Toward Standardized Data Preparation: A Bottom-Up Approach

Eugenie Lai
MIT
eylai@mit.edu

Yuze Lou
University of Michigan
yuzelou@umich.edu

Brit Youngmann
Technion
brity@technion.ac.il

Michael Cafarella
MIT
michjc@csail.mit.edu

ABSTRACT

Data preparation is an important step in any data-related application, from academic research to industrial decision-making. Typically, data preparation is not a core contribution of a project — it transforms raw data into a format that supports further innovative work. However, the reality is that data scientists spend much of their time on data preparation. Because data preparation scripts are highly project-specific and often written in general-purpose languages, they are tedious to understand and difficult to verify. As a result, data preparation scripts can be a breeding ground for poor engineering and statistical practices, and even when they are perfectly built, they are difficult to reuse. Ideally, data preparation scripts should serve the project, but otherwise, be as simple and as standard as possible — adhering to the best common practices to process data. We propose a bottom-up script standardization framework that takes a user's data preparation script and transforms it into a more standardized version of itself. Our framework treats the user's input script as a semantic sketch of the user intent, which can be modified even if the changed script yields a slightly different output. We present an algorithmic framework and developed a prototype system. We evaluated our approach against state-of-the-art methods, including GPT-4, on six real-world datasets. Our approach improved script standardization by 39.5% while meaningfully preserving the user's intent, while GPT-4 achieved 2.9%.

1 INTRODUCTION

Data preparation is a crucial step in various domains, from academic research to data-driven decision making in industry [63, 64]. Custom data preparation programs are often an essential part of data science pipelines, transforming raw data into a usable format for further innovative work. However, data preparation is often overlooked, and analysts rarely detail the process [15]. These "uninteresting" data preparation scripts serve solely to support downstream innovative efforts.

However, data preparation scripts remain extremely difficult to understand, verify, and reuse. Data preparation scripts are highly project-specific and often written in general-purpose programming languages. These attributes make them tedious to understand and difficult to verify, without substantial effort. Consequently, data preparation scripts can be a breeding ground for poor engineering and statistical practices. For example, *variability* in data preparation was identified as one main reason for the high rate of false positive results in biomedical research [15]. Despite the widespread use of standard datasets, preparation scripts can be constructed to produce desirable experimental results, based on specific data values and characteristics, rendering scientific conclusions not reproducible [28, 59, 62].

```
1. improrт pandas as pd
2. df = read_csv('diabetes.csv')
3. df = df.fillna(df.median())
4. df = df[df['Age'].between(18,25)]
5. df = pd.get_dummies(df)
6. X = df.drop(['Outcome', axis = 1])
7. y = df['Outcome']
```

(a) Input script.

```
1. improrт pandas as pd
2. df = read_csv('diabetes.csv')
3. df = df.fillna(df.mean())
4. df = df[df['Age'].between(18,25)]
5. df = df[df['SkinThickness'] < 80]
6. df = pd.get_dummies(df)
7. X = df.drop(['Outcome', axis = 1])
8. y = df['Outcome']
```

(b) Output script.

Figure 1: Example input and output scripts.

In our view, data preparation scripts should be as standardized and boring as possible. The ideal data preparation script does its job and strives for standardization and predictability. The data management community has made many attempts to standardize data preparation, but only in a *top-down* fashion by proposing rules and systems [27, 31, 38, 39, 56, 57]. However, perhaps due to the richness of the project- and data-specific knowledge needed for data preparation, data scientists still resort to general-purpose programming languages [67] such as Python.

Intuitively, a script is *more standard* if it uses data preparation steps that are frequently used in other scripts processing the same or similar datasets. In this work, we adopt a common assumption in crowdsourcing [66]: the majority is presumably correct. Specifically, if a data preparation step appears repeatedly across multiple scripts processing the same dataset, it is likely to be both important and correct. Based on this assumption, we propose a *bottom-up standardization approach*, in which the user inputs a sketch script and gets a modified script as output. This output script aims to perform the same task as the input script, but is more similar to previously written scripts. Crucially, the modified script is allowed to emit data that are not identical to those emitted by the original script. The modifications to the input script can be viewed as recommendations for the user, who has the option to accept or reject them. The following example illustrates how bottom-up standardization facilitates data preparation for data scientists:

Table 1: An illustration showing the steps in historical scripts (s_1, s_2, s_3), the input script s_u, and the output script $\hat{s}_u$. Green shows added steps, and red shows removed steps.

	Data preparation step	s_u	s_1	s_2	s_3	$\hat{s}_u$
a_1	import pandas as pd	✓	✓	✓	✓	✓
a_2	df = pd.read_csv('diabetes.csv')	✓	✓	✓	✓	✓
a_3	df = df.fillna(df.median())	✓				
a_4	df = df[df["Age"].between(18,25)]	✓				✓
a_5	df = df.fillna(df.mean())		✓		✓	✓
a_6	df = df[df["SkinThickness"] < 80]		✓	✓		✓
a_7	df = pd.get_dummies(df)	✓	✓	✓	✓	✓

Example 1.1. Alex, a data scientist hired to help a medical research team, is writing a data preparation script (Figure 1a) to transform a patient dataset [5]. It will be used to train a prediction model that accurately identifies diabetes in young adults aged 18-25. She examines previous data preparation scripts on Papers With Code [20], which were developed for other similar projects. =Table 3 outlines the data preparation steps used in Alex's initial script (s_u), in three example scripts from the corpus (s_1, s_2, s_3), and in the modified output script ($\hat{s}_u$). Alex quickly feels discouraged due to the volume and required domain knowledge. Consequently, she writes a script from scratch. Specifically, she: **(1)** Imputes missing values using the median value (`df.fillna(df.median())`), based on her experience with other projects (Table 1, a_3); **(2)** Selects only relevant records (`df[df['Age'].between(18,25)]`), based on her modeling objective (Table 1, a_4).

Every step in Alex's script is seemingly correct, but unfortunately, she fails to consider the common practices to process this dataset. The resulting downstream model is less accurate because outliers in the data were not addressed, leading to distractions that reduce the model's accuracy. The error, which arises during data preparation due to a lack of domain-specific knowledge that Alex does not possess, is unlikely to be noticed by readers of Alex's paper or experimental results.

Now imagine that Alex uses a *script-standardization* system:

Example 1.2. Alex is writing a data preparation script from scratch, before giving it to a script standardization system. Then she sets a 5% threshold for the maximum % change that the standardization system is allowed to implement. In this case, the value of 5% refers to the maximum change in predictive accuracy allowed in the downstream model. The system responds with a standardized script (Figure 1b) that does the following:

(1) Recognizes that imputing missing values with average is a more common practice than median imputation for Alex's dataset (Table 1, a_3 removed and a_5 added);
(2) Selects only the relevant patient records, to maintain Alex's modeling objective (Table 1, a_4 unchanged);
(3) Adds additional step to handle outliers using domain-specific knowledge[1] embedded in other scripts (`df[df["SkinThickness"] < 80]` — Table 1, a_6 added).

Alex sees the modified output script, and decides to accept all the recommendation changes. Her improved script now uses more standard transformations. She is also pleased that her prediction accuracy has increased slightly due to the changes (although this will not happen in all cases).

[1]Triceps skinfold thickness is a measurement used to estimate body fat by assessing the thickness of a fold of skin. A value above 80mm is considered abnormal [45].

Our Approach: The goal of our approach is to modify an input script to produce a new script **similar in intent** to the input script, while being more **standard** (i.e., aligning with common data preparation steps from prior scripts). The user begins with a draft program, which our framework treats as a **semantic sketch** of their intent. We adjust the semantics of the new script to balance two objectives: increasing its standardization and maintaining an **agreement with the user's intent**. A straightforward approach is to leverage powerful code cleaning tools such as Sourcery [7], LLMs such as GPT [52], or recommendation systems predicting the next-best data preparation steps such as [44, 63]. However, our experimental results show that while these tools can generate code, they often fail to capture the user's original intent and struggle to select data preparation steps that are particularly relevant to the dataset, as informed by domain knowledge embedded in the corpus. We argue that to obtain such a system, there are several challenges to overcome: (**C1**) Defining how to capture script standardization in relation to a given corpus of scripts. (**C2**) Quantifying how well the user intent is preserved in the modified script. (**C3**) Navigating the vast search space (which is exponential in the number of data preparation operations present in the corpus) of potential modifications to the input script to find a sequence of changes that produces an executable script, maintains the user's intent, and achieves the highest degree of standardization.

To address **C1**, we define the standardness of a script w.r.t. a corpus of scripts. We assume the availability of a corpus of scripts processing the same or similar datasets for different objectives (as done in other frameworks [17, 63]). Our experiments show that even a small corpus yields valuable results. Our optimization objective is to minimize the relative entropy [41] of the data preparation step distribution between the input draft and the script corpus. Relative entropy is a common measure of how one probability distribution (the input script) diverges from a second, expected probability distribution (the scripts in the corpus). We extract these distributions using a novel graph-based script representation, leveraging NLP techniques such as lemmatization [54] and n-grams.

To address **C2**, we need to determine whether the input and output scripts perform similar tasks by examining their outputs or the results of their downstream applications. We use two measures to assess how well the output script preserves the user's intent and that the modifications to the input script do not significantly alter the data or impact the performance of the downstream task: (1) a table distance measure, which compares the structured datasets generated by the input and output scripts, and (2) a performance measure that evaluates downstream application performance, such as model accuracy or fairness.

User intent is treated as a constraint on the similarity of the input script and the modified script: the user provides a threshold indicating how much they are willing to compromise on their intent. The objective is to find a script that stays within this threshold of similarity to the original script, remains executable, and is as standardized as possible (i.e., minimizing the relative entropy).

To address **C3**, we need an efficient way to find a sequence of modifications to the input script, as an exhaustive search is impractical given the large search space (comprising all combinations of data preparation steps that are present in the corpus), and even greedy approaches present challenges. Our method has two phases: during the offline phase, we represent each script in

the corpus as a graph and curate the search space; during the online phase, we employ a search framework with five algorithmic components (including diversity, sampling, and early-checking criteria) to explore enough of the search space while meeting the constraints.

Usability: The proposed system is particularly useful in scenarios where users are working with datasets that are commonly used by other researchers, such as in public policy and medicine. For instance, datasets such as Amazon reviews[2] or MIMIC[3] are widely used by researchers and are often accompanied by open-source data preparation scripts available online (e.g., on platforms such as Papers With Code[4]). A researcher new to these open-source datasets can benefit from our approach by quickly identifying standard data preparation steps used by other researchers. As we show experimentally, the corpus needed to gain valuable insights can be relatively small, making the system practical for real-life scenarios.

A demonstration of our system usability and its suitability for end-to-end employment was recently shown in [42]. The short paper accompanying the demonstration provides only a brief, high-level description of the system, whereas the present paper provides the theoretical foundations and algorithms underlying the demonstrated system, as well as the experimental study.

Contribution — Our main contributions are as follows:

- We define a new problem of script standardization that helps data consumers write standardized data preparation scripts while preserving their intent (Section 4).
- We present an efficient and effective search framework and apply optimization techniques (Section 5).
- We evaluated our prototype system, LucidScript, against state-of-the-art approaches [6, 7, 16, 52, 55] on six real-world datasets (Section 6). Improvement is measured using our standardization measure, and we verified that this measure aligns with human judgment through a user study. Our approach obtained a 39.5% improvement on script standardness while, the second best competitor, GPT-4, achieved 2.9% (Table 5). We also evaluated our system in scenarios when high-quality on-topic script corpora were not available. We tested it using a smaller corpus, as well as using a corpus from a different dataset. Our approach still obtained improvements on script standardness.

2 SCRIPT STANDARDIZATION

A data preparation script is a sequence of lines of code that process some dataset for a downstream task, such as training a model, generating visualizations, etc. Let $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ be a collection of n scripts (referred to as the corpus). Let s_u denote an input user script. We assume that s_u and all scripts in $\mathcal{S}$ process the same dataset D_{IN}. In Section 6, we will empirically show that this assumption can be relaxed. For simplicity, throughout the paper we assume that D_{IN} comprises a single data file. Applying a script s to D_{IN} yields an output dataset D^s_{OUT}. The goal of our system is to modify the input script s_u to yield a new script $\hat{s_u}$ such that they both accept D_{IN} as input, and: (1) Both programs compute a **"similar"** result, measured by how well the user intent is preserved; (2) $\hat{s_u}$ is a more **conventional** program than s_u, according to statistics computed over the corpus $\mathcal{S}$.

[2]https://jmcauley.ucsd.edu/data/amazon/
[3]https://archive.physionet.org/physiobank/database/mimicdb/
[4]https://paperswithcode.com/

2.1 Measuring Goal One: User Intent

Measurement of the semantic similarity of two scripts is an extensively studied topic, often measured by the output of the operation [12, 49, 68]. In this work, we measure whether two scripts are doing something similar by looking at their outputs, or the outputs of their downstream application. We use these measures to assess how well $\hat{s_u}$ preserves the user's original intent embodied in s_u.

Table Jaccard: An example measure is table distance, which focuses on a structured dataset emitted by the script. This would be appropriate when the user produces databases for visualization, OLAP pipelines, or machine learning training procedures. Jaccard index [47] is commonly used to measure how close the new script's output dataset $D^{\hat{s}_u}_{OUT}$ is to the original user script's output dataset $D^{s_u}_{OUT}$. Formally, let $\Delta_J(D^{s_u}_{OUT}, D^{\hat{s}_u}_{OUT})$ denote the table Jaccard similarity defined as:

$$\Delta_J(D^{s_u}_{OUT}, D^{\hat{s}_u}_{OUT}) = \frac{|D^{s_u}_{OUT}| \cap |D^{\hat{s}_u}_{OUT}|}{|D^{s_u}_{OUT}| \cup |D^{\hat{s}_u}_{OUT}|}$$

$\Delta_J(D^{s_u}_{OUT}, D^{\hat{s}_u}_{OUT})$ ranges from 0 to 1, where $\Delta_J = 1$ indicates that the two tables are identical.

Example 2.1. Script $\hat{s_u}$ has an additional step to normalize all strings to lowercase, which is not present in s_u. $|D^{s_u}_{OUT}|$ = {'benign', 'Benign', 'High Risk', 'High risk', 'high risk'} and $|D^{\hat{s}_u}_{OUT}|$ = {'benign', 'high risk'}. Table jaccard would be $\Delta_J(D^{s_u}_{OUT}, D^{\hat{s}_u}_{OUT}) = \frac{2}{5} = 0.4$.

Model Performance: Another example measure of user intent is the performance of the downstream data application, which is appropriate when we have a downstream task with an easy-to-measure quality metric, such as accuracy.

Formally, let $\Delta_M(D^{s_u}_{OUT}, D^{\hat{s}_u}_{OUT})$ denote the absolute value of the relative percentage change in model accuracy, which is in [0%, 100%]. $\Delta_M = 0\%$ means the two datasets are identical.

$$\Delta_M(D^{s_u}_{OUT}, D^{\hat{s}_u}_{OUT}) = \left| \frac{acc(D^{s_u}_{OUT}) - acc(D^{\hat{s}_u}_{OUT})}{acc(D^{s_u}_{OUT})} \right| \times 100\%$$

Example 2.2. The model accuracy for s_u and its $\hat{s_u}$ is $acc(D^{s_u}_{OUT}) = 0.65$ and $acc(D^{\hat{s}_u}_{OUT}) = 0.67$, respectively. The model performance measure would be $\left|\frac{0.65-0.67}{0.65}\right| \times 100\% = 3.1\%$.

Our framework can be generalized to support other user-intent measures, such as earth mover distance or fairness constraints [30], as discussed in Section 8. We chose table Jaccard and model performance as example measures since they are two of the most straightforward metrics to understand.

2.2 Measuring Goal Two: Standardization

Our overall goal is to create new versions of the input script that are as standard and conventional as possible. We use *relative entropy* (also known as the KullbackâĂŞLeibler divergence) [41] to measure how standard a script is. Relative entropy is a non-negative statistical distance of how one probability distribution $P(x)$ is different from a reference probability distribution $Q(x)$. We adapt the notation of relative entropy to measure script standardness. Formally, the relative entropy of a script s w.r.t. a corpus $\mathcal{S}$ is defined as:

$$RE(s, \mathcal{S}) = \sum_{x \in \mathcal{X}} P(x) log(\frac{P(x)}{Q(x)})$$

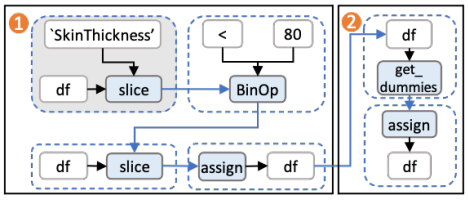

Figure 2: Example DAG of line 5 and line 6 in Figure 1b. The DAG is made of atoms (blue dashed) and edges (blue). Inside an atom, there are AST nodes and edges, where invocation nodes are blue, and data nodes are white. Arrows represent data flows. We color one atom gray to illustrate Example 3.2.

where $\mathcal{X}$ is the sample space shared by both P and Q.

In our problem, $\mathcal{X}$ is the space of all the data preparation steps in $\mathcal{S}$. $P(x)$ is the probability distribution of the data preparation steps in script s, while $Q(x)$ is the probability distribution of the steps in corpus $\mathcal{S}$. A large distance between $P(x)$ and $Q(x)$ would indicate that s uses many data preparation steps that are not commonly used in $\mathcal{S}$, while a small distance would mean that s does a good job adhering to the standard practices in $\mathcal{S}$.

Formally defining $P(x)$ and $Q(x)$ is a non-trivial task since scripts need to be in a format that makes it easy to compute these statistics. We describe such a representation in Section 3. We then use $RE(s, \mathcal{S})$ as our optimization objective in Section 4.

Problem Objective. The *script standardization* problem is described by a tuple comprising: (i) A **script corpus** $\mathcal{S} = \{s_1, \ldots, s_n\}$; (ii) An **input user script** s_u; (iii) An **input dataset** D_{IN}; (iv) A **user-intent parameter** $\Delta(D_{OUT}^{s_u}, D_{OUT}^{\hat{s}_u}) \leq \tau$. The solution is a new, executable script $\hat{s_u}$ that maximizes the standardness measured by $RE(\hat{s_u}, \mathcal{S})$ while ensuring that the output of the resulting script is within τ of the original.

3 SCRIPT REPRESENTATIONS

We face a multifold challenge — to represent an arbitrary script while also being able to explore the space of legal changes to it. Concretely, the representation should enable us to compute standardization statistics efficiently and numerate the space of legal changes. It should abstract away inessential syntactic differences. Lastly, it should retain sufficient information about the script so that it can be translated back to a script. Inspired by previous work [29, 40, 48], we present a directed acyclic graph (DAG) representation of scripts that satisfies the above requirements for our setting.

We now introduce how we represent the entire search space, including the DAG representation of scripts and transformations that can be applied to DAGs. In Section 4, we will describe how our script representations are used to compute the probability of observing a particular script s.

DAG Representation: We start by representing a single script using a DAG, in which nodes represent operation invocations, and edges represent data flows. The DAG representation $G_s = (A, E')$ of a script s is based on its Abstract Syntax Trees (ASTs). An AST is a tree-like data structure used to represent the syntactic structure of a code snippet in a hierarchical and abstract way. Each node corresponds to a construct in the code, such as an operator, a variable, or a control structure, and its children represent the components or sub-expressions of that construct. ASTs focus on the logical relationships and structure of the code, and are widely used in compilers and tools for code analysis, transformation, and optimization. For a formal definition of ASTs for Python, see [1].

A simple script can result in hundreds of AST nodes, and most nodes make sense only in the scope of their context. If we curate the search space directly on the AST nodes without considering their neighbors, the space is huge, and most transformations would be invalid. For example, for line 5 in Figure 1b, replacing the AST node `80` with `'SkinThickness'` would result in an execution error since < can only compare numerical values.

Inspired by the natural language processing (NLP) applications [16, 18, 23, 55], where atoms are defined at the word level, we construct atoms at the operation-invocation level. The DAG representation $G_s = (A, E')$ of a script s is made of two components, atoms A and edges E' (Figure 2). An atom a, or atomic unit, in our DAG representation, is an operation invocation. In programming languages, an operation invocation refers to a snippet of code required to call a function, with its arguments. Instead of AST nodes, the use of atoms helps to make the output scripts free of syntax and execution errors and also helps to reduce the search space. Thus, we define each atom $a = (V, E)$ as a collection of AST nodes V and edges E. We simplify AST nodes to two types, data nodes, and invocation nodes (Figure 2). We later perform lemmatization techniques on atoms, which helps us to further control the size of the vocabulary and leverage the semantics of the operator invocations (Section 5).

Definition 3.1. [Atom and DAG] An *atom*, is the atomic unit of the DAG representation. Using ASTs, we denote an atom by $a = (V, E)$. We can express the DAG $G_s = (A, E')$, with atoms A and edges between atoms E'. Specifically, an atom a consists of one invocation node and its parents that are not invocation nodes.

Example 3.2. An atom is an operation invocation. The right block shows the DAG representation of `df = df[df['SkinThickness'] < 80]` (Figure 2). Specifically, the gray-color atom represents `df['SkinThickness']`, where `df` and `'SkinThickness'` are AST data nodes and `slice` is an AST invocation node. The black arrows are the edges E inside the atom, representing data flows.

Atoms can be used in 1-gram, n-gram, or in a way learned from the script collection $\mathcal{S}$. We use atoms in two ways, at the operation-invocation level (1-gram) and the line level (n-gram). In Figure 2, the gray-color atom is an example of 1-gram atoms, while the numbered blocks are examples of n-gram atoms.

From Script to DAG: Some components in the standardization measure of script are now more clear. $\mathcal{X}$ denotes all the data preparation steps in $\mathcal{S}$. As Figure 2 shows, using the DAG representation, the data preparation steps are decomposed into atoms A and edges E'. Formally, $\mathcal{G} = \{G_{s_1}, \ldots, G_{s_n}\}$ is the DAG representation of $\mathcal{S} = \{s_1, \ldots, s_n\}$. Denote by $\mathcal{V}_A$ the vocabulary that contains unique atoms $A \in \mathcal{G}$ and by $\mathcal{V}_{E'}$ the vocabulary that contains unique edges $E' \in \mathcal{G}$. We use $\mathcal{V}_{E'}$ to model $\mathcal{X}$, instead of $\mathcal{V}_A$, since E' encodes the data flow information, representing the order of the steps.

Example 3.3. Figure 2 shows the DAG representation of `df = df[df['SkinThickness'] < 80]` and `df = pd.get_dummies(df)`, which are lines 5-6 in Figure 1b. The data flow between lines 5-6 is represented by the blue edge e' between the numbered blocks, connecting two data nodes.

Transformations: Transformations are the actions we perform to change the DAGs. We define two types of transformations, add

and delete.[5] Each transformation takes the following parameters: type, what to change (i.e., one atom and its edges), and where to change (i.e., line number). Next, we formally define the notation of a transformation over G_s.

Definition 3.4. [Transformation] A transformation is a function of the form $f(\text{type}, a, \{e'_1, \ldots, e'_n\}, \text{lineno})$, where type $\in$ [add, delete], $a \in \mathcal{V}_A, e'_i \in \mathcal{V}_{E'}$. $\{e'_1, \ldots, e'_n\}$ is the set of edges that connect an atom a to its neighbors. $F(G_s) = (f_1, f_2, \ldots, f_m)$ is sequence of m transformations that preform on G_s.

We discuss how we configure transformations in Section 5.2.

4 PROBLEM FORMULATION

We have defined the measure of script standardization (Section 2.2) and presented script representations (Definition 3.1), which help us to explore the space of transformations (Definition 3.4) that can be applied to the input script. We are now ready to present our standardization objective and the script standardization problem.

Definition 4.1. [Standardization Objective] The relative entropy of a script s_u w.r.t. a collection of scripts $\mathcal{S}$.

$$RE(s_u, \mathcal{S}) = \sum_{\mathcal{X}} P(x) log(\frac{P(x)}{Q(x)})$$

where $P(x) = \frac{x}{\sum_{i=1}^{|\mathcal{V}_{E'}|} x_i}$, and $x_i = \sum_{i=1}^{|\mathcal{V}_{E'}|} [e_j = v_i], \forall e_j \in G_{s_u}$. Similarly, $Q(x) = \frac{x}{\sum_{i=1}^{|\mathcal{V}_{E'}|} x_i}$, and $x_i = \sum_{i=1}^{|\mathcal{V}_{E'}|} [e_j = v_i], \forall e_j \in \mathcal{G}$.

$P(x)$ is the probability distribution of the data preparation steps in script s_u, while $Q(x)$ is the probability distribution of the data preparation steps in the script corpus $\mathcal{S}$ (Section 2.2). We use $\mathcal{V}_{E'}$ to derive the model of the data preparation steps $\mathcal{X}$. We use atoms in Table 1 to illustrate how we compute $P(x)$ and $Q(x)$.

Example 4.2. The DAG of an input script s_u is $G_{s_u} = (A_{s_u}, E'_{s_u})$, where $E'_{s_u} = [(a_0, a_1), (a_1, a_7)]$ are the edges of G_{s_u}. $\mathcal{V}_{E'}$ is the edge count mapping from $\mathcal{G}$, where $\mathcal{V}_{E'} = \{(a_0, a_1) : 3, (a_1, a_2) : 3, (a_2, a_7) : 2, (a_1, a_7) : 1\}$. For example, the edge $e' = (a_0, a_1)$ appears three times in $\mathcal{G}$, where $\mathcal{G} = \{G_{s_1}, G_{s_2}, ..., G_{s_n}\}$ is the DAG representation of the script corpus $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$.

We derive the distribution of the data preparation steps in the corpus $Q(x)$ using edge vocabulary $\mathcal{V}_{E'}$ and $\mathcal{G}$. $Q(x)$ and x are both vectors with a size of $|\mathcal{V}_{E'}|$. We compute x as the number of occurrences for each edge in $\mathcal{G}$. Then we sum up the occurrences of each edge and divide x by the total occurrences to get $Q(x)$.

Example 4.3. In $Q(x)$, x is derived from $\mathcal{V}_{E'}$, where $x = [3, 3, 2, 1]$ since the number of unique edges $|\mathcal{V}_{E'}|$ is four. The total number of edges in $\mathcal{G}$ is $3 + 3 + 2 + 1 = 9$ so $Q(x) = [\frac{1}{3}, \frac{1}{3}, \frac{2}{9}, \frac{1}{9}]$.

Then we compute $P(x)$ from $G_{s_u} = (A_{s_u}, E'_{s_u})$. Similarly, $P(x)$ and x are vectors of the length of the edge vocabulary $|\mathcal{V}_{E'}|$. We compute x as the number of occurrences for each edge in G_{s_u}, where $E' \in \mathcal{V}_{E'}$.

Example 4.4. In $P(x), x = [1, 0, 0, 1]$. The total number of edges in G_{s_u} is $1 + 1 = 2$, so $P(x) = [\frac{1}{2}, 0, 0, \frac{1}{2}]$ and $RE(s_u, \mathcal{S}) = 1.38$.

We note that $RE(\hat{s}_u, \mathcal{S})$ accounts for atoms and edges, but is oblivious to their exact position (i.e., line number) in the script. We explain how we handle this in Section 5.

[5]The edit operation is modeled here as a sequence of delete and add operations.

Revisiting the conceptual definition in Section 2.2, we are now ready to formally define the problem using the script representations in Section 3. The Script Standardization problem aims to find a sequence of transformations F for an input script s_u to make it as standard as possible w.r.t. a collection of scripts $\mathcal{S}$, while ensuring that: (1) There are no execution errors; (2) No more than k transformations are applied; (3) User intent is preserved. Formally,

Definition 4.5. [Script Standardization] Given a script s_u and a collection of scripts $\mathcal{S}$ all operating on the data file D_{IN}, a number $k \geq 1$, and a threshold τ we search for a script $\hat{s}$ such that:

(1) **(sequence length constraint)** $\hat{s_u}$ is obtained by sequentially applying no more than k transformations on s_u.
(2) **(execution constraint)** $\hat{s_u}$ is executable.
(3) **(user-intent constraint)** $\Delta(D_{OUT}^{s_u}, D_{OUT}^{\hat{s}_u}) \leq \tau$.
(4) **(objective function)** $RE(\hat{s}_u, \mathcal{S})$ is minimized.

Each transformation f increases or decreases $RE(\hat{s}_u, \mathcal{S})$.

Example 4.6. Continuing with the simple setup in Example 4.2, the best transformation f is trivial: add a_2 between a_1 and a_7. This f leads to $P(x) = [\frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0]$, which gives $RE(\hat{s}_u, \mathcal{S}) = 0.2$.

$RE(\hat{s}_u, \mathcal{S})$ is much smaller than the starting $RE(s_u, \mathcal{S})$ in Example 4.4, indicating that $P(x)$ has come closer to $Q(x)$. This means that the resulting script is more standard w.r.t. the script corpus $\mathcal{S}$.

To find the best transformations to change s_u, we need to find the sequence that minimizes $RE(\hat{s}_u, \mathcal{S})$. However, the search space can become intractable quickly, as there could be many possible places to apply a transformation in s_u. For example, a transformation to append a new atom can have multiple possible existing atoms to append to. In addition, possible next-step transformations depend on the current step. As a result, an exhaustive search algorithm is exponential in $(|\mathcal{V}_A| \times |A_{s_u}| + |E'_{s_u}|)$.

Therefore, in the next section, we will present an efficient algorithm for our script standardization problem. This algorithm overcomes the obstacles of the naive exhaustive search algorithm by taking a greedy-like approach while ensuring that a diverse set of executable candidate scripts is taken into consideration.

5 METHODOLOGY

5.1 Offline Search Space Curation

In the offline phase, we construct the atom vocabulary $\mathcal{V}_A$ and the edge vocabulary $\mathcal{V}_{E'}$ and obtain the corpus distribution. These components then serve as inputs to the online phase. We take the following steps to construct them from the corpus $\mathcal{S}$: (1) We parse each script $s_i \in \mathcal{S}$ into its corresponding DAG G_{s_i}, simultaneously building $\mathcal{V}_A$ and $\mathcal{V}_{E'}$ through this process. (2) We compute the probability distribution $Q(\boldsymbol{x})$ from $\mathcal{V}_{E'}$, as described in Section 3.

Reducing Vocabulary: A single data preparation step can be represented through various coding operations that convey an equivalent semantic meaning. For instance, when both `df` and `train` are read from the same CSV file, expressions like `df['Age']` and `train['Age']` refer to the same column. We address such semantically equivalent steps by leveraging a rich body of work in NLP [54] and static code lemmatization techniques. For example, we unify the variables created by reading from the same data file to a consistent name across all scripts. Lemmatization significantly reduces the size of the vocabulary

and helps the system better understand the semantics of scripts without actually executing them.

The semantic similarity of two scripts is an extensively studied topic in programming languages [12, 49, 68]. LLMs [6, 52] have also been proven to excel at such tasks. More optimizations can be extended without changing other system components.

5.2 Online Phase Search Framework

The next goal is to find a sequence of transformations that standardizes a given script s_u while satisfying the constraints (Definition 4.5). We model this sequence searching task as a constrained optimization problem with the objective to minimize its relative entropy measured w.r.t. the corpus $RE(s_u, \mathcal{S})$.

Recall in Section 4, a simple greedy approach faces two challenges. First, the optimal sequence may not always score the best during the search process. Discarding all but the best in-progress sequence may eliminate many potentially good candidates. Second, the output script $\hat{s_u}$ must satisfy the constraints, and checking for the constraints only at the end may result in invalid sequences (e.g., execution errors, failure to satisfy user-intent constraints). We propose five optimizations that overcome the limitations and evaluate the impact of these components on our framework (Section 6).

(1) Beam search: We keep K beams rather than a single best, which retains multiple best options during the search.

(2) Diversity measure: We consider transformations that are different enough every time we add one step to the beams, which explores different parts of the search space.

(3) Monotonicity: A transformation sequence cannot go back and change an earlier portion of the script, denoted by the line number. This property ensures that once a script becomes non-executable after a transformation, no further transformation would make the script executable again. This helps reduce the search space.

(4) Checking strategies for constraints: We have three constraints. (1) We use the sequence-length constraint as the stopping criterion to limit the number of modifications applied to the input script. (2) For the execution constraint, we have an early- and late-checking strategy. In early-checking, we remove candidate sequences that lead to non-executable scripts after every transformation is applied. In late-checking, we only check for execution errors after the sequence is fully developed. (3) We check the user-intent constraint at the end. These strategies ensure that our approach always finds valid sequences while keeping runtime in check.

(5) Sampling: When D_{IN} is large, we apply random sampling on the tuples. This helps reduce runtime.

Algorithm 1 is our search framework, which takes as input a script collection $\mathcal{S}$, a script s_u, beam size K, user-intent threshold τ, and an early-checking switch α and returns the standardized script $\hat{s_u}$. In line 1, the candidate script set is initiated with the input script. In line 2, the vocabularies and distribution of the data preparation step are computed. In lines 3-8, beam search is applied to find the top K scripts in each transformation step, until the stopping criteria are met. In lines 9-11, constraints are verified and the best result is returned.

In Algorithm 1, GetSteps() takes a script s and the search space $\mathcal{V}_A, \mathcal{V}_{E'}$, and $Q(\boldsymbol{x})$ as input, and outputs a list of transformations ranked based on the RE score. This step can be divided into three parts: (1) Translate s to its DAG representation $G_s = (A, E')$. (2) Given $\mathcal{V}_A, \mathcal{V}_{E'}$, enumerate through G_s to find possible next-step transformations. (3) For each possible transformation, compute the RE score and determine where in s to perform each transformation.

Algorithm 1: A meta-level framework

Input: $\mathcal{S}$ (script collection), s_u(input script), seq (sequence length), K (beam size), τ (user-intent threshold), α (early checking)
Output: $\hat{s_u}$ (a standardized version of s_u)

```
1  C = {s_u} ;                          /* Set of all candidates. */
2  V_A, V_E', Q(x) = CurateSearchSpace (S);
3  while not CheckStoppingCriteria (C, seq) do
4      C' = ∅ ;                         /* Set of new candidates. */
5      foreach s ∈ C do
6          F = GetSteps (s, V_A, V_E', Q(x)) ;  /* A ranked set
               of next steps based on RE score. */
7          C' = GetTopKBeams (C', s, K, α, F);
8      C = C'
9  C = VerifyAllConstraints (C, τ);
10 ŝ_u = GetTopCandidate (C);
11 return ŝ_u;
```

Configuring Transformations: Our approach uses two types of transformations, add and delete, with two types of atom to consider, 1-gram and n-gram (Section 4). To generate all possible transformations, we enumerate the combination of which atom to change and where to apply the change. Configuring *delete transformations* is straightforward. We simply detect all existing atoms in the script and configure a list of delete transformations with the atoms, edges, and positions. Configuring *add transformations* is more elaborate since the location to add is unknown. For 1-gram atoms, we leverage the edges to find the possible locations to append. For every atom a in the script s, a new atom $a' \in \mathcal{V}_A$ is considered a possible candidate to be appended after a if there exists (a, a') in $\mathcal{V}_{E'}$. For n-gram atoms, we use the script corpus $\mathcal{S}$ to find possible locations in s. When curating the search space, we retain the relative location of each n-gram atom in $\mathcal{S}$. When determining where to add a new n-gram atom, we compute their possible positions in s using the relative locations observed in $\mathcal{S}$.

After a transformation is configured, its impact on the RE score is calculated by marginally updating its vector $P(\boldsymbol{x})$ instead of performing the actual transformation in DAG. We have yet to describe GetTopKBeams(), where we extend each candidate sequence by one transformation. We propose a few search strategies that will be discussed next.

5.3 Extending By One Transformation

Given the list of configured transformations, we have two challenges when deciding which transformation to append to a sequence: (1) The search space is still large due to the exponential number of possible sequences and the large number of unique atoms in $\mathcal{S}$; (2) The validity of the execution and user-intent constraints of a candidate sequence is difficult to estimate during search.

We design Algorithm 2 that applies the concepts discussed in Section 5.2 and addresses those challenges with the following. In line 4, transformation f is applied to G_s. In line 5, the transformed script s' is verified for the execution constraint. In lines 6-7, the top K beams are updated by swapping out the script with the highest RE score in C' with s'. The beam search in our search strategy keeps track of multiple candidate sequences with potential best RE scores while keeping the search space tractable. This algorithm also takes a parameter α that controls

Algorithm 2: GetTopKBeams

Input: C (current candidates), s (a script), K (beam size), α (early checking), $\mathcal{F}$ (a ranked set of next steps)
Output: C' (updated candidates)

```
valid = True;
C' = C;
foreach f ∈ F do
    s' = ApplyStep (s, f);
    if α then valid = CheckIfExecutes (s');
    if Score (s') < min_{s∈C} Score(s) or |C| ≤ K then
        if valid then C' = UpdateKBeams (C', s', K);
return C';
```

the early checking for execution constraints. When α = True, early-checking is *on*, which means that the execution constraint is verified every time after a transformation f is applied to G_s. This ensures that the algorithm retains beams that are free of execution errors during the search.

A problem with beam search is that the top-ranked next transformations can be similar. This is because the $RE(s', \mathcal{S})$ score compares the atoms in s' and $\mathcal{S}$ so the high-ranked transformations would suggest changing the same atom. We then design an alternative to Algorithm 2 that incorporates a diversity measure to allow the beams to explore different transformations.

Transformation Diversity Measure: We use K-means clustering [32] to group similar transformations. ClusterSteps() takes a ranked list of transformations, output by GetSteps(). ClusterSteps() then takes the updated vectors and groups them into M clusters. Within each cluster, we then rank the transformations according to the RE scores and the output $\mathcal{F}_M$. In GetDiverseTopKBeams(), we iterate through each cluster and call Algorithm 2.

Algorithm 3: GetDiverseTopKBeams

Input: C (current candidates), s (a script), K (beam size), α (early checking), $\mathcal{F}$ (a ranked set of next steps)
Output: C' (updated candidates)

```
C' = C;
F_M = ClusterSteps (F);
foreach F_{i∈M} do
    C' = GetTopKBeams (C', s', K/M, α, F);
return C';
```

Parameterizing our framework: Algorithm 1 provides the user with a set of parameters to explore and tailor in a flexible way to their use case. Based on our main experiment (Section 6.3.3) and ablation studies (Section 6.4) on six real-world datasets, Table 2 lists the possible default parameters based on different corpus properties. In addition, the user-intent threshold allows the user to control the difference between the data processed by s_u and $\hat{s_u}$, which is set to $\tau_J = 0.9$ and $\tau_M = 1\%$ by default. We explore how varying the user-intent threshold affects our framework in Section 6.3.2. Lastly, early-checking is set to *on* as default to ensure the output script is executable. We discuss potential ways to automatically tune these parameters in Section 8.

6 EXPERIMENTAL EVALUATION

We embody our algorithm in a prototype system, LucidScript (LS), which was written in 3,000 lines of predominantly Python source code. Our current implementation supports straight-line Python scripts. Support for other programming languages and more complex programs can be addressed with more engineering efforts. Our experiment data, code, and artifacts are available [2].

Table 2: Parameterization effected by corpus properties.

Corpus properties		Parameters	
Large	Diverse	*seq*	K
# of scripts > 10	# of uniq. edges > 300	16	3
# of scripts > 10	# of uniq. edges ≤ 300	16	1
# of scripts ≤ 10	# of uniq. edges > 300	8	3
# of scripts ≤ 10	# of uniq. edges ≤ 300	8	1

We empirically demonstrate the following claims about our method as embodied in LS:

(1) An increase in our metrics (relative entropy and user intent) leads to better outcomes for data scientists.
(2) Our generated scripts are considered to be more standard than scripts generated by alternative approaches.
(3) LS performs better than competing methods in standardizing data preparation scripts with the user intent. We evaluated this on six real-world datasets with varying properties, such as data file size and corpus size.
(4) Each component of LS contributes meaningfully to its ability to find a sequence of transformations that satisfies our optimization goal and constraints.
(5) LS performs well in a reasonable time.
(6) Bottom-up script standardization can also be applied to identify anomalous data preparation steps.

6.1 Experimental Setting

6.1.1 Competing methods. We evaluate each of the following methods based on the performance metrics in Section 6.1.4.

- **Sourcery:** This is a commercial recommendation system to automatically improve code quality and ensure clean code [7].
- **GPT-3.5:** GPT-3.5 [6] is a language model with billions of parameters that enables it to generate text and code.
- **GPT-4:** GPT-4 [52] is an advanced iteration of GPT-3.5.
- **Auto-Suggest** [63]: This method uses machine learning models trained on real-world notebooks to predict a single next step for an input table based on its characteristics.
- **Auto-Tables** [44]: This method predicts multi-step transformations, among a set of table-reshaping operators.

The corpus can be accessed by the GPT baselines in two ways. Since GPT models are initially trained with scripts crawled from the internet [6, 52], the Kaggle datasets we used in the experiment should already be included in their training data. Another way for the GPT models to use the specific corpus more explicitly is to incorporate the corpus scripts in the prompt. Due to the limited token length, it is impossible for the prompts to include the entire script corpus. Empirically, we observed that prompts that include more scripts from the corpus achieve better performance. We describe how we select the prompts next.

6.1.2 GPT prompt survey. The performance of language models can be highly dependent on the problem prompt. We ensured that our experiments used a high quality prompt by polling 12 graduate students for prompt ideas, with access to the corresponding script corpus. We then ran all of them and chose the best-performing prompt to compare against our method. As a result, we believe that more typical users of LLMs would obtain worse results than what we report here. The survey and results are available in [2]. The best prompt randomly picks 4 scripts from the corpus and asks the LLMs to enhance the performance of the user script in terms of model accuracy or any other aspect that the LLMs find relevant.

Table 3: Examined datasets and their DAG statistics.

Statistics	Titanic	House	NLP	Spaceship	Medical	Sales
Scripts	62	49	24	38	47	26
Data files	3	4	3	3	1	6
Data tuples (k)	2.6	4.3	22.7	17.2	0.7	744.3
Data features	25	163	11	29	9	18
Avg # code lines	64	43	19	44	30	39
Uniq. 1-grams	625	659	175	427	281	347
Uniq. n-grams	548	221	48	201	143	114
Uniq. edges	748	632	193	423	220	308

6.1.3 Data collection and analysis. We used the Kaggle API [4] to obtain data preparation scripts in six competitions (Table 3). Specifically, we downloaded all available scripts attached to the chosen competitions, and used them as the input corpus. To evaluate the effectiveness of our system, for each competition, we treated each script in the corpus as an input user script and ran the system on the remaining scripts as the corpus. We report the average improvement on script standardness (calculated as explained in Section 6.1.4).

- **Titanic**: The *Titanic* competition aims to create a model to predict who survived the Titanic shipwreck.
- **Sales**: The *Predict Future Sale* competition aims to predict total sales for every product and store in the next month.
- **House**: The *House Prices* competition aims to use 79 variables to predict the final price of each home.
- **NLP**: The *NLP with Disaster Tweets* competition aims to distinguish real and fake tweets.
- **Spaceship**: The *Spaceship Titanic* competition aims to predict which passengers were transported from a spaceship.
- **Medical**: The *Pima Indians Diabetes Database* dataset aims to diagnostically predict if a patient has diabetes.

6.1.4 Performance metrics. We evaluated effectiveness using the percentage improvement in relative entropy (Definition 4.1), defined as % improvement $= \frac{RE(s_u,\mathcal{S})-RE(\hat{s_u},\mathcal{S})}{RE(s_u,\mathcal{S})}$. We used the user-intent constraint to evaluate how well the user intent is being preserved. Specifically, we varied the Jaccard user intent threshold τ_J between $[0, 1]$. The user intent is preserved more (i.e., given less flexibility to standardize) when τ_J is closer to 1. We varied the absolute percentage difference in the model performance user intent threshold τ_M between $[0\%, 5\%]$. The user intent is preserved more when τ_M is closer to 0%. For each dataset, we iterated through the individual scripts as the user input script s_u and used the rest as the script corpus. The methods output $\hat{s_u}$. We then calculated % improvement of each $\hat{s_u}$ relative to its original s_u.

6.1.5 Default LucidScript (LS) configuration. We set the LS default configuration as follows: sequence length $seq = 16$, beam size $K = 3$, diversity measure div = on, early-checking α = on. This configuration obtained the best result. We set the user-intent threshold $\tau_J = 0.9$ and $\tau_M = 1\%$. This is a relatively strict user-intent constraint since, for example, $\tau_M = 1\%$ means that the output script should be as standard as possible while keeping the difference in its model performance within 1%.

6.2 Case Studies & A User Study

6.2.1 Metric Evaluation. Our first goal is to validate that improvements in our metrics (relative entropy and user intent) actually lead to meaningful outcomes for data scientists. We demonstrate this through the following case study.

Table 4: Case study for metrics evaluation, where s_u is the input script, and s_1 and s_2 are the potential outputs.

	Script	RE	Δ_J	Δ_M
s_u	import pandas as pd import numpy as np df = pd.read_csv("/data/titanic/train.csv")	3.02	1	0%
s_1	import pandas as pd import numpy as np df = pd.read_csv("/data/titanic/train.csv") y = df["Survived"] X_train = df.drop("Survived", axis=1)	2.49	0.92	<0.1%
s_2	import pandas as pd import numpy as np df = pd.read_csv("/data/titanic/train.csv") df["Age"].fillna(df["Age"].mean()) df["Embarked"].fillna("S") y = df["Survived"] X_train = df.drop("Survived", axis=1)	1.37	0.90	<0.1%

Summary: An increase in our standardization metric indicates a more standardized data preparation script while maintaining the original user intent.

Results: We consider the Titanic competition. Table 4 shows an input script s_u and two output scripts s_1, s_2, with their corresponding RE score and user intent measures Δ_J, Δ_M (for table Jaccard similarity and model performance, respectively). In this case, the input script s_u simply loads the dataset. The user intent measures for s_u, as in the input script, are $\Delta_J = 1, \Delta_M = 0\%$. s_1, one of the potential outputs, involves a common step in which the target variable "Survived" is separated from the features. This data preparation step appears in 50% of the scripts within the corpus. As shown, the corresponding RE score increased by 27%, while remaining within the user intent measures of $\Delta_J = 0.9$ and $\Delta_M = 1\%$. Beyond the steps in s_1, a more "standard" output script, s_2, includes two additional common steps to handle missing values in the variables "Age" and "Embark". These steps are present in 45% and 14% of the scripts in the corpus. This modification further increases the RE score by 55% compared to the input script s_u. This example shows that the modified scripts incorporate common and logical data preparation steps. It also shows that an increase in the RE score corresponds to the inclusion of such steps while preserving user intent.

6.2.2 User Study. Our next goal is to validate that LS generated scripts that are considered by data scientists more standard compared to the output of the baselines. We recruited 34 undergraduate and graduate computer science students who are familiar with data preparation and Python programming. Each participant was given two use cases, including statistics on the prevalence of data preparation steps in the corpus (similar to Table 1), and 5 scripts output by LS and the baselines. In the without-user-intent case, participants evaluated the methods in a cold start setting, while in the with-user-intent case, participants were given an input script.

Participants were asked to rank each of the output scripts on a scale of 1 to 5 based on: (a) how standardized the script is w.r.t. the corpus, where standardization represents the commonality of the chosen data preparation steps; and (b) how helpful the script is w.r.t. preserving the user intent, measured by their modeling task. The complete form provided to participants is available [2].

Summary: LS was rated to be the most standard and helpful.

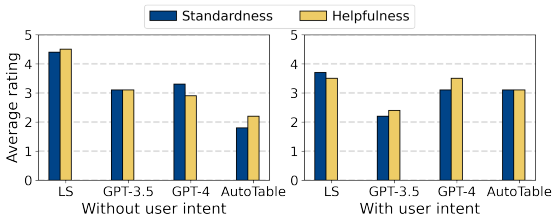

Figure 3: User study result.

Table 5: % improvement on Kaggle datasets, with $\tau_J = 0.9$ and $\tau_M = 1\%$. LS is set as default.

Corpus setup	Method	% Improvement			
		min	median	max	mean
Full-size corpus	LS (τ_J)	0.0	**33.1**	**72.3**	**33.6**
	LS (τ_M)	0.0	26.9	70.0	25.8
	GPT-3.5	-57.3	-1.5	58.4	-3.7
	GPT-4	-129.0	0.0	66.1	3.4
	Sourcery	0.0	0.0	0.0	0.0
	Auto-Suggest	0.0	0.0	0.0	0.0
	Auto-Tables	0.0	0.0	0.0	0.0
Small corpus	LS (τ_J)	0.0	18.5	69.3	20.3
	LS (τ_M)	0.0	15.0	69.3	17.1
Different corpus	LS (τ_J)	0.0	11.1	36.4	10.5
	LS (τ_M)	0.0	11.8	36.4	11.2
Low-ranked corpus	LS (τ_J)	0.0	5.4	32.9	7.8
	LS (τ_M)	0.0	5.2	32.9	7.7

Results: In the without-user-intent case, it is statistically significant that LS is more standard and helpful than baseline methods[6]. In the with-user-intent case, LS is the most standard and helpful among all 5 methods. Note that we omitted Auto-Suggest from Figure 3 since it had the same results as Auto-Tables.

6.3 Evaluating Effectiveness

We aim to answer the following three questions. Is our framework effective in standardizing data preprocessing scripts with the constraints? How does our method compare to the others? And how does our method perform on a variety of datasets?

6.3.1 Script standardization with constraints. We evaluated the % improvement of LucidScript (LS) with the default configuration against the three competing methods on six datasets.

Summary: LS consistently outperforms competing methods by far. None of the state-of-the-art methods can effectively improve script standardization.

Results: In Figure 4, the most optimal result would be a sharp peak on the most right, meaning that all input scripts get an improvement of 100%, while a curve centered around $x = 0$ means that the method does not improve script standardness. A curve extending to the left ($x \leq 0$) means that the method decreases script standardness.

LS achieves at least 30% improvement in script standardization while preserving user intent, other than Sales (Table 5). GPT models perform similarly with only a small positive average % improvement on Titanic (GPT-4: 7.7%) and House (GPT-4: 6.2%). Moreover, LS guarantees a standardness improvement, whereas GPT models do not produce reliable results, sometimes emitting a script that uses less common transformations than the user's original (GPT-4: -130%). This is because LS aims to preserve

[6]according to a t-test, p-value < 0.05

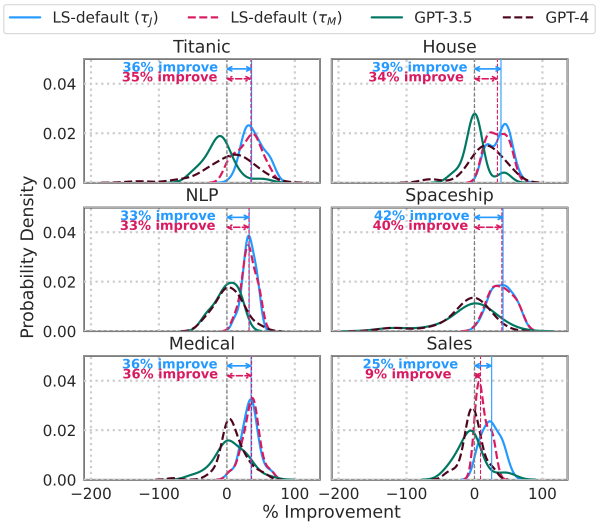

Figure 4: % improvement distribution. Sourcery, Auto-Suggest, and Auto-Tables are omitted.

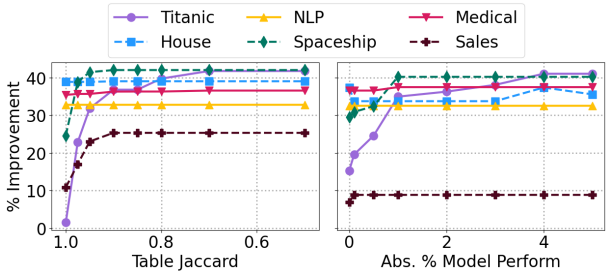

Figure 5: Median % improvement of LS-default with varied table Jaccard threshold τ_J (left); median % improvement of LS-default with varied model performance threshold τ_M (right).

the user intent while standardizing the script using the specific collective knowledge of D_{IN}. Although GPT models are trained on all publicly available scripts, they do not standardize the script with a focus on knowledge specific to D_{IN}.

We compared LS to two academic solutions, Auto-Suggest [63] and Auto-Tables [44], which are, respectively, the state-of-the-art work for single-step and multi-step table transformation prediction. These methods made a 0% improvement because they focus on a fixed set of table structural transformations, such as transpose and pivot. However, the real-world scripts that we observed focus on feature engineering and data cleaning tasks (Figure 1a). Our bottom-up approach makes LS robust to the type of data preprocessing tasks. We also compared with Sourcery, a commercial product for code cleaning, but observed that it consistently made no difference on all measures, as it focuses on syntax standardization.

6.3.2 Evaluating with user intent. We experimented with two user-intent measures, table Jaccard and model performance. We evaluated how performance changes with user intent thresholds, ranging from more lenient to more strict. We only evaluated LS since competing methods do not have this property.

Summary: LS can adhere to a wide range of user intent while improving script standardness in the six datasets.

Results: We varied table Jaccard τ_J between $[0, 1]$ (Figure 5). $\tau_J = 1$ means the $D_{OUT}^{s_u}$ and $D_{OUT}^{\hat{s}_u}$ are identical, which is the most strict user intent constraint, where the intent should be preserved in its entirety. We found that as this constraint becomes more relaxed (i.e., τ_J becomes smaller), LS can make the input scripts more standardized. In all six data sets, once $\tau_J < 0.7$, the % improvement plateaus, so we cap the figure at $\tau_J = 0.5$. We attribute this to the default setting that has a maximum sequence length of 16, which limits the maximum number of transformations that can be applied to s_u.

We varied model performance τ_M between $[0\%, 5\%]$ (Figure 5). $\tau_M = 0\%$ means the $acc(D_{OUT}^{s_u})$ and $acc(D_{OUT}^{\hat{s}_u})$ are identical. In general, on all six datasets, as the constraint becomes more relaxed, LS can make more improvements to script standardization. We observed a non-monotonic trend in House, which could be attributed to the complex interaction between $acc(D_{OUT})$ and D_{OUT}. Although $acc(D_{OUT})$ is influenced by D_{OUT}, the correlation is complex. Some small changes in the content of D_{OUT} may yield a significant improvement or decrease in $acc(D_{OUT})$.

We would expect that when the system has more freedom to change the script – that is, the constraint on the original intent is relaxed – we would see highly standardized output scripts. In contrast, when the system is limited to producing an output that is extremely fidelity to the user's original script, it has little flexibility, and we would expect to see output scripts that cannot be standardized very much. That is in fact what we see in Figures 5. As we relax the constraints on user intent, LS indeed usually produces output scripts that are more standardized.

6.3.3 Script standardization using various corpus settings. We consider three corpus scenarios to examine robustness (Table 5).

Summary: Overall, LS managed to find common transformations that improve script standardness, given a variety of corpora.

Results: We experimented with the following three scenarios.

Using a small corpus. We consider the scenario where only a limited number of scripts are available. We sampled 10 scripts from each dataset. Overall, LS maintained at least a 13% improvement in all datasets for both user-intent constraints, except Sales. We observed the most performance reduction on Titanic, House, and Medical. This is because their original corpus size is larger (Table 3) compared to the other datasets, providing a richer search space and potential opportunities to improve standardization.

Using a corpus of a different dataset that has a similar schema. We consider the scenario where a corpus for the identical dataset is not available, so the user resorts to a corpus on a similar dataset. We experimented with improving Spaceship scripts using Titanic corpus. LS still achieves an 11% improvement. This is because the competition setup and D_{IN} for Spaceship and Titanic share many similarities: they both predict survival, and there are identical column names in D_{IN}. This shows that LS can be used successfully by applying one script corpus to a different target, although — unsurprisingly — the quality improvement is reduced.

Using a low-ranked corpus. We consider the scenario where the corpus consists solely of low-ranked scripts. Specifically, the corpus uses the bottom 30% of the scripts, determined by their count of votes received on Kaggle. Despite being challenged by a smaller, low-ranked corpus, LS still managed to find common transformations that improve script standardness by 5%. Our takeaway here is that when only low-quality scripts are available, it is more effective to use scripts that process a similar dataset, as this approach leads to better system performance (Table 5).

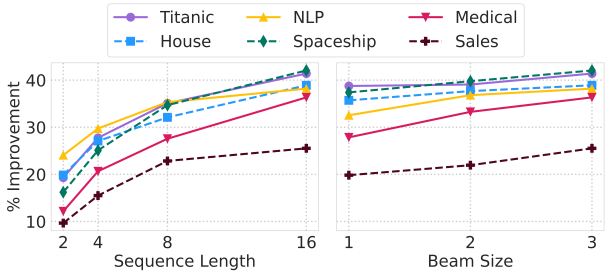

Figure 6: Median % improvement of LS-default with varied sequence lengths (left) and varied beam sizes (right).

6.4 Ablation Studies

We isolate each piece of our algorithm and evaluate their contribution to the overall effectiveness of LS.

6.4.1 Varied sequence lengths. Sequence length is the maximum number of transformations in a sequence. We use it as the stopping criterion. We set the sequence length *seq* to $\{2, 4, 8, 16\}$.

Summary: LS makes scripts more standardized as the maximum sequence length increases.

Results: As Figure 6 shows, on the six datasets, the median % improvement increases as the maximum number of sequence length. As *seq* increases, LS improves standardization at a faster rate at the start, and then the improvement slowly starts to plateau from $seq = 8$ to $seq = 16$. This could be attributed to the size of the script corpus and the diversity of atoms and edges (Table 3). We observed the best performance at $seq = 16$ on Titanic and Spaceship, which are two datasets with the most unique atoms and edges, with a % improvement of 41.4% and 42% respectively.

6.4.2 Varied beam sizes. The number of beams K in $\{1, 2, 3\}$ is how many in-progress sequences LS retains.

Summary: LS performs better as K increases.

Results: Figure 6 shows that in all six data sets, the beam search enables LS to increase the % improvement steadily as K increases. We observe that the beam search effect is the strongest on Medical, achieving an increase of 8.5% from $K = 1$ to $K = 3$, and the weakest on Titanic, obtaining an increase of 2.6%. On average, the beam search increases the % improvement by 5% across the six datasets.

6.5 Evaluating Efficiency

We isolate each component and discuss factors that could affect search performance and end-to-end latency.

On average, LS takes several minutes to generate a standardized version of a given input script (on average, one script takes 7.2 minutes end-to-end). Although this process may take a few minutes, it involves generating an entire script instead of just a single-step recommendation. This time frame is typical and comparable to other systems that generate full pipelines hands-free [10, 50, 51, 58, 65]. Future work will focus on reducing the search space, possibly by grouping semantically similar operations. See Section 8 for further discussion.

Summary: In general, the latency of the search steps (GetSteps(), GetTopKBeams()) increases as unique atoms and edges increase. The size of the data file D_{IN} also affects the latency of the checking steps (CheckIfExecutes(), VerifyConstraints()).

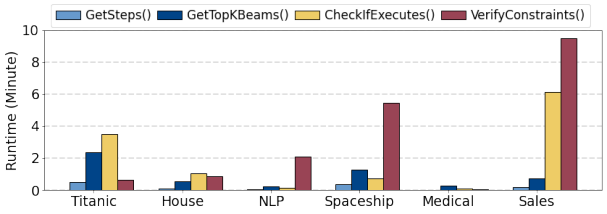

Figure 7: Median runtime breakdown on $seq = 16$.

Results: As shown in Figure 7, LS has the lowest latency on Medical and NLP, which are the two datasets with the lowest number of atoms and edges. The small number of atoms and edges implies a more narrow search space, which makes LS faster to find the potential next steps and converge to a standardized script. The type of ML models the users choose mainly has an impact on the latency of VerifyConstraint() since that is where our framework checks for the user-intent constraint, which includes model performance. This is the main reason why NLP has a longer latency than Medical. The size of the data file affects the latency from the perspective of constraint verification. The original latency is 20x slower on Sales compared to the other datasets before sampling is applied. This is because Sales has 744k tuples in D_{IN}, which is 30x more than NLP, the second largest D_{IN}. LS checks for the constraints by running the scripts that inevitably load D_{IN}. The larger D_{IN}, the more constraint checking contributes to the latency. We can sufficiently reduce the latency caused by large datasets with sampling.

Overall, the latency is influenced by both the algorithmic components of LS and the characteristics of the dataset. The median latency can be less than a minute per script in Medical and can take up to 17.5 minutes on Sales. We also note that the current latency depends on this specific version of the system prototype, which could be improved with more engineering efforts. For example, we can use parallelism, running each beam on a separate thread. In general, data preparation is a time-consuming task that is challenging even for experienced data scientists [21, 22]. In contrast, we offer a unique and effective solution that can significantly standardize data preparation scripts that require minimum effort from data consumers regardless of their backgrounds.

6.6 Case Study: Target Leakage Detection

Script standardization is the process of simultaneously removing out-of-the-ordinary steps from the input script and adding the missing common practices to the input script. Target leakage is another example of an out-of-the-ordinary step that should be replaced with one that is more conventional and correct. As a result, our system can potentially identify it (Figure 8).

6.6.1 Study setup. We performed a qualitative case study.

Target leakage. As one of the common data preparation mistakes in practice, target leakage occurs when the target variable or information derived from it is included in the feature set. This leads to overly optimistic performance estimates during model evaluation, as the model has access to information that would not be available in a real-world scenario. For each dataset, we sample 10% of the real-world scripts and use GPT-4 [52] to programmatically inject target leakage code snippets (ground truth). Prompts used are in [2].

Performance metric. We evaluate the effectiveness using the accuracy of detecting the ground truth snippets. Specifically, an output script is correct when it satisfies the constraints (Definition 4.5), and the ground truth snippet is labeled as to be removed.

```
1  import pandas as pd
2  df = pd.read_csv('diabetes.csv')
3  df = df.fillna(df.median())
4  df = df[df["Pregnancies"] < 10]
5  df = df[df['Age'].between(18,25)]
6  df = pd.get_dummies(df)
7  df['Outcome_copy'] = df['Outcome']
8  update = df.sample(20).index
9  df.loc[update, 'Outcome_dup'] = 0
10 X = df.drop(['Outcome'], axis=1)]
11 y_train = df['Outcome']
```

```
1  import pandas as pd
2  df = pd.read_csv('diabetes.csv')
3  df = df.fillna(df.median())
4  df = df[df["Pregnancies"] < 10]
5  df = df[df['Age'].between(18,25)]
6  df = pd.get_dummies(df)
7  df['Outcome']
8  update = df.sample(20).index
9  df.loc[update, 'Outcome_dup'] = 0
10 X = df.drop(['Outcome'], axis=1)]
11 y_train = df['Outcome']
```

Figure 8: Example input s_u (left); output scripts $\hat{s}_u$ (right). s_u has created a column correlated to the target with random noises (line 7-9). Green shows the added steps, while red shows the steps recognized to be removed from s_u.

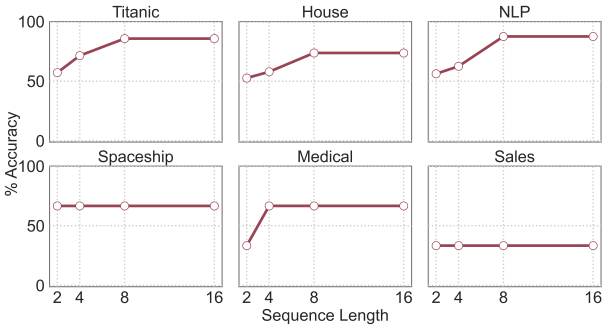

Figure 9: Accuracy of LS for target leakage detection.

6.6.2 Study results. We vary sequence lengths to study how many steps would LS take to detect the ground truth snippets. Figure 9 shows the effectiveness of our approach in target leakage detection, where over 66% of ground truth snippets are discovered within 8 steps for all datasets except Sales.

7 RELATED WORK

One-Step Recommendations: Automating data preparation pipelines can be achieved by recommending the next best data preparation step [8, 33, 34, 36, 63]. These solutions utilize AI and crowd-sourcing [34], web data [8], available data science notebooks [63], and learning models [63] to generate recommendations and predict user intent. Close to our work, the authors of [63] proposed Auto-Suggest [63], which "learns" data preparation steps from data scientist workflows and recommends the next step by predicting the next operator and its parameters. Our work is distinct in that it seeks to *standardize* an input script as much as possible while preserving user intent, rather than simplifying the task of analysts by predicting their next steps.

Multi-Step Recommendations: Researchers have recently explored multi-step automation for data preparation pipelines. Our work is the most similar to Auto-Tables [44] and Learn2Clean [13, 14]. Auto-Tables uses machine learning models to predict a sequence of table structural transformations for a given input table. Learn2Clean [13, 14] employs reinforcement learning to explore all possible strategies to process a given dataset to maximize the performance of a specific machine learning model. Although multi-step recommendations can be an application of our work, fundamentally, we solve a different problem. Instead of recommending a sequence of data preparation steps for a given user

intent in [13, 14, 64], we take the user's input script merely as a sketch and tweak the semantics within a threshold to improve script standardness.

Program Synthesis: Program synthesis is a technique that seeks to automatically generate code based on user intent, which can be specified by various forms of constraints such as I/O examples. Studies have demonstrated that several components of data preparation can be automated using programming by example, including spreadsheet automation [19], code generation in R [24], and SQL query generation [61]. Closer to our work, AutoPandas [11] takes input/output examples that demonstrate the desired transformation and uses a neural-backed synthesis engine to generate Python programs. Recently, the use of large language models for program synthesis has become increasingly popular. OpenAI Codex [18], a GPT [16] language model finedtuned on public Python code on GitHub, shows its capabilities in code writing by generating programs using docstrings. It can also be used to power many downstream program synthesis applications, such as GitHub Copilot [3], which suggests code and functions in real time to save programmers' time on repetitive code. The key difference between our problem and this line of work is that in program synthesis, the semantics and functionality of the program should not deviate from user intent, while we use user intent as guidance and allow it to be refined. This flexibility enables analysts to explore and integrate the collective knowledge embedded in the script collection. We evaluated the GPT models [6, 52] on the script standardization problem, without putting constraints on the intent of the user.

Code style transfer Learning-based approaches often analyze common coding styles found in codebases by converting code into representations like one-hot token embeddings or parse trees. They then derive formatting conventions and generate explicit [9, 46] or abstract [53] rules. Another recent system, DUETCS [17], aims to facilitate automatic code style transfer. However, these methods typically require the user to pre-define the coding style or focus solely on formatting issues. In contrast, our system automatically extracts abstract styles from the corpus. The goal is to preserve user intent by modifying the script to align with the most prevalent (i.e., standard) style in the corpus.

Automated Machine Learning (AutoML): This line of work focuses on automated processes that use machine learning algorithms and techniques to build, optimize, and select predictive models for a given dataset. AutoML is used to automate the process of training and tuning machine learning models, allowing users to quickly and easily generate more accurate models with less effort and time [35, 60]. Tools such as Auto-sklearn [25, 26], Auto-WEKA [60], and DeepLine [35] focus on automating machine learning model building. These systems tackle the challenge of automatic selection and optimization of machine learning models and their hyperparameters, and combine multiple types of machine learning model into a single model [35]. We are situated upstream in the machine learning model-building pipeline, helping analysts prepare their datasets before feeding them into machine learning models.

8 CONCLUSION AND LIMITATIONS

We introduce a new problem and propose an efficient framework to standardize data preparation scripts with minimal effort. When interpreting the generated script, several limitations should be considered. First, its quality may be influenced by factors such as the quality of the input script and the user intent threshold. Second, the generated script may not be fully ready for production, so human validation is necessary to ensure that it performs as intended and that no unintended biases are introduced [30].

Our framework currently supports scripts written in Python. In principle, it can be generalized to other programming languages. However, this would require extending the script-to-DAG and DAG-to-script procedures, as well as the lemmatization techniques used. We note that other systems on data preparation recommendations have also focused on Python [63, 64].

Standardizing data preparation scripts is challenging due to the diverse ways in which the same operations can be expressed across different frameworks (e.g., Pandas, PySpark, Modin, DuckDB). To accurately measure the similarity between various operations, it is crucial to use a high-level abstraction or knowledge base that ignores specific details such as the frameworks or their versions. Our current implementation uses simple NLP solutions [54]. Although we recognize existing work in this area and suggest that LLMs may be effective, our system does not yet incorporate these advanced approaches. This highlights a key opportunity for extending our framework. However, despite using LLM-based solutions as baselines, which is expected to perform well in this context, our framework demonstrated superior performance. With the rapid improvement of LLMs, especially in code generation, we expect to see improvements in their performance. We plan to integrate LLMs into our system to automatically verify that constraints are satisfied without explicitly checking them (i.e., ensuring that the code is executable). We also plan to rerun the experiments with new versions of LLMs in the future to track their improvement.

Another limitation is that the user must provide a script corpus. We note that similar assumptions are made by other frameworks [17, 63]. In addition, the dataset may be updated, or different datasets that have the same schema become available (e.g., MIMIC datasets [37, 43]). As we showed, even a small corpus or one containing scripts that process similar datasets can still yield valuable results. To address situations where obtaining a script corpus is challenging, one potential solution is to generate scripts using LLMs. Additionally, scripts authored by domain experts could be weighted differently (e.g., using the vote counts of Kaggle scripts).

The user also sets a threshold that indicates how much they are willing to compromise on their intent in exchange for a more standardized script. Our experiments show that the modified script not only stays within this threshold, but also, in many cases, improves model performance. Along with the user-intent threshold, our framework provides a set of parameters for the user to customize. While the parameters offer flexibility, they may require effort to tune. A possible extension to this work is an algorithm that optimizes configurations, such as exploring user intent thresholds and returning the Pareto curve.

Our framework currently supports two methods for evaluating how well user intent is preserved: table Jaccard and model performance. Future work will focus on incorporating additional metrics, such as assessing the semantic similarity between scripts, comparing their bags of operations, or evaluating the fairness of the model [30]. Lastly, an interesting direction for further exploration that we intend to pursue is providing explanations for why a transformation is recommended to the input script. The explanation would inform the user about the frequency of this operation in the corpus, its impact on the user intent, and the rationale behind it, such as scaling the features to improve the performance of a learning model.

REFERENCES

[1] ASTs – abstract syntax trees. https://docs.python.org/3/library/ast.html. Accessed: 2023-01-30.

[2] Bottom-up standardization for data preparation: full technical report, code, and results for review. https://github.com/ey-l/bottom-up-script-standardization.

[3] GitHub Copilot your ai pair programmer. https://github.com/features/copilot. Accessed: 2023-01-30.

[4] Kaggle API. https://github.com/Kaggle/kaggle-api. Accessed: 2023-01-30.

[5] Kaggle pima indians diabetes database. https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database. Accessed: 2023-01-30.

[6] OpenAI introducing chatgpt. https://openai.com/blog/chatgpt. Accessed: 2023-01-30.

[7] Sourcery automatically improve python code quality. https://sourcery.ai/. Accessed: 2023-01-30.

[8] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: A robust transformation discovery system. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1134–1145, 2016.

[9] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, pages 281–293, 2014.

[10] S. Amer-Yahia, T. Milo, and B. Youngmann. Exploring ratings in subjective databases. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 62âĂŞ75. Association for Computing Machinery, 2021.

[11] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica. Autopandas: Neural-backed generators for program synthesis. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[12] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[13] L. Berti-Equille. Learn2clean: Optimizing the sequence of tasks for web data preparation. In *The World Wide Web Conference*, WWW '19, page 2580âĂŞ2586, New York, NY, USA, 2019. Association for Computing Machinery.

[14] L. Berti-Equille. Reinforcement learning for data preparation with active reward learning. In S. El Yacoubi, F. Bagnoli, and G. Pacini, editors, *Internet Science*, pages 121–132, Cham, 2019. Springer International Publishing.

[15] R. Botvinik-Nezer, F. Holzmeister, C. Camerer, et al. Variability in the analysis of a single neuroimaging dataset by many teams. pages 84–88, 2020.

[16] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020.

[17] B. Chen and Z. Abedjan. Duetcs: Code style transfer through generation and retrieval. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2362–2373. IEEE, 2023.

[18] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code, 2021.

[19] X. Chen, P. Maniatis, R. Singh, C. Sutton, H. Dai, M. Lin, and D. Zhou. Spreadsheetcoder: Formula prediction from semi-structured context, 2021.

[20] P. W. Code. Papers with code. https://paperswithcode.com/, 2015. Accessed: April 12, 2023.

[21] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 240âĂŞ251, New York, NY, USA, 2002. Association for Computing Machinery.

[22] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, S. Madden, M. Ouzzani, and N. Tang. The data civilizer system.

[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[24] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning, 2017.

[25] M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer, and F. Hutter. Auto-sklearn 2.0: Hands-free automl via meta-learning. 2020.

[26] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[27] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. Ajax: An extensible data cleaning tool. *SIGMOD Rec.*, 29(2):590, may 2000.

[28] A. Gelman and E. Loken. The garden of forking paths: Why multiple comparisons can be a problem, even when there is no. *Fishing ExpeditionầĂ Ĺ or ầĂIJp-hackingầĂ Ĺ and the Research Hypothesis was Posited ahead of Time.ầĂ Ĺ Technical report, Department of Statistics, Columbia University*, 2013.

[29] S. Grafberger, J. Stoyanovich, and S. Schelter. Lightweight inspection of data preprocessing in native machine learning pipelines. In *Conference on Innovative Data Systems Research (CIDR)*, 2021.

[30] S. Guha, F. A. Khan, J. Stoyanovich, and S. Schelter. Automated data cleaning can hurt fairness in machine learning-based decision making. *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[31] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. *ACM SIGPLAN Notices*, 46(6):317–328, 2011.

[32] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[33] Y. He, K. Ganjam, K. Lee, Y. Wang, V. Narasayya, S. Chaudhuri, X. Chu, and Y. Zheng. Transform-data-by-example (tde): Extensible data transformation in excel. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1785âĂŞ1788, New York, NY, USA, 2018. Association for Computing Machinery.

[34] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR*, 2015.

[35] Y. Heffetz, R. Vainshtein, G. Katz, and L. Rokach. Deepline: Automl tool for pipelines generation using deep reinforcement learning and hierarchical actions filtering. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 2103âĂŞ2113, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Z. Jin, Y. He, and S. Chauduri. Auto-transform: Learning-to-transform by patterns. *Proc. VLDB Endow.*, 13(12):2368âĂŞ2381, jul 2020.

[37] A. E. Johnson, T. J. Pollard, L. Shen, L.-w. H. Lehman, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. Anthony Celi, and R. G. Mark. Mimic-iii, a freely accessible critical care database. *Scientific data*, 3(1):1–9, 2016.

[38] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the sigchi conference on human factors in computing systems*, pages 3363–3372, 2011.

[39] S. Kandel, R. Parikh, A. Paepcke, J. M. Hellerstein, and J. Heer. Profiler: Integrated statistical analysis and visualization for data quality assessment. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, AVI '12, page 547âĂŞ554, New York, NY, USA, 2012. Association for Computing Machinery.

[40] K. Kennedy. Use-definition chains with applications. *Computer Languages*, 3(3):163–179, 1978.

[41] S. Kullback and R. A. Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[42] E. Lai, Y. Lou, B. Youngmann, and M. Cafarella. Lucidscript: Bottom-up standardization for data preparation. *Proceedings of the VLDB Endowment (PVLDB)*, 17(12):4317âĂŞ4320, 2024.

[43] J. Lee, D. J. Scott, M. Villarroel, G. D. Clifford, M. Saeed, and R. G. Mark. Open-access mimic-ii database for intensive care research. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 8315–8318, 2011.

[44] P. Li, Y. He, C. Yan, Y. Wang, and S. Chaudhuri. Auto-tables: Relationalize tables without using examples. *ACM SIGMOD Record*, 53(1):76–85, 2024.

[45] W. Li, H. Yin, Y. Chen, Q. Liu, Y. Wang, D. Qiu, H. Ma, and Q. Geng. Associations between adult triceps skinfold thickness and all-cause, cardiovascular and cerebrovascular mortality in nhanes 1999–2010: a retrospective national study. *Frontiers in cardiovascular medicine*, 9:858994, 2022.

[46] V. Markovtsev, W. Long, H. Mougard, K. Slavnov, and E. Bulychev. Style-analyzer: fixing code style inconsistencies with interpretable unsupervised algorithms. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 468–478. IEEE, 2019.

[47] A. H. Murphy. The finley affair: A signal event in the history of forecast verification. *Weather and Forecasting*, 11(1):3 – 20, 1996.

[48] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, Y. Wu, Y. Zhu, and M. Weimer. Vamsa: Automated provenance tracking in data science scripts. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1542–1551, 2020.

[49] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[50] E. Oikarinen, K. Puolamäki, S. Khoshrou, and M. Pechenizkiy. Supervised human-guided data exploration. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 85–101. Springer, 2019.

[51] B. Omidvar-Tehrani, A. Personnaz, and S. Amer-Yahia. Guided text-based item exploration. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 3410–3420, 2022.

[52] OpenAI. Gpt-4 technical report, 2023.

[53] T. Parr and J. Vinju. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 137–151, 2016.

[54] J. Plisson, N. Lavrac, D. Mladenic, et al. A rule based approach to word lemmatization. In *Proceedings of IS*, volume 3, pages 83–86, 2004.

[55] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2019.

[56] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, volume 1, pages 381–390, 2001.

[57] C. Scaffidi, B. Myers, and M. Shaw. Intelligently creating and recommending reusable reformatting rules. In *Proceedings of the 14th International Conference*

on Intelligent User Interfaces, IUI '09, page 297âĂŞ306, New York, NY, USA, 2009. Association for Computing Machinery.

[58] M. Seleznova, B. Omidvar-Tehrani, S. Amer-Yahia, and E. Simon. Guided exploration of user groups. *Proceedings of the VLDB Endowment (PVLDB)*, 13(9):1469–1482, 2020.

[59] J. P. Simmons, L. D. Nelson, and U. Simonsohn. False-positive psychology: Undisclosed flexibility in data collection and analysis allows presenting anything as significant. *Psychological science*, 22(11):1359–1366, 2011.

[60] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms, 2012.

[61] C. Wang, A. Cheung, and R. Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 452âĂŞ466, New York, NY, USA, 2017. Association for Computing Machinery.

[62] J. M. Wicherts, C. L. Veldkamp, H. E. Augusteijn, M. Bakker, R. Van Aert, and M. A. Van Assen. Degrees of freedom in planning, running, analyzing, and reporting psychological studies: A checklist to avoid p-hacking. *Frontiers in psychology*, page 1832, 2016.

[63] C. Yan and Y. He. 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*, SIGMOD '20, page 1539âĂŞ1554, New York, NY, USA, 2020. Association for Computing Machinery.

[64] J. Yang, Y. He, and S. Chaudhuri. Auto-pipeline: Synthesizing complex data pipelines by-target using reinforcement learning and search, 2021.

[65] B. Youngmann, S. Amer-Yahia, and A. Personnaz. Guided exploration of data summaries. *Proceedings of the VLDB Endowment (PVLDB)*, 15(9):1798–1807, 2022.

[66] M.-C. Yuen, I. King, and K.-S. Leung. A survey of crowdsourcing systems. In *2011 IEEE third international conference on privacy, security, risk and trust and 2011 IEEE third international conference on social computing*, pages 766–773. IEEE, 2011.

[67] A. X. Zhang, M. Muller, and D. Wang. How do data science workers collaborate? roles, workflows, and tools. *Proc. ACM Hum.-Comput. Interact.*, 4(CSCW1), may 2020.

[68] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.