$, where $P(O_i(p_i))$ are estimates from single-operator models in [52]. We can output a ranked list of top-K pipelines, by simply ranking all candidate pipelines using $P(\widehat{L})$.

On the other hand, the sub-routine GetPromisingTopK() evaluates all *depth*-step pipelines currently explored, where we need to find top-K promising candidates in order to prune down the search space. We note that GetPromisingTopK() could not use the same strategy as GetFinalTopK() by simply maximizing $P(\widehat{L})$, because this may lead to pipelines that cannot satisfy PMSP constraints (Equation 2-(4)), resulting in infeasible solutions.

Because of this reason, we design a diversity-based strategy in GetPromisingTopK(), by not only picking partial pipelines that maximize the objective function in PMSP (Equation (1)), but also the ones that satisfy the most number of FD/key/column-mapping constraints in (Equation (2)-(4)). Specifically, given a budget of K promising partial pipelines that we can keep in S_{depth}, we consider a balanced set of criteria by selecting $\frac{K}{3}$ pipelines from each of the three groups below:

(1) We select $\frac{K}{3}$ pipelines that have the highest overall probabilities $P(\widehat{L})$;

(2) We select $\frac{K}{3}$ pipelines whose output tables satisfy the most number of FD/Key constraints in the target table;

(3) We select $\frac{K}{3}$ pipelines whose output tables can "map" the most number of columns in the target table.

We demonstrate this using an example below.

Example 6. We continue with the Example 3 in Figure 3. Suppose we have a budget of $K = 3$ pipelines to keep. Using the diversity-based search in GetPromisingTopK(), we can keep 1 pipeline each based on (1) probabilities, (2) key/FD constraints, and (3) column-mapping, respectively. For all 1-step pipelines considered in Example 3, based on the criterion (1) we can see that the partial pipeline $L'_1 = \{GroupBy(\texttt{Fare-Class})\}$ has the highest probability and will be selected, while based on the criterion (2) the pipeline $L'_2 = \{GroupBy(\texttt{Gender})\}$ will be selected as it satisfies an additional FD constraint found in the target table, etc.

Suppose that among all 1-step pipelines, we select the set $S_1 = \{L'_1, L'_2, L'_5\}$ as promising partial pipelines in GetPromisingTopK() given a $K = 3$. In the next iteration when we consider 2-step pipelines, we will start from S_1 and consider different ways to extend pipelines in S_1 using AddOneStep(). We can see that extending $L'_2 \in S_1$ with a Join on `Gender` yields a high probability pipeline satisfying all constraints, which becomes a solution to PMPS.

Note that in this example, we prioritize our search on a promising set of $K = 3$ pipelines at each depth-level, without exploring all possible 1-step and 2-step pipelines.

Additional details. While Algorithm 1 outlines key steps in our synthesis algorithm, there are a few additional optimizations, such as normalizing non-relational input tables, and fine-tuning candidate pipelines in final steps, etc. In the interest of space we refer readers to a full version of this paper [1] for additional details.

4 LEARNING-BASED AUTO-PIPELINE

In addition to the search-based synthesis, we also design a learning-based synthesis, which follows the exact same steps in Algorithm 1, except that we replace the search-based heuristics in GetPromisingTopK and GetFinalTopK (in Section 3.4), using deep reinforcement-learning (DRL) models.

Learning-to-synthesize: key intuition. At a high-level, our pipeline synthesis problem bears strong resemblance to game-playing systems like AlphaGo [47] and Atari [40].

Recall that in learning-to-play games like Go and Atari, agents need to take into account game "states" they are in (e.g., visual representations of game screens in Atari games, or board states in the Go game), in order to produce suitable "actions" (e.g., pressing up/down/left/right/fire buttons in Atari, or placing a stone on the board in Go) that are estimated to have the highest "value" (producing the highest likelihood of winning).

In the case of pipeline synthesis, our problem has a very similar structure. Specifically, like illustrated in Figure 7, at a given "state" in our search graph (representing a partial pipeline L), we need to decide suitable next "actions" to take – i.e., among all possible ways to extend L using different operators/parameters, which ones have the highest estimated "value" (giving us the best chance to synthesize successfully).

Just like game-playing agents can be trained via "self-play" [40, 47], or by playing many episodes of games with win/loss outcomes to learn optimized "policies" for games (what actions to take in which states), we hypothesize that for pipeline-synthesis an optimized synthesis "policy" may also be learned via "self-synthesis" – namely, we could feed an RL agent with large numbers of real data pipelines, asking the agent to synthesize pipelines by itself and assigning rewards when it succeeds.

Deep Q-Network (DQN). Given this intuition, we set out to replace the search-based heuristics in GetPromisingTopK and GetFinalTopK, using a particular form of reinforcement learning called Deep Q-Network (DQN) [40], which uses a deep neural network to directly estimate the "value" of a "state", or intuitively how "promising" a partial pipeline is to ultimately produce a successful synthesis.

More formally, like in Markov Decision Process (MDP), we have a space of *states* $\mathbf{S}$ where each state $s \in \mathbf{S}$ corresponds to a pipeline $L(s) = \{O_1(p_1), O_2(p_2), \ldots, O_s(p_s)\}$, which in turn corresponds to a node in our search graph in Figure 3.

From each state $s \in \mathbf{S}$, we can take an *action* $a \in \mathbf{A}$, which adds a parameterized operator to the pipeline $L(s)$ and leads to a new state s' corresponding to the pipeline $L(s') = \{O_1(p_1), O_2(p_2), \ldots, O_s(p_s), O_a(p_a)\}$. Unlikely MDP, state transition in our problem is deterministic, because adding $O_a(p_a)$ to pipeline $L(s)$ uniquely determines a new pipeline.

The challenge in our pipeline-synthesis problem, however, is that the state/action space of one data-pipeline will be different from another data-pipeline – for example, the action of adding an operator "Join(`Gender`)" in the pipeline of Figure 3 would not apply to other pipelines operating on different input tables. This calls for a way to better "represent" the states and actions, so that learned synthesis policies can generalize across pipelines.

Because of this reason, we choose to use Deep Q-Network (DQN) to directly learn the *value-function* [49] of each state s, denoted as $Q(s)$, which estimates the "value" of a state s, or how "promising" s is in terms of its chance of reaching the desired target.

State representation and model architecture. In order to represent states $\mathbf{S}$ of different pipelines in a manner that generalizes across different pipelines and tables, we need a representation that abstracts away the specifics of each pipeline (e.g., which table column is used), and instead encodes generic information important to by-target synthesis that is applicable to all pipelines.

Recall that in our problem formulation PMPS, our end-goal is to synthesize a pipeline $\widehat{L}$ that can produce all FD/Key constraints discovered from the target T^{tgt}, while the operators invoked in $\widehat{L}$ are plausible with high estimated probabilities. As such, in the representation we design, we directly model these signals, which are data/pipeline independent.

Given a pipeline L_T with T pipeline steps/operators, Figure 8 shows the representation we use to encode the state of L_T, including FD/Key/operators/column-mapping, etc. We will start with the representation for FD, which is encoded using the matrix at lower-left corner (other types of information are encoded similarly and will be described later).

Recall that we discover FDs from the target table T^{tgt}, and our goal is to synthesize pipelines matching all these FDs. We arrange these FDs in T^{tgt} as rows in the matrix, and encode the FDs satisfied by L_T using the right-most columns (marked with T), where a "`0`" entry indicates that this corresponding FD has not been satisfied by L_T yet, while a "`1`" indicates that FD has been satisfied. Columns to the left correspond to FDs of pipelines from previous time-steps, with $T-1$, $T-2$ steps/operators, etc., up to a fixed number of previous steps.

This representation, together with a convolutional architecture, has two benefits. First, this explicitly models historical information (i.e., what FDs are satisfied from previous pipelines), so that with convolution filters, we can directly "learn" whether adding an operator at T-th step makes "progress" in satisfying more FDs. As a concrete example, Figure 8 shows a convolutional filter that may be learned, which have an orange "-1" and a green "+1". This filter would allow us to check whether an increased number of FDs are satisfied from time $(T-1)$ to T. In this example, `FD-2` is not satisfied at the $(T-1)$ step but is now satisfied at the T step, as marked by an orange "`0`" and green "`1`" in the matrix. Applying a dot-product between the matrix and the example conv-filter, will yield (0*(-1) + 1*(+1)) = 1 on this portion of data, which can be intuitively seen as signaling "positive-progress" in covering more FDs from time $(T-1)$ to T. Observe that in comparison, from time $(T-1)$ to T, both `FD-1` (being 0/0 before/after) and `FD-3` (being 1/1 before/after) will get a 0 from applying a dot-product with this filter, indicating no progress for these two FDs.

In typical computer-vision tasks where convolutional architectures are applied, many conv-filters are stacked together to learn visual features (e.g., circles vs. lines). We apply similar conv-filters in our synthesis problem, which interestingly learn to observe local features like whether a pipeline-step is making progress in FD/key/mapping, etc.

A benefit of using this representation with a convolutional architecture is its ability to represent pipelines with varying numbers of FDs/Keys, etc., because we can set the number of rows in the matrix as the maximum number of FDs across all pipelines, and conveniently "pad" rows not used for a specific pipeline as "0"s (which will always result in 0 irrespective of the conv-filters used).

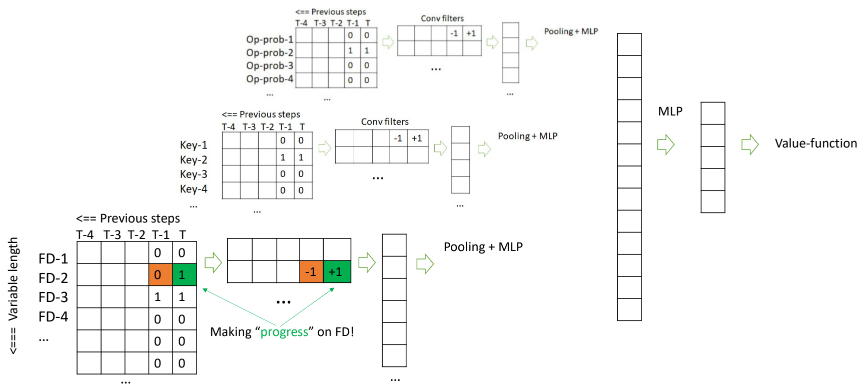


Figure 8: State representation for a partial pipeline at time-step T, using a convolution-based model.

In addition to FD, other types of information (e.g., Key constraints, operator probabilities, column-mapping) can be modeled similarly using the same matrix-representation and conv-filter architecture, as shown in the top part of Figure 8. These representations are then fed into pooling and MLP layers, before being concatenated and passed into additional layers to produce a final function-approximation of $Q(s)$.

Training via prioritized experience-replay. We now describe our approach to train this model to learn the value-function $Q(s)$, using "self-synthesis" of real data pipelines harvested from GitHub.

Similar to [52], we crawl large numbers of Jupyter notebooks on GitHub, and "replay" them programmatically to re-create real data pipelines, denoted by **L**. We then train a reinforcement-learning agent to learn-to-synthesize **L**, by using Algorithm 1 but replacing GetPromisingTopK and GetFinalTopK with learned $Q(s)$ (i.e., picking top-K pipelines with the highest $Q(s)$).

We start with a $Q(s)$ model initialized using random weights. In each subsequent episode, we sample a real pipeline $L = \{O_1(p_1), O_2(p_2), \ldots, O_n(p_n)\} \in \mathbf{L}$ and try to synthesize L using the current $Q(s)$ and Algorithm 1. If we successfully synthesize this L, we assign a reward of +1 for all previous states traversed by L in the search graph – that is, for all $i \in [n]$, we assign $Q(s_i) = +1$ where $s_i = \{O_1(p_1), \ldots, O_i(p_i)\}$[6]. For all remaining states s' traversed that do not lead to a successful synthesis, we assign $Q(s') = -1$. By training the value-function $Q(s)$ using immediate feedback, our hope is that an optimized synthesis policy can be learned quickly that can take into account diverse factors (operator probabilities and various constraints).

We use *prioritized experience replay* [46],in which we record all $(s, Q(s))$ pairs in an internal memory M and "replay" events sampled from M to update the model. (This is shown to be advantageous because of its data efficiency, and the fact that events sampled over many episodes have weak temporal correlations [39]). We train $Q(s)$ in an iterative manner in experience replay. In each iteration, we use the $Q(s)$ from the previous iteration to play self-synthesis and collect a fixed n number of $(s, Q(s))$ events into the memory M. We use [46] to sample events in M to update weights of $Q(s)$, and the new $Q'(s)$ will then be used to play the next round of self-synthesis.

In our experiments, we use $n = 500$, and find the model to converge quickly with 20 iterations. We also observe a clear benefit of using RL over standard supervised-learning (SL), because in RL we get to learn from immediate positive/negative feedback tailored to the current policy, which tends to be more informative than SL data collected over fixed distributions. [7]

[6]This corresponds to not discounting rewards for previous steps, which is reasonable since we typically have only around 10 steps.

5 EXPERIMENTS

We evaluate different pipeline synthesis algorithms by both success rates and efficiency. All experiments were performed on a Linux VM from a commercial cloud, with 16 virtual CPU and 64 GB memory. Variants of Auto-Pipeline are implemented in Python 3.6.9.

5.1 Evaluation Datasets

We created two benchmarks of data pipelines to evaluate the task of pipeline synthesis, which have been made publicly available[8] to facilitate future research.

The GitHub Benchmark. Our first benchmark, referred to as GitHub, consists of real data pipelines authored by developers and data scientists, which we harvested at scale from GitHub. Specifically, we crawled Jupyter notebooks from GitHub, and replayed them programmatically on corresponding data sets (from GitHub, Kaggle, and other sources) to reconstruct the pipelines authored by experts, in a manner similar to [52]. We filter out pipelines that are likely duplicates (e.g., copied/forked from other pipelines), and ones that are trivially small (e.g., input tables have less than 10 rows). These human-authored pipelines become our ground-truth for by-target synthesis.

We group these pipelines based on *pipeline-lengths*, defined as the number of steps in a pipeline. Longer pipelines are intuitively more difficult to synthesize, because the space of possible pipelines grow exponentially with the pipeline-length. For our synthesis benchmark, we randomly sample 100 pipelines of length {1, 2, 3, 4, 5, [6-8], 9+}, for a total of 700 pipelines.

The Commercial Benchmark. Since there are many commercial systems that also help users build complex data pipelines (e.g., vendors discussed in Section 1), we create a second benchmark referred to as Commercial, using pipelines from commercial vendors. We sample 4 leading vendors [9], and manually collect 16 demo pipelines from official tutorials of these vendors, as ground-truth pipelines for synthesis.

Recall that these commercial tools help users build pipelines step-by-step (via drag-and-drop) – with this benchmark we aim to

[7]Additional details of our model can be found in a full paper [1].

[8]Our benchmark data is publicly available at: https://gitlab.com/jwjwyoung/autopipeline-benchmarks

[9]Alteryx [8], SQL Server Integration Services [10], Microsoft Power Query [3], Microsoft Azure Data Factory [2]

Table 1: Characteristics of pipeline synthesis benchmarks.

Benchmark	# of pipelines	avg. # of input files	avg. # of input cols	avg. # of input rows
GitHub	700	6.6	9.1	4274
Commercial	16	3.75	8.7	988

understand what fraction of pipelines from standard commercial use cases (ETL and data-prep) can be automated using Auto-Pipeline.

Note that for learning-based Auto-Pipeline, we use models trained on the GitHub pipelines to synthesize pipelines from the Commercial benchmark, which tests its generalizability.

5.2 Methods Compared

Because "by-target" is a new paradigm not studied in the literature before, we compare Auto-Pipeline with methods mostly from the "by-example" literature.

• **SQL-by-example** [51]. This recent "by-example" approach synthesizes SQL queries by input/output tables. Like other "by-example" approaches, SQL-by-example requires users to provide an *exact* output-table matching the given input-tables. In order to make it work, we provide the exact output of the ground-truth pipelines to SQL-by-example. We use the author's implementation [9], and set a timeout of 3600 seconds per pipeline. For cases where this method fails due to timeout we give it another try using small input tables with 5 sampled rows only.

• **SQL-by-example-UB** [51]. Because SQL-by-example frequently times-out on large input tables, we analyze its theoretical upper-bound of "coverage", based on the operators it supports in its DSL (Join, Aggregation, Union, etc.). If all operators used in a benchmark pipeline are included in its DSL, we mark the pipeline as "covered" in this theoretical upper-bound analysis.

• **Query-by-output-UB (QBO-UB)** [50]. Query-by-output is another influential "by-example" approach that synthesizes SQL by input/output tables. Since its code is not available, we also evaluate its theoretical upper-bound coverage based on its operators.

• **Auto-Pandas** [19]. Auto-Pandas is another by-example approach that synthesizes Pandas programs instead of SQL. We use the authors' implementation [11], and like in SQL-by-Example we feed it with ground-truth output-tables matching the given input tables so that it can function properly.

• **Data-Context-UB** [37]. This recent work proposes to leverage data context (including data values and schema) to automate mapping. Like Query-by-output-UB we evaluate its theoretical upper-bound coverage based on the operators it handles.

• **Auto-Suggest** [52]. Auto-Suggest is a recent approach that automates *single* table-manipulation steps (e.g., Join, Pivot, GroupBy) by learning from Jupyter Notebooks on GitHub. We use Auto-Suggest to synthesize multi-step pipelines, by greedily finding top-K most likely operators at each step.

• **Auto-Pipeline**. This is our proposed method. We report results from three variants, namely the search-based Auto-Pipeline-Search, the supervised-learning-based Auto-Pipeline-SL, and the reinforcement-learning-based Auto-Pipeline-RL.

For learning-based methods, we randomly sample 1000 pipelines with at least 2 steps as training data that are completely disjoint with the test pipelines – specifically, we not only make sure that the train/test pipelines have no overlap, but also the data files used by the train/test pipelines have no overlap (e.g., if a pipeline using "titanic.csv" as input is selected in the test set, no pipelines using an input-file with the same schema would be selected into training). This ensures that test pipelines are completely unseen to learning-based synthesizers, which can better test synthesis on new pipelines. Because learning-based variants uses stochastic gradient descent during training, we report numbers averaged over 5 offline training runs with different seeds.[10]

5.3 Evaluation Method and Metric

Accuracy. Given a benchmark of P pipelines, accuracy measures the fraction of pipelines that can be successfully synthesized, or $\frac{\text{num-succ-synthesized}}{P}$.

Mean Reciprocal Rank (MRR). MRR is a standard metric that measures the quality of ranking [21]. In our setting, a synthesis algorithm returns a ranked list of K candidate pipelines for each test case, ideally with the correct pipeline ranked high (at top-1). The *reciprocal-rank* [21] in this case is defined as $\frac{1}{rank}$, where $rank$ is the rank-position of the first correct pipeline in the candidates (if no correct pipeline is found then the reciprocal-rank is 0). For a benchmark with P test pipelines, the *Mean Reciprocal Rank* is the mean reciprocal rank over all pipelines, defined as:

$$\text{MRR} = \frac{1}{P}\sum_{i=1}^{P}\frac{1}{rank_i}$$

We note that MRR is in the range of [0, 1], with 1 being perfect (all desired pipelines ranked at top-1).

5.4 Comparison of Synthesis

Overall comparisons. Table 2 and Table 3 show an overall comparison on the GitHub and Commercial benchmark, respectively, measured using accuracy, MRR, and latency. We report average latency on successfully-synthesized cases only, because some baselines would fail to synthesize after hours of search.

As can be seen from the tables, Auto-Pipeline based methods can consistently synthesize 60-70% of pipelines within 10-20 seconds across the two benchmarks, which is substantially more efficient and effective than other methods.

While our search-based Auto-Pipeline-Search is already effective, Auto-Pipeline-RL is slightly better in terms of accuracy. The advantage of Auto-Pipeline-RL over Auto-Pipeline-Search is more pronounced in terms of MRR, which is expected as learning-based methods are better at understanding the nuance in fine-grained ranking decisions than a coarse-grained optimization objective in our search-based variant (Equation (1)).

We note that because our input/output tables are from real pipeline and are typically large (as shown in Table 1), existing by-example synthesis methods like SQL-by-Example frequently timeout after hours of search, because their search methods are exhaustive in nature. We should also note that even the theoretical upper-bound coverage of existing by-example methods (based on their DSL) are substantially smaller than Auto-Pipeline, showing the richness of the operators supported in Auto-Pipeline.

[10]It should be noted that once a model is trained, its predictions are deterministic.

Table 2: Results on the GitHub benchmark

	Accuracy	MRR	Latency (seconds)
Auto-Pipeline-Search	76.6%	0.724	18
Auto-Pipeline-SL	73.7%	0.583	20
Auto-Pipeline-RL	**76.9%**	**0.738**	21
SQL-by-Example	14.7%	0.147	49
SQL-by-Example-UB	56%	0.56	-
Query-by-Output-UB	15.7%	0.157	-
Auto-Suggest	29.7%	0.297	11
Data-Context-UB	43%	0.43	-
AutoPandas	9 %	0.09	600

Table 3: Results on the Commercial benchmark

	Accuracy	MRR	Latency (seconds)
Auto-Pipeline-Search	62.5%	0.593	13
Auto-Pipeline-SL	**68.8%**	0.583	14
Auto-Pipeline-RL	**68.8%**	**0.645**	14
SQL-by-Example	19%	0.15	64
SQL-by-Example-UB	37.5%	0.375	-
Query-by-Output-UB	18.8%	0.188	-
Auto-Suggest	25%	0.25	13
Data-Context-UB	25%	0.25	-
AutoPandas	25%	0.25	34.5

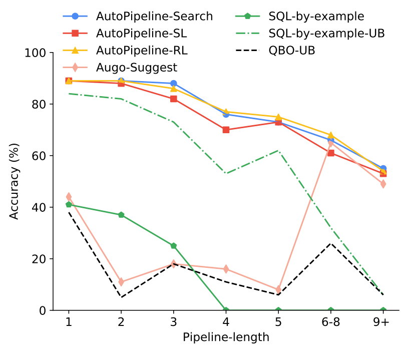


Figure 9: Accuracy results by pipeline-lengths on the GitHub benchmark.

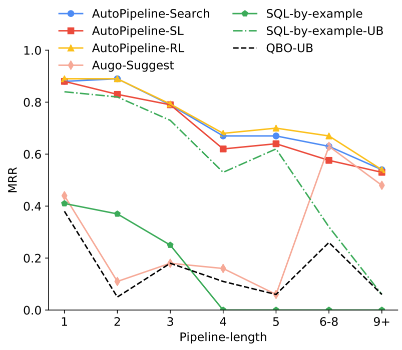


Figure 10: MRR results by pipeline-lengths on the GitHub benchmark.

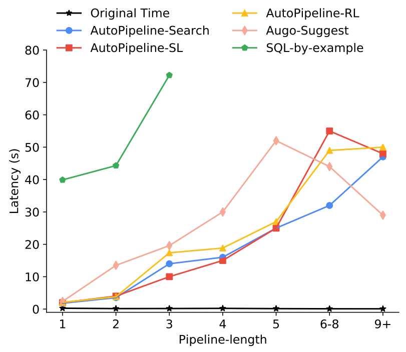


Figure 11: Latency results by pipeline-lengths on the GitHub benchmark

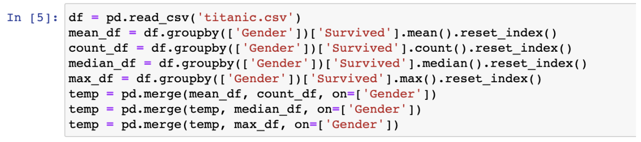
Synthesis Quality. Figure 9 and Figure 10 show detailed comparisons of accuracy and MRR, between different methods on the GitHub benchmark. Test pipelines are bucketized into 7 groups based on their lengths (shown on the x-axis), with longer pipelines being more difficult to synthesize. It can be seen that Auto-Pipeline-RL and Auto-Pipeline-Search are comparable in terms of quality, with Auto-Pipeline-RL being slightly better in terms of MRR. We can see that Auto-Pipeline-RL is noticeably better than Auto-Pipeline-SL, showing the benefit of using RL to proactively select examples to learn from.

All Auto-Pipeline variants are substantially better than QBO and SQL-by-example baselines. We note that SQL-by-example fails to synthesize any pipeline longer than 3-steps within our 1 hour timeout limit, showing its limited efficacy when dealing with large input tables from real pipelines.

Latency. Figure 11 shows a comparison of the average latency to successfully synthesize a pipeline between all methods on the GitHub benchmark. While all Auto-Pipeline methods have comparable latency, SQL-by-example requires 20x-7x more time to synthesize pipelines up to 3 steps.

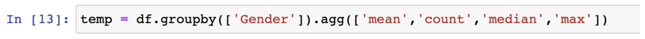
Pipeline simplification. We observe in experiments that our synthesized pipelines can sometimes be simpler (with fewer steps) than human-authored ground-truth pipelines, while still being semantically equivalent. Figure 12 shows a real example from GitHub, where the human-authored pipeline would group-by on column `Gender` for four times with different aggregation, before joining them back. A synthesized pipeline from Auto-Pipeline is a one-liner in this case and more succinct. While this example is intuitive, there are many more involved examples of simplifications – for example, having an Unpivot on each of K similar files followed by (K-1) union, is equivalent to a Join between the K similar files followed by one Unpivot, etc.

Out of the 700 pipeline in the GitHub benchmark, our synthesized pipelines are simpler on 90 cases (12.85%), which we believe is an interesting use and an added benefit of Auto-Pipeline.



```
In [5]: df = pd.read_csv('titanic.csv')
        mean_df = df.groupby(['Gender'])['Survived'].mean().reset_index()
        count_df = df.groupby(['Gender'])['Survived'].count().reset_index()
        median_df = df.groupby(['Gender'])['Survived'].median().reset_index()
        max_df = df.groupby(['Gender'])['Survived'].max().reset_index()
        temp = pd.merge(mean_df, count_df, on=['Gender'])
        temp = pd.merge(temp, median_df, on=['Gender'])
        temp = pd.merge(temp, max_df, on=['Gender'])
```

(a) A human-authored pipeline



```
In [13]: temp = df.groupby(['Gender']).agg(['mean','count','median','max'])
```

(b) A synthesized pipeline with fewer steps

Figure 12: An example pipeline simplified after synthesis.

5.5 Robustness Analysis

To understand the robustness of our algorithm in the presence of noisy input/output tables, we perform various robustness tests.

Add irrelevant input tables. For each pipeline synthesis task, we inject K extra input tables randomly sampled from other test pipelines irrelevant to this synthesis task, which is used to test the robustness of our algorithm in the presence of irrelevant data sources. Figure 13 shows that even with 3 extra irrelevant input

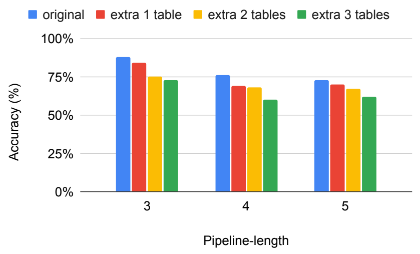


Figure 13: Robustness: add extra input tables irrelevant to pipelines.

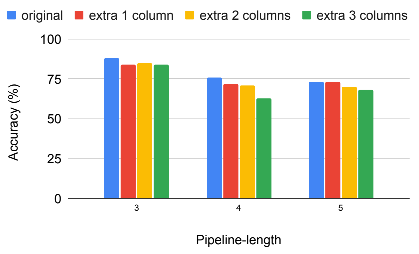


Figure 14: Robustness: add extra columns irrelevant to pipelines.

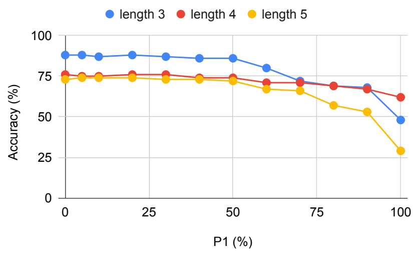


Figure 15: Robustness: randomly perturb column values.

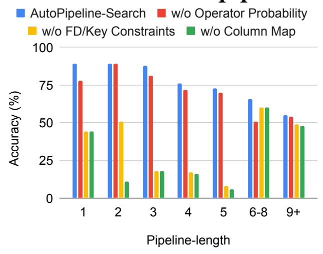


Figure 16: Ablation study.

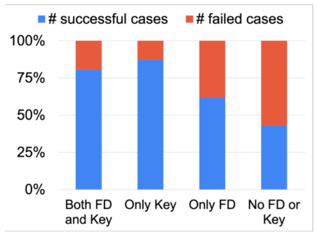
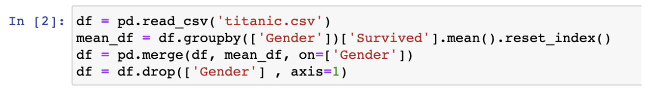


Figure 17: Error Analysis.



```
In [2]: df = pd.read_csv('titanic.csv')
        mean_df = df.groupby(['Gender'])['Survived'].mean().reset_index()
        df = pd.merge(df, mean_df, on=['Gender'])
        df = df.drop(['Gender'] , axis=1)
```

Figure 18: An example pipeline that we fail to synthesize.

tables, our synthesis algorithm (not specifically optimized to handle noisy and irrelevant tables) only has a slight decrease in accuracy.

Add irrelevant columns to input tables. In addition to injecting extra irrelevant tables, for each test pipeline, we also test the scenario in which we inject K extra columns to each input table randomly sampled from other irrelevant pipelines. Figure 14 shows that our accuracy also drops slightly in such settings.

Perturb column values. In order to test the robustness of our synthesis in the presence of noisy data values (e.g., typos and name variations), for each input table in a pipeline, we randomly select 50% of string columns and randomly perturb p fraction of their values. Specifically, we randomly initialize a character scrambling scheme (e.g., 'a' $\longrightarrow$ 't', 'b' $\longrightarrow$ 'f', etc.), and for p fraction of distinct values in a selected column, we apply the scrambling character-by-character to perturb values (e.g., 'abc store' $\longrightarrow$ 'tfg kabzo'). Such perturbations are performed on each input table independently. Figure 15 shows the synthesis accuracy when varying p_1 from 0 to 100%. It can be seen that our synthesis is still robust even when 50% of values are perturbed.

5.6 Error Analysis

In this section, we analyze 70 sampled failed cases (10 cases for each group of pipeline lengths), to understand why AUTO-PIPELINE fails to synthesize. We categorize the failed reasons as follows:

- Incorrect singe operator parameter. 41% of failed cases fall in this category (e.g., a ground-truth join column is not in the top-K parameters predicted for the given pair of tables).
- Incorrect string transformation. 26% failed cases are in this category, which can be attributed to the fact that the synthesis of string transformations in our case do not have paired input/output examples (unlike the traditional by-example setting).
- False-positive FDs. There are two cases where false-positive FDs are discovered from target tables, which prevents synthesized pipelines to produce matching FDs, causing the synthesis to fail.
- Deleted key column. In three test cases, key columns are deleted in final steps of the pipelines, leading to missing constraints.

Spurious discovery of constraints. While it is known that spurious FDs/keys can be discovered [43], especially on small input tables, they do not contribute significantly to failed synthesis in our evaluation. We believe this is because unlike by-example synthesis that uses only a few rows, our test pipelines operate on real data tables (e.g., from Kaggle) that are typically large (e.g., on average an input table used in GitHub pipelines have over 4K rows, and an output table has over 41K rows). Such large tables make false discovery of spurious constraints substantially less likely. To confirm this, we down-sample target tables of each test case to 20 rows each and re-run AUTO-PIPELINE. We observe 37 out of 70 cases would then fail due to spurious FDs/Keys, confirming the hypothesis that large realistic tables indeed prevents spurious discovery of constraints.

Contribution of constraints in synthesis. Figure 17 shows how fail-rates vary when FDs/Keys do not exist in the target table T^{tgt}. We can see from the two rightmost bars that when only FDs exist T^{tgt} (key columns are missing), or when both FDs/Keys are missing, fail-rates go up significantly.

Additional results. We present additional results such as sensitivity analyses of in a full version of the paper [1].

6 RELATED WORKS

We briefly review related work on automating pipeline-building in this section in the interest of space. We give additional discussions of the broad area of data preparation in a full version of the paper [1].

Automate data transformations. Data transformation is a long-standing problem and a common step in data-pipelines. Significant progress has been made in this area, with recent work for "by-pattern" [36] and "by-example" approaches [17, 28, 30, 31, 33–35, 48]. These techniques have generated substantial impacts on commercial systems, with related features shipping in popular systems such as Microsoft Excel [28], Power BI [30], and Trifacta [7].

Multi-step pipeline synthesis. As discussed, existing work on multi-step pipelines mostly focus on the by-example paradigm (e.g., SQL-by-example [51], Query-by-output [50]), where an exact output-table is often difficult for users to provide. In addition, most approaches support a limited set of operators, which limits their ability to synthesize complex pipelines.

7 CONCLUSIONS AND FUTURE WORKS

In this paper, we propose a new by-target synthesis paradigm to automate pipeline-building. We design search and learning-based synthesis algorithms, which are shown to be effective on real data pipelines. Future directions include extending our current DSL, and incorporating user feedback to facilitate synthesis.

REFERENCES

[1] [n.d.]. Full version of Auto-Pipeline: Synthesize Data Pipelines By-Target Using Search and Reinforcement Learning. https://arxiv.org/abs/2106.13861.

[2] [n.d.]. Microsoft Azure Data Factory. https://azure.microsoft.com/en-us/services/data-factory/.

[3] [n.d.]. Microsoft Excel Power Query. http://office.microsoft.com/powerbi.

[4] [n.d.]. Titanic challenge on Kaggle. https://www.kaggle.com/c/titanic.

[5] [n.d.]. Titanic Challenge on Kaggle. https://www.kaggle.com/c/titanic.

[6] [n.d.]. Transform-by-Example feature in Power Query. http://powerbi.microsoft.com/en-us/blog/power-bi-desktop-june-feature-summary/#addColumn.

[7] [n.d.]. Transform-by-Example feature in Trifacta. https://www.trifacta.com/blog/transform-by-example-your-data-cleaning-wish-is-our-command.

[8] 2016.10.21. Alteryx. https://www.alteryx.com/.

[9] 2016.10.21. Implementation for SQL-by-example. https://github.com/Mestway/Scythe.

[10] 2016.10.21. SQL Server Integration Service. https://docs.microsoft.com/en-us/sql/integration-services/sql-server-integration-services.

[11] 2017.04.26. Auto-Pandas code. https://github.com/rbavishi/atlas/blob/oopsla19-snapshot/autopandas_v2/evaluation/benchmarks/stackoverflow.py.

[12] 2017.04.26. FlashFill in Excel. https://www.microsoft.com/en-us/microsoft-365/blog/2012/08/09/flash-fill/.

[13] 2017.04.26. Jupter Notebooks. https://jupyter.org/.

[14] 2017.04.26. Python Pandas Library. https://pandas.pydata.org/.

[15] 2017.04.26. Recommended Join in Tableau. https://help.tableau.com/current/prep/en-us/prep_combine.htm.

[16] 2017.04.26. Recommended Join in Trifacta. https://www.trifacta.com/blog/making-data-blending-faster-easier/.

[17] Ziawasch Abedjan, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2016. DataXFormer: A robust transformation discovery system. In *ICDE*.

[18] Daniel W Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. *ACM SIGPLAN Notices* 50, 6 (2015), 218–228.

[19] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[20] Matt Buranosky, Elmar Stellnberger, Emily Pfaff, David Diaz-Sanchez, and Cavin Ward-Caviness. 2018. FDTool: a Python application to mine for functional dependencies and candidate keys in tabular data. *F1000Research* 7 (2018).

[21] UP Cambridge. 2009. *Online edition (c) 2009 Cambridge UP An Introduction to Information Retrieval Christopher D.* Manning Prabhakar Raghavan Hinrich Schütze Cambridge University Press ....

[22] Anish Das Sarma, Aditya Parameswaran, Hector Garcia-Molina, and Jennifer Widom. 2010. Synthesizing view definitions from data. In *Proceedings of the 13th International Conference on Database Theory.* 89–103.

[23] Tamraparni Dasu and Theodore Johnson. 2003. *Exploratory Data Mining and Data Cleaning.* John Wiley & Sons, Inc., New York.

[24] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System.. In *Cidr*.

[25] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. 2015. Unsupervised learning by program synthesis. *Advances in neural information processing systems* 28 (2015), 973–981.

[26] Peter D Grünwald and Abhijit Grunwald. 2007. *The minimum description length principle.* MIT press.

[27] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[28] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM Sigplan Notices*, Vol. 46. ACM, 317–330.

[29] William R. Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *Proceedings of SIGPLAN.* 317–328. https://doi.org/10.1145/1993498.1993536

[30] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-data-by-example (TDE): an extensible search engine for data transformations. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1165–1177.

[31] Yeye He, Kris Ganjam, Kukjin Lee, Yue Wang, Vivek Narasayya, Surajit Chaudhuri, Xu Chu, and Yudian Zheng. 2018. Transform-Data-by-Example (TDE) Extensible Data Transformation in Excel. In *Proceedings of the 2018 International Conference on Management of Data.* 1785–1788.

[32] Jeffrey Heer, Joseph M Hellerstein, and Sean Kandel. 2015. Predictive Interaction for Data Transformation.. In *CIDR*.

[33] Jeffrey Heer, Joseph M. Hellerstein, and Sean Kandel. 2015. Predictive Interaction for Data Transformation. In *CIDR*.

[34] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *SIGMOD*.

[35] Zhongjun Jin, Michael Cafarella, HV Jagadish, Sean Kandel, Michael Minar, and Joseph M Hellerstein. 2018. CLX: Towards verifiable PBE data transformation. *arXiv preprint arXiv:1803.00701* (2018).

[36] Zhongjun Jin, Yeye He, and Surajit Chauduri. 2020. Auto-transform: learning-to-transform by patterns. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2368–2381.

[37] Martin Koehler, Edward Abel, Alex Bogatu, Cristina Civili, Lacramioara Mazilu, Nikolaos Konstantinou, Alvaro Fernandes, John Keane, Leonid Libkin, and Norman W Paton. 2019. Incorporating Data Context to Cost-Effectively Automate End-to-End Data Wrangling. *IEEE Computer Architecture Letters* 01 (2019), 1–1.

[38] Oliver Lehmberg, Dominique Ritze, Petar Ristoski, Robert Meusel, Heiko Paulheim, and Christian Bizer. 2015. The mannheim search join engine. *Journal of Web Semantics* 35 (2015), 159–166.

[39] Long-Ji Lin. 1993. *Reinforcement learning for robots using neural networks.* Technical Report. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

[40] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[41] Peng Si Ow and Thomas E Morton. 1988. Filtered beam search in scheduling. *The International Journal Of Production Research* 26, 1 (1988), 35–62.

[42] 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. *Proceedings of the VLDB Endowment* 8, 10 (2015), 1082–1093.

[43] Thorsten Papenbrock and Felix Naumann. 2016. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data.* 821–833.

[44] Erhard Rahm and Philip A Bernstein. 2001. A survey of approaches to automatic schema matching. *the VLDB Journal* 10, 4 (2001), 334–350.

[45] Rita L. Sallam, Paddy Forry, Ehtisham Zaidi, and Shubhangi Vashisth. 2016. Gartner: Market Guide for Self-Service Data Preparation. (2016).

[46] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[47] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.

[48] Rishabh Singh. 2016. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment* 9, 10 (2016), 816–827.

[49] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction.* MIT press.

[50] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data.* 535–548.

[51] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 452–466.

[52] Cong Yan and Yeye He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 1539–1554.

[53] Erkang Zhu, Yeye He, and Surajit Chaudhuri. 2017. Auto-join: Joining tables by leveraging transformations. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1034–1045.