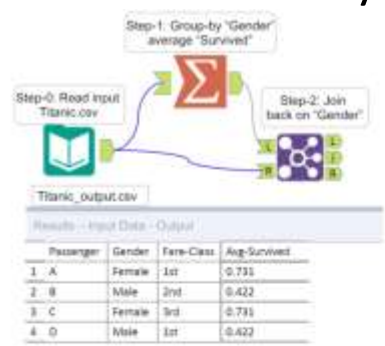
Auto-Pipeline: Synthesizing Complex Data Pipelines By-Target Using Reinforcement Learning and Search

Authors: Junwen Yang (University of Chicago),
Yeye He (Microsoft Research), Surajit Chaudhuri (Microsoft Research)
Published: VLDB, August 2021, Copenhagen, Denmark
Presentation: Georgios Xydias

The Data Preparation Bottleneck

- **Data preparation** involves multiple table-level transformations (e.g., Join, Pivot, GroupBy) to shape raw data for analysis
- Consumes up to 80% of a data scientist's time and poses even greater challenges for non-technical users
- The final outcome of this work is a **multi-step data pipeline**, which must be **reused and deployed** in production
- Traditional tools automate single-step transformations only (e.g., drag-and-drop ETL tools), leaving users to manually build complex workflows

(a) A pipeline authored using Python Pandas



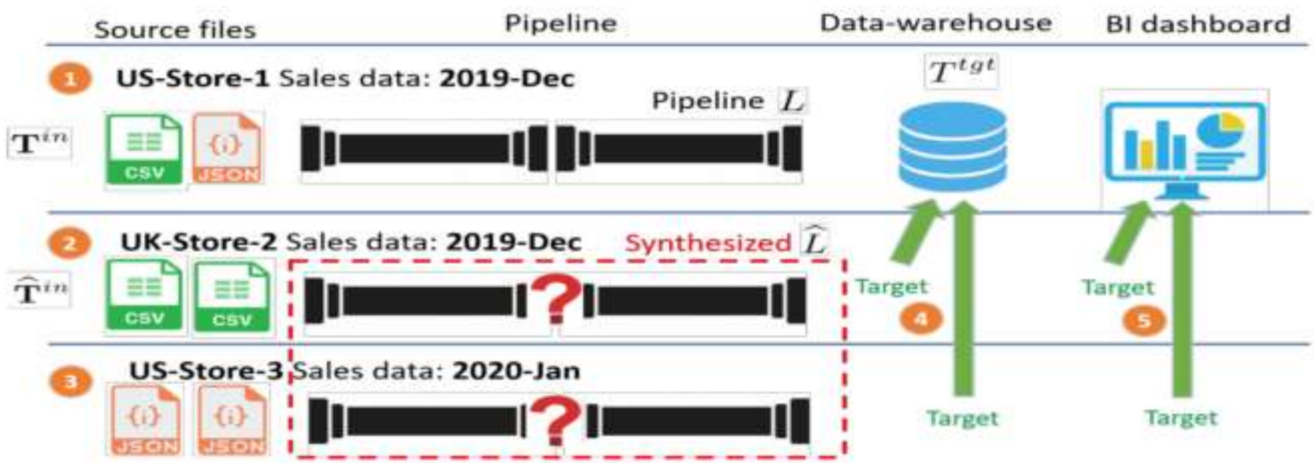
(b) A pipeline authored in visual drag-and-drop tool

- There is a critical need to **automate the entire pipeline construction process** to reduce effort, errors and time

Existing Methods vs. New Paradigm

- **By-Example Synthesis (Traditional)**
  - Requires exact input-output table pairs (e.g., SQL-by-Example)
  - Users must manually construct the entire final output table
  - Quickly becomes tedious and unscalable for large or complex pipelines
- **By-Target Synthesis (Auto-Pipeline)**
  - Users point only a target table or dashboard as a fuzzy reference that illustrates the desired output format
  - No need to construct the full output as the system infers the desired structure, schema and transformation logic from the target
  - Much easier to specify, especially for non-technical users

Example: “By-Target” Synthesis

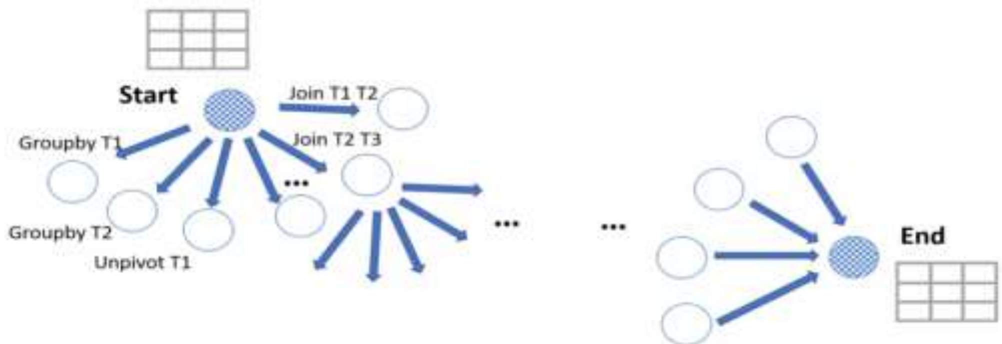


- In real-world settings, new data sources (e.g., stores, time periods) often arrive in different formats or schemas
- Instead of manually building new pipelines for each new data chunk, **Auto-Pipeline** automatically synthesizes pipelines that transform new input to match the target output
- Users can easily trigger synthesis by:
  - Right-clicking an existing table and selecting **“Append data to this table”**, or
  - Pointing to a dashboard and choosing “Create a dashboard like this”

Introducing Auto-Pipeline

- Given a new input and a target table, the system must synthesize a pipeline that transforms the input to match the target
- This task is formulated as a **search problem** over a space of candidate pipelines
- Auto-Pipeline begins with an empty pipeline and builds it step-by-step by adding one operator at each step forming increasingly longer partial pipelines
- To manage the exponential number of candidates, it relies on:
  – **Auto-Suggest** to rank the most likely next steps
    - Predicts the top-K most likely next operations (Join, Pivot, etc.) based on the past operator sequence and the current table's structure
  – **Search and Learning-based strategies** to limit exploration
- The process stops when the top-K synthesized pipelines satisfy all key constraints derived from the target, including functional dependencies, keys, and column mappings

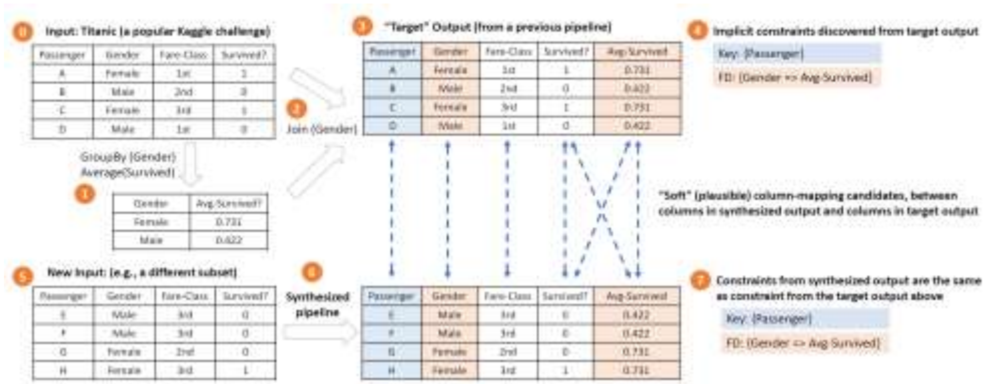
Auto-Pipeline Search Process



- Nodes represent partial pipelines and edges correspond to the application of a single operator
- At each step, the pipeline is expanded by one new operator at a time, forming a longer partial (and candidate) pipeline
- Example: starting from an empty pipeline with 6 operator choices:
  - Step 1: -> 6 candidate pipelines
  - Step 2: -> each expands to 6 more -> 36 total
  - And so on - the number of pipeline paths grows exponentially
- This search space becomes intractable without smart pruning
- Auto-Suggest guides this expansion by ranking the most likely operators, focusing only on the top-M options

Key Insight – Why it works

- The target table acts as a "fuzzy" specification (that's surprisingly sufficient) - not row-complete, but rich in structural constraints
- From the target, we extract **Functional Dependencies (FDs)** and **Key constraints** that must hold in any valid output
- These constraints help **drop invalid pipelines** early as most candidate pipelines break FDs or keys constraints and can be discarded
- **Soft column mappings** (based on names, values, types) also link outputs to the target, enabling semantic alignment even when schemas differ
- Together, FDs, Keys, and column mappings provide enough semantic signal to guide synthesis without needing full output examples – they act as filters, **eliminating invalid pipelines** early in process



Problem Formulation

- **Goal**: Given new input data (T^{in}) and a target table (T^{tgt}), synthesize a pipeline (L) such that $L(T^{\text{in}})$ structurally **matches the target output**
- **Operators** (from a fixed DSL):
  - table-level (Join, GroupBy, Pivot, etc.)
  - string-level (Split, Substring, etc.)
- **Constraints**: The synthesized pipeline must preserve the target's **FDs**, **key constraints**, and **column mappings** to the target table
- **Formal Objective**: Probabilistic Multi-Step Pipeline Synthesis (PMPS)

  (Maximize operator probabilities while satisfying structural constraints)

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

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

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

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

The Algorithm

Algorithm 1 Synthesis: A meta-level synthesis algorithm

procedure Synthesis($\widehat{\mathbf{T}}^{in}, T^{tgt}, \mathbf{O}$)
  $depth \leftarrow 0, candidates \leftarrow \emptyset$
  $S_{depth} \leftarrow \{empty()\}$ ▷ #initialize an empty pipeline
  while $depth < maxDepth$ **do**
    $depth \leftarrow depth + 1$
    for each ($L \in S_{depth-1}, O \in \mathbf{O}$) **do**
      $S_{depth} \leftarrow S_{depth} \cup \text{AddOneStep}(L, O)$
    $S_{depth} \leftarrow \text{GetPromisingTopK}(S_{depth}, T^{tgt})$
    $candidates \leftarrow candidates \cup \text{VerifyCands}(S_{depth}, T^{tgt})$
  return GetFinalTopK($candidates$)

At each step of the synthesis:

- **AddOneStep(L, O)**: Extend each partial pipeline by applying one new operator. Uses Auto-Suggest to predict the top-M most likely parameters, e.g. GroupBy(Gender), GroupBy(Fare-Class)
- **GetPromisingTopK(S, T^{tgt})**: Selects the most promising pipelines using either:
  - a **diversity-based heuristic**, or
  - a **learning-based ranking model**
- **VerifyCands (S , T^{tgt})** : Filters candidates that satisfy all structural constraints from the problem formulation (Equations 2-4: FDs, Keys, Column-Mapping)
- **GetFinalTopK(candidates)**: Selects the top-K pipelines by re-ranking candidates using either a **search-based operator likelihood score** or a **learned Q-value policy** (RL setting)
- The algorithm builds pipelines layer by layer, expanding only promising branches based on operator predictions and constraint satisfaction

Auto-Pipeline-Search

Subroutine: GetPromisingTopK(S, T^{tgt})

- Applied during search (at each intermediate level, e.g. depth-1, depth-2,..) to retain the most promising partial pipelines
- Filters candidates using (VerifyCand):
  - Auto-Suggest scores (likelihood of next operator)
  - FD and Key constraint satisfaction
  - Soft column mappings
- Promotes **diverse and viable paths** for further expansion

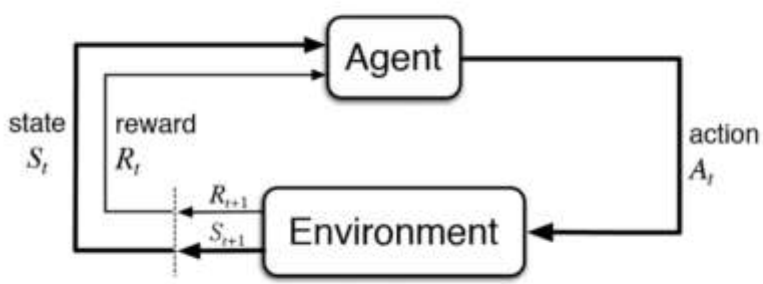
Subroutine: GetFinalTopK(candidates)

- Applied at the end of synthesis process to rank **fully constructed pipelines**
- Computes the **joint likelihood** of each pipeline using the product of per-step operator probabilities
- Returns the top-K candidates with the highest total scores

- **Why both are needed:**
  - GetPromisingTopK enables early filtering and guides exploration on valid and high-potential candidates
  - GetFinalTopK ensures the best complete pipelines are selected using global scores
  - This separation improves efficiency and robustness by combining **early-stage constraint filtering** with **final global ranking**

Auto-Pipeline-RL

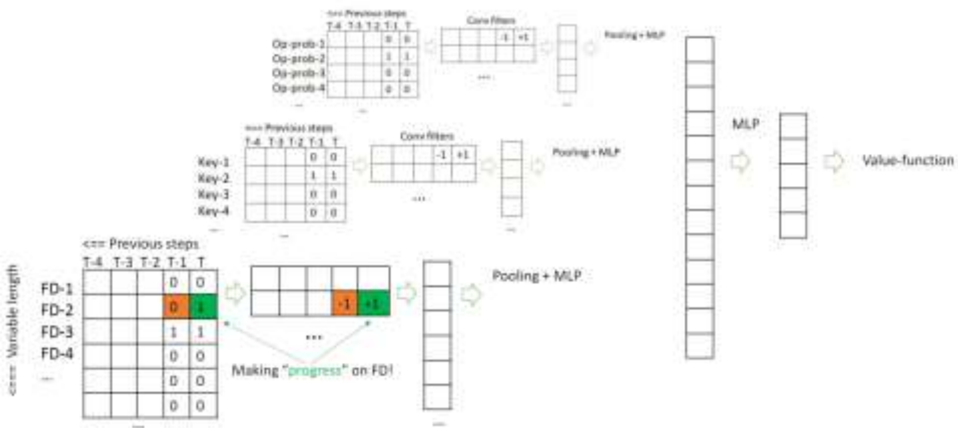
- Replaces the search-based heuristics in GetPromisingTopK and GetFinalTopK with a deep reinforcement learning (RL) model
- Learns to rank pipeline candidates by estimating **Q-values** of partial pipeline states instead of relying on probability-based scoring
- Uses a Q-function, trained via a **Deep Q-Network (DQN)**, to guide the selection of promising actions
- **Reinforcement learning**: An agent interacts with an environment by taking actions to maximize the expected cumulative reward
  - In RL, an agent in a state takes an action, if the result is correct, it's rewarded, otherwise penalized. **Q-value** estimates the **expected reward** of taking an action in a given state
  - The goal is the agent learn a **policy** that selects actions with **high Q-values** in order maximize the long-term reward
  - Similar to how agents learn to play games like AlphaGo or Atari through trial and error



Deep Q-network (DQN)

- Each **node** (partial pipeline) in search graph is a **state** in RL framework
- An **action** is applying an operator to move to a new state
- **Reward**: +1 for successful synthesis, –1 otherwise
- Since each state corresponds to a **different intermediate table**, we must use general, schema-independent features to represent them
- We encode each state using:
  - Functional Dependencies (FDs)
  - Key constraints
  - Column mappings
  - Operator likelihood scores (Auto-Suggest scores)
- The **DQN** learns to predict a **Q-value** for each state, estimating how promising a partial pipeline is for reaching a valid target

RL State Representation

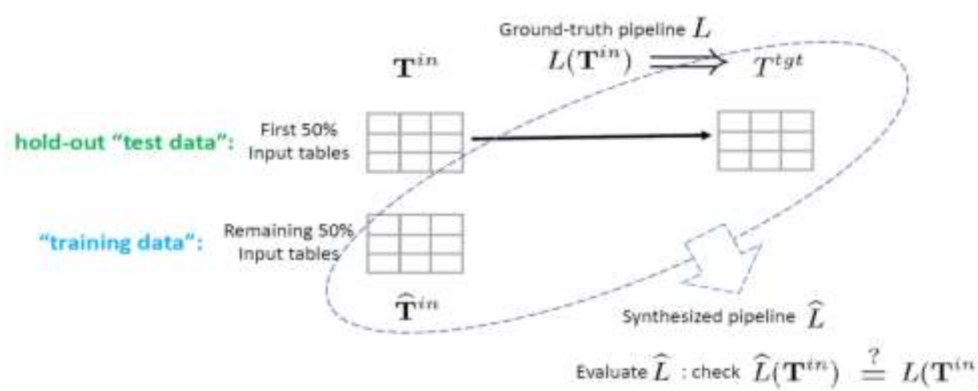


- Each constraint type (FDs, Keys, Column Mappings, operator probs, etc.) is encoded as a matrix that captures progress across the pipeline history
- 1 is assigned if the constraint is satisfied at a given step, 0 otherwise
- A 1D convolutional filter slides over each matrix to detect local progress
- The outputs of all these conv layers are **concatenated** into a single state vector for the RL agent
- Benefits of method:
  - Captures historical progress and the DQN can learn if the pipeline is improving toward satisfying the target
  - Matrices can be padded (if needed) and convolved to a fixed-size vector, so each RL state is encoded as a fixed-size numeric vector, regardless of pipeline length

Evaluation Protocol

- Split each real-world pipeline’s input data 50/50 into training and test
  - First 50%: used to generate the target output
  - Second 50%: used as input for pipeline synthesis
- Use the original pipeline’s output from the first half as the target
  - The system must synthesize a pipeline that transforms the second half to match this target
- **Evaluate correctness** by checking whether the synthesized pipeline **reproduces the original pipeline’s output**
- If the output matches: reward = **+1**; otherwise: reward = **–1**
  - This reward signal is used during training to guide the learning process

➢ This protocol enables training and evaluation without manual labeling



Training the DQN via Self-Synthesis

- RL agent is trained using **self-synthesis episodes**:
  - Try to construct a pipeline from input $T^{\mathrm{i?}}$ that matches the target output T^{tgt}
- Each episode produces a trajectory of transitions (agent steps):
  - A sequence of multiple steps the agent takes while constructing a pipeline
  - Agent step: *(s, a, r, s')* - (state, action, reward, next state)
- No manual labels are needed — success is evaluated by checking if the synthesized output matches the original
- *Q(s, a)* is the **predicted Q-value** of taking action *a* in state *s*, initialized using random network weights
- For each episode:
  - Evaluate episode by assigning reward based on pipeline success, compute **target Q-value** using Bellman equation and **TD error** (target Q-value – predicted Q-value)
- Sample 500 episodes using **prioritized experience replay**:
  - Transitions with larger TD error are sampled more often (e.g. failed steps)
- Update *Q(s, a)* using Bellman equation
- Over time *Q(s, a)* approximates the true long-term reward, enabling the agent to prefer promising pipelines, effectively replacing the 2 subroutines

Evaluation Datasets

- **Github benchmark**: crawled 4M repositories and replayed jupyter notebooks to extract 700 real-world data pipelines
- **Commercial benchmark**: 16 pipelines collected from 4 enterprise leading data platforms
- Both cover diverse pipeline lengths and transformation complexities
- 1000 random pipelines used for learning-based methods, with strict train/test input data table split (no overlap)
- Benchmarks reflect real usage patterns from both open-source notebooks and enterprise workflows

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

Overall Results

Table 2: Results on the GitHub benchmark

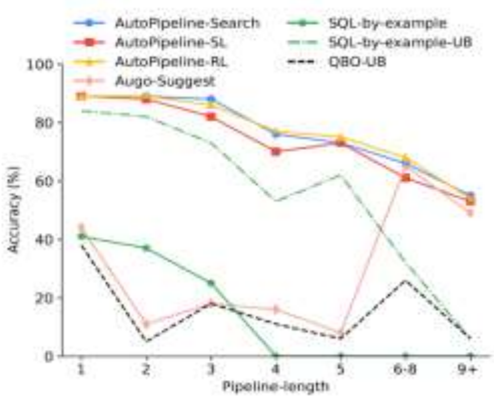
	Accuracy	MRR	Latency (seconds)
Auto-Pipeline-Search	76.6%	0.724	18
Auto-Pipeline-SL	73.7%	0.583	20
Auto-Pipeline-RL	**76.9%**	**0.738**	21
SQL-by-Example	14.7%	0.147	49
SQL-by-Example-UB	56%	0.56	-
Query-by-Output-UB	15.7%	0.157	-
Auto-Suggest	29.7%	0.297	11
Data-Context-UB	43%	0.43	-
AutoPandas	9 %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

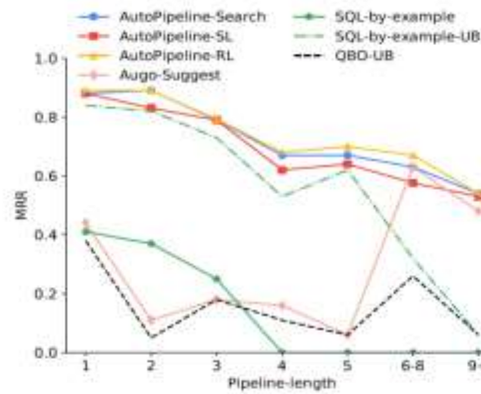
Table 3: Results on the Commercial benchmark

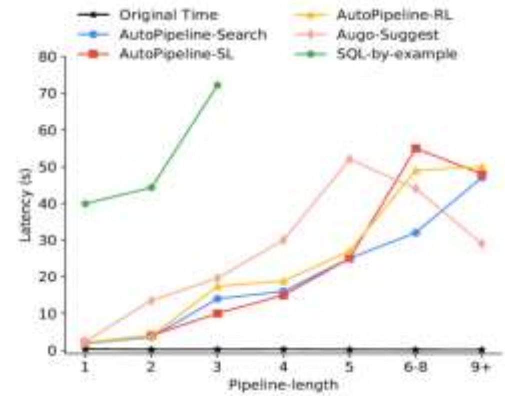
	Accuracy	MRR	Latency (seconds)
Auto-Pipeline-Search	62.5%	0.593	13
Auto-Pipeline-SL	**68.8%**	0.583	14
Auto-Pipeline-RL	**68.8%**	**0.645**	14
SQL-by-Example	19%	0.15	64
SQL-by-Example-UB	37.5%	0.375	-
Query-by-Output-UB	18.8%	0.188	-
Auto-Suggest	25%	0.25	13
Data-Context-UB	25%	0.25	-
AutoPandas	25%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

- Metrics:
  - **Accuracy**: Fraction of pipelines for which the synthesized output exactly matches the ground truth
  - **Mean Reciprocal Rank (MRR)**: Measures how high the correct output ranks among candidate outputs
- Auto-Pipeline models outperform all baselines in both accuracy and MRR
- RL-based model slightly outperforms search-based, especially in MRR
  - **Best overall performance**: Auto-Pipeline-RL achieves highest accuracy and MRR on both benchmarks, while maintaining fast inference speed
- **Learning-based models generalize better** to new data

Robustness & Performance by Pipeline Length







- **Auto-Pipeline maintains strong accuracy even on long pipelines**, while baselines fail on pipelines longer than 3 steps

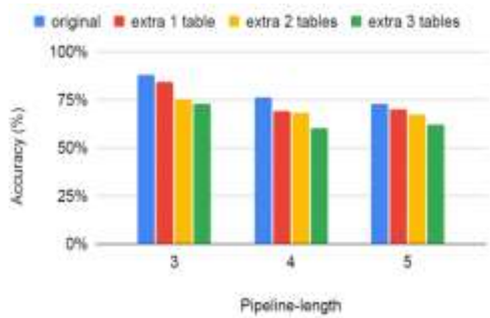


Figure 13: Robustness: add extra input tables irrelevant to pipelines.

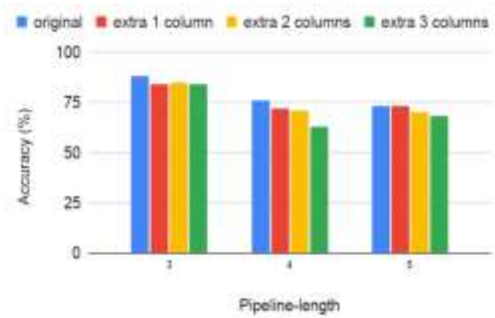


Figure 14: Robustness: add extra columns irrelevant to pipelines.

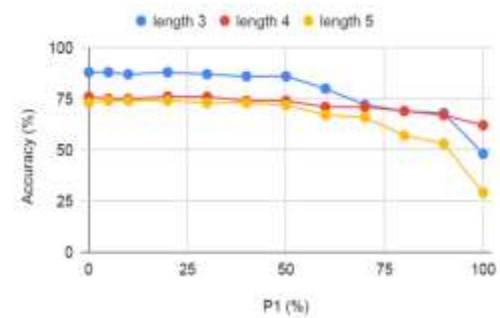


Figure 15: Robustness: randomly perturb column values.

- **Auto-Pipeline demonstrates robustness** to irrelevant tables, extra columns, and noisy data values

Conclusion & Future Work

- Introduced the **first framework for "by-target" pipeline synthesis**
  - Requires only a desired output table, not full input-output examples
- Demonstrated feasibility of **automating multi-step data preparation pipelines** by combining Search-based and Learning-based models
- Showed that **learning-based models (Auto-Pipeline-RL)** generalize better across pipeline lengths and noisy inputs
  - **Achieved best performance on both Github and Commercial benchmarks, and demonstrated robustness**
- Opened a new research direction beyond traditional by-example approaches
- **Future directions**:
  - **Extend to richer DSLs** (e.g. full Pandas API coverage),
  - **Incorporate interactive user feedback into synthesis**
- Ultimately, **Auto-Pipeline aims to make powerful data preparation accessible, reliable, and automatic for all users**

Questions?

Thank you