LucidScript: Bottom-up Standardization for Data Preparation

Eugenie Lai
MIT
eylai@mit.edu

Yuze Lou
University of Michigan
yuzelou@umich.edu

Brit Youngmann
Technion
brity@technion.ac.il

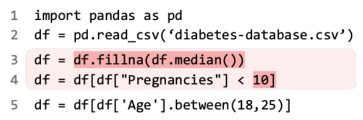
Michael Cafarella
MIT
michjc@csail.mit.edu

ABSTRACT

Data preparation is an essential step in every data-related effort, from scientific projects in academia to data-driven decision-making in industry. Typically, data preparation is not an interesting piece of a project — it transforms raw data into a format that enables further innovative work. Because such scripts are never intended to be interesting, are project-specific, and are written in general-purpose languages, they can be tedious to understand and difficult to verify. As a result, data preparation scripts can easily become a breeding ground for poor engineering and statistical practices. Ideally, data preparation scripts are "admirably boring" — they should serve the project, but otherwise be as simple and as standard as possible. We propose a bottom-up script standardization framework that takes a user's data preparation script and transforms it into a simpler, more standardized version of itself. Our framework takes the user's script not as an unchangeable definition of correctness, but as a sketch of the user's intent. We embedded this framework in a system called LucidScript.

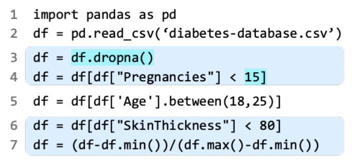
PVLDB Reference Format:
Eugenie Lai, Yuze Lou, Brit Youngmann, and Michael Cafarella.
LucidScript: Bottom-up Standardization for Data Preparation. PVLDB, 17(12): 4317 - 4320, 2024.
doi:10.14778/3685800.3685864

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ey-l/bottom-up-script-standardization.



```
1  import pandas as pd
2  df = pd.read_csv('diabetes-database.csv')
3  df = df.fillna(df.median())
4  df = df[df["Pregnancies"] < 10]
5  df = df[df['Age'].between(18,25)]
```

(a) Red shows the removed steps.



```
1  import pandas as pd
2  df = pd.read_csv('diabetes-database.csv')
3  df = df.dropna()
4  df = df[df["Pregnancies"] < 15]
5  df = df[df['Age'].between(18,25)]
6  df = df[df["SkinThickness"] < 80]
7  df = (df-df.min())/(df.max()-df.min())
```

(b) Blue shows added steps that are commonly used in script corpus.

Figure 1: Input in Example 1 (a); output in Example 2 (b).

1 INTRODUCTION

Data preparation is pivotal across various domains, from academic research to corporate decision-making. Custom data preparation programs are often essential parts of data pipelines, transforming raw data into a usable format for further innovation. However, data preparation is frequently overlooked, with analysts rarely detailing the process or its significance. These "uninteresting" data preparation scripts serve solely to support downstream innovative efforts. Unfortunately, these qualities make these scripts dangerous "attractive nuisances" from the perspective of reproducibility and reliability. Being project-specific and written in general-purpose languages, they are difficult for others to comprehend without substantial effort. Consequently, data preparation scripts can foster poor engineering and statistical practices. For example, variability in data preparation has been identified as a major contribution to the high rate of false positive results in biomedical research [7], despite the widespread use of standard datasets.

We argue that data preparation scripts should be *admirably boring*. The ideal data preparation script should perform its task without surprises, striving for standardization and predictability. Efforts within our community to standardize data preparation have largely taken a top-down approach, proposing new rules or paradigms [13, 15]. However, given the project- or data-specific knowledge involved, analysts continue to rely on general-purpose languages like Python. To this end, we developed a *bottom-up script standardization system* called LucidScript, in which the user inputs a sketch script and gets a modified script. This output script aims to obtain the same overall goal as the input script but is simpler and more standard. The following example illustrates how bottom-up standardization facilitates data preparation for analysts:

Example 1. Alex, a data scientist assisting a medical research team, is crafting a data preparation script (Figure 1a) to prepare a patient dataset for training a diabetes prediction model for young adult women. She explores numerous data preparation scripts on Kaggle from various medical projects. Initially, Alex addresses missing values and removes outliers based on her expertise (lines 3-4). Next, she filters patient records relevant to the objective (line 5). Alex tried to find useful insights from the historical scripts but

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685864

was discouraged by the volume and the required domain knowledge to understand them. Unfortunately, Alex's script implements data-cleaning rules (lines 3-4) that are specific to diabetes in North America. When the script is applied to a region with different patient demographics, these rules will yield a misleading dataset. □

In an ideal scenario, Alex would leverage historical scripts to enhance her own by identifying standardized processes used in other scripts, requiring minimal customization to align with her modeling objective. Consider Alex using LucidScript:

Example 2. Once again, Alex is writing a script from scratch. She puts down some data preparation steps based on her knowledge. Then she configures LucidScript to obtain code changes that yield only a modest change to the downstream model's performance and runs the system. LucidScript responds with a standardized script (Figure 1b). It identifies the uncommon transformations in Alex's sketch and replaces them with lines 3-4, which are applicable to more countries, and thus more commonly seen in relevant data processing scripts. The script selects only the relevant records (line 5) since Alex provides a threshold that indicates her intent. LucidScript also augments the script with lines 6-7, which frequently appear in historical scripts. Alex sees her script now detects outliers and normalizes numerical features, which were steps overseen previously. She is pleased as her prediction accuracy has increased due to applying the additional transformations. □

LucidScript takes an input draft script as a **sketch** of the user intent. It changes the semantics of the new program in service of two goals: its degree of "standardness" and its degree of "agreement with user intent". Intuitively, a script is more standard if it uses data preparation steps similar to the script corpus — a collection of historical data preparation scripts. Standardization is measured by the relative entropy of the data preparation step distribution between the input draft script and the script corpus.

For bottom-up script standardization, a search-based strategy is necessary to satisfy the multiple constraints we must place on a huge search space. But an exhaustive search is intractable, and even a greedy approach poses problems. We, therefore, developed an efficient framework that operates in two phases. In the offline phase, we calculate the corpus distribution and curate the search space. In the online phase, we employ a search algorithm to standardize the input script w.r.t. user intent. We developed multiple optimizations to explore sufficient search space while being efficient and satisfying the constraints. This work presents LucidScript usability and its suitability for end-to-end deployment.

Related Work. Automating data preparation pipelines can be achieved by recommending the next preparation step [12, 18]. Existing solutions often utilize available scripts, and learning models [18] to generate recommendations and predict user intent. Researchers have also explored multi-step automation for data preparation [5, 6, 19]. Although multi-step recommendations can be an application of LucidScript, our goal is different. Instead of recommending a sequence of steps for a given user intent, we take the user's input script merely as a sketch and tweak the semantics within a threshold to improve script standardness.

Program synthesis automates code generation based on user intent. Studies have shown that data preparation can be automated using programming by example techniques [4, 17]. Large language models like OpenAI Codex [8] have gained popularity for program synthesis. This technology powers applications like Github Copilot [2], which provides real-time code suggestions to save programmers' time. Our approach differs in that it allows user intent to guide the process, offering flexibility to refine it. This enables analysts to integrate collective knowledge embedded in the script corpus. In program synthesis, the semantics and functionality of the program should not deviate from user intent.

A related line of work is AutoML. AutoML is used to automate the process of training and tuning ML models, allowing users to easily generate accurate models with less effort. Tools such as Auto-sklearn [9], and DeepLine [11] focus on automating ML model selection, model building, and optimization. We are situated upstream in the ML building pipeline, helping analysts prepare their datasets before feeding them into ML models.

2 TECHNICAL BACKGROUND

2.1 Problem Formulation

A data preparation script is a sequence of lines of code that process some dataset for a downstream task, such as an ML model or visualizations. Let $\mathcal{S}=\{s_1,\ldots,s_n\}$ be a corpus of n scripts, and s_u is an input script provided by the user. We assume that all scripts in $\mathcal{S}$ as well as s_u process the same (or similar) dataset D_{IN}. Applying a script s to D_{IN} yields an output dataset D^{s}_{OUT}. LucidScript modifies s_u to yield a new script $\hat{s_u}$ s.t.: (1) Both scripts compute a **"similar"** result. (2) $\hat{s_u}$ is a more **conventional** program than s_u, according to statistics computed on the corpus $\mathcal{S}$.

Measuring Goal One: Output Similarity Finding the semantic similarity of two scripts is an extensively studied topic [20]. We measure whether two scripts are semantically 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. LucidScript supports various distance measures, including the Jaccard index, earth mover distance, and Jenson-Shannon distance, which all assess structured dataset outputs. It can also consider downstream application performance, suitable for tasks with measurable quality metrics like accuracy.

Measuring Goal Two: Standardization Our goal is to create a new version of the input script that is as standard and conventional as possible. We use *relative entropy* 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 distribution $Q(x)$. We adapt the notation of relative entropy to measure script standardness. Intuitively, $P(x)$ is the probability distribution of the preparation steps in script s_u, while $Q(x)$ is the probability distribution of the preparation steps in the corpus $\mathcal{S}$. A large distance between $P(x)$ and $Q(x)$ means that $s_u$uses many data preparation steps that are not commonly used in $\mathcal{S}$.

Problem Definition: The user provides the following: (1) A **script corpus** $\mathcal{S}$, (2) An **input script** s_u, (3) An **input dataset** D_{IN}, and (4) **User-intent parameters** $\Delta(D^{s_u}_{OUT}, D^{\hat{s}_u}_{OUT})\leq\tau$, where the user specifies a distance metric Δ and a threshold τ. LucidScript then outputs a new program $\hat{s_u}$ that maximizes the program standardization as measured by $RE(\hat{s_u}, \mathcal{S})$, while ensuring that the resulting program's output is executable and within τ of the original.

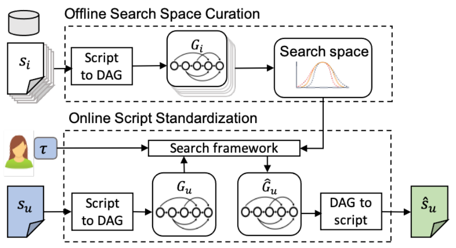


Figure 2: LucidScript Overview.

2.2 Script Representation

We considered several representations. Free text is less optimal since it does not recognize the syntactic constructs. Abstract syntax trees (AST) resolve the syntactic problems but do not explicitly indicate data flows and do not separate data variables and function operators. DAGs are another common script representation. Following previous work [10], we introduce a tailored DAG representation for our context. We face a multifold challenge: to represent an arbitrary script and explore legal changes efficiently. This requires a format that enables efficient computation of standardization statistics, enumeration of legal changes, and abstraction of non-essential syntactic differences. Bottom-up standardization also necessitates retaining sufficient information for translating the DAG back to a script. In our DAG representation, nodes represent operation invocations and edges represent data flows.

Next, we describe how to compute the probability of observing a particular script s. Derived from the DAG, each data preparation step is broken into atoms set A and the edges between atoms E. An atom $a \in A$ consists of one invocation node and its parents that are not an invocation node. We use atoms and edges to compute $RE(\hat{s_u}, \mathcal{S})$. Transformations are the actions we perform to change a DAG. We support two transformation types, `add` and `delete`. A transformation is defined by its type, what to change (i.e., an atom and its necessary edges), and where to change (i.e., line number). We use transformations to make changes to function invocations in a script by adding and deleting atoms from its DAG representation.

2.3 The LucidScript System

As depicted in Fig. 2, LucidScript comprises a UI (shown in Fig. 3) and two main components. The user submits an input sketch script s_u. In the offline phase, LucidScript processes the corpus to calculate the corpus distribution. In the online phase, it employs an optimized search algorithm to find a series of transformations for s_u. It then outputs the standardized, modified script $\hat{s_u}$ to the user.

Offline Phase. During the offline phase, we construct atom and edge vocabularies to obtain the corpus distribution from the scripts in the provided corpus. These components serve as inputs for the search framework in the online phase. To obtain the atom vocabulary $\mathcal{V}_A$ and edge vocabulary $\mathcal{V}_E$ from the corpus $\mathcal{S}$, we parse each script $s_i \in \mathcal{S}$ into its corresponding DAG, then compute the probability distribution $Q(x)$ from the edge vocabulary $\mathcal{V}_E$.

We address semantically equivalent data preparation steps by leveraging NLP techniques, static code lemmatization, research on semantic similarity assessment between scripts (e.g., [20]), and LLMs like GPT-4 [14], known for excelling in such tasks. These techniques help reduce vocabulary size and improve efficiency

Online Phase. The next goal is to find a sequence of transformations that standardizes the script s_u while satisfying the constraints. We first configure a set of candidate transformations. To generate all possible transformations, we enumerate the combination of which atom to change and where to apply the change.

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

To this end, we propose an optimized search algorithm that overcomes the limitations. Our optimizations enable LucidScript to find high-quality sequences of transformations efficiently. These optimizations are summarized as follows: **(1) Monotonicity:** A transformation sequence cannot go back and make changes to an earlier portion of the script. This ensures that once a script becomes non-executable, no further transformation would make it executable again; **(2) Sampling:** When the dataset D_{IN} is large, we apply random sampling on the tuples; **(3) Checking strategies for constraints:** We have two constraints: (a) For the execution constraint, we remove candidate sequences that lead to non-executable scripts after every transformation is applied. (b) We check the user-intent constraint at the end. These strategies ensure our approach always finds valid sequences while keeping runtime in check; **(4) Beam search:** We keep K beams rather than a single best, which retains multiple best options during the search; **(5) Diversity:** Beam search drawback is that the top-ranked next transformations can be similar. Thus, we prioritize sufficiently different transformations to explore diverse parts of the search space.

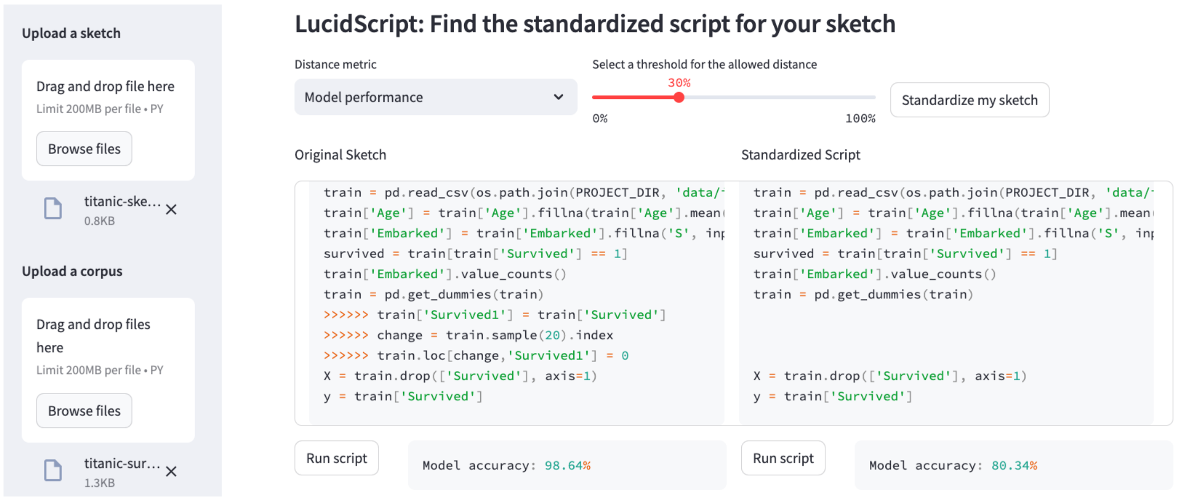
3 DEMONSTRATION

We implemented LucidScript using Python and Streamlit [16]. LucidScript was written in 3,000 lines of Python source code and currently supports straight-line Python scripts available at [1].

We demonstrate the operation of LucidScript over six datasets from various domains, which are associated with multiple scripts extracted from Kaggle. Statistics on the obtained corpora are given in Table 1: **Titanic**: predicting survival; **Sales**: forecasting future sales; **House**: predicting house prices; **NLP**: distinguishing real

Table 1: Examined datasets.

Statistics	Titanic	House	NLP	Spaceship	Medical	Sales
Scripts	62	49	24	38	47	26
Data files	3	4	3	3	1	6
Data tuples	2,613	4,381	744,302	22,701	17,250	769
Data features	25	163	11	29	9	18
Avg # code lines	64	43	19	44	30	39



Original Sketch:

```
train = pd.read_csv(os.path.join(PROJECT_DIR, 'data/
train['Age'] = train['Age'].fillna(train['Age'].mean
train['Embarked'] = train['Embarked'].fillna('S', inp
survived = train[train['Survived'] == 1]
train['Embarked'].value_counts()
train = pd.get_dummies(train)
>>>>>> train['Survived1'] = train['Survived']
>>>>>> change = train.sample(20).index
>>>>>> train.loc[change,'Survived1'] = 0
X = train.drop(['Survived'], axis=1)
y = train['Survived']
```

```
Model accuracy: 98.64%
```

Standardized Script:

```
train = pd.read_csv(os.path.join(PROJECT_DIR, 'data/
train['Age'] = train['Age'].fillna(train['Age'].mean
train['Embarked'] = train['Embarked'].fillna('S', inp
survived = train[train['Survived'] == 1]
train['Embarked'].value_counts()
train = pd.get_dummies(train)



X = train.drop(['Survived'], axis=1)
y = train['Survived']
```

```
Model accuracy: 80.34%
```

Figure 3: UI of LucidScript (example usability for anomalous step detection).

vs. fake Tweets; **Spaceship Titanic**: predicting passengers from a spaceship, and **Medical**: predicting diabetes in patients.

The participants will play the role of data analysts, attempting to improve their data preparation script. The demonstration begins by allowing attendees to select the dataset to investigate. The audience then engages with LucidScript via the following scenarios:

Script Standardization: To illustrate the usability of LucidScript and for the sake of demonstration, the audience will be exposed to an end-to-end operation of LucidScript. The participants will select a random script from the relevant corpus to be improved. The audience would then examine the script and its output (e.g., predication accuracy), and will be asked to set a threshold on the user intent parameter (see upper part of Fig. 3). By clicking on the `Standardize my sketch` button, LucidScript will generate an improved version of the script, highlighting the recommended transformations (see the lines marked by >>> or <<<). The participant will then run the modified script to ensure the output script is executable and maintains its initial intent. The audience will be invited to compare the output with scripts generated by existing solutions such as GPT [14] and Sourcery [3]. This comparison is intended to showcase that, while ML-based solutions are trained on numerous publicly available scripts, they lack the ability to standardize input scripts with a focus on domain-specific knowledge.

Anomalous Step Detection: Participants will explore LucidScript's versatility beyond script standardization, focusing on its capability to assist inspectors in identifying anomalous data preparation steps. Script standardization entails removing unconventional steps from the input script and incorporating missing common practices. Attendees will insert anomalous steps into a provided script (e.g., dropping columns/rows, randomly adding noise) and input it into LucidScript. Upon clicking the `Standardize my sketch` button, LucidScript will standardize the input by detecting and eliminating the anomalous steps. While the classification accuracy for the Titanic dataset declined (Figure 3), note that the deceptive steps involving the replication of the outcome variable `Survived` were excluded (the lines marked by >>>).

Looking under the hood: Our demonstration invites participants to explore LucidScript. This includes adjusting parameters such as corpus size and sequence lengths. LucidScript can generate high-quality scripts even with the presence of a small corpus size or, alternatively, is highly efficient even if the corpus contains hundreds of scripts, which showcases the robustness and effectiveness of LucidScript across diverse scenarios.

REFERENCES

[1] GitHub. https://github.com/ey-l/bottom-up-script-standardization.
[2] GitHub Copilot your ai pair programmer. https://github.com/features/copilot.
[3] Sourcery automatically improve python code quality. https://sourcery.ai/.
[4] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica. Autopandas: Neural-backed generators for program synthesis. *Proc. ACM Program. Lang.*, oct 2019.
[5] L. Berti-Equille. Learn2clean: Optimizing the sequence of tasks for web data preparation. In *WWW '19*, page 2580–2586, 2019.
[6] 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.
[7] 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.
[8] M. Chen et al. Evaluating large language models trained on code, 2021.
[9] M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer, and F. Hutter. Auto-sklearn 2.0: Hands-free automl via meta-learning. 2020.
[10] S. Grafberger, J. Stoyanovich, and S. Schelter. Lightweight inspection of data preprocessing in native machine learning pipelines. In *CIDR*, 2021.
[11] Y. Heffetz, R. Vainshtein, G. Katz, and L. Rokach. Deepline: Automl tool for pipelines generation using deep reinforcement learning and hierarchical actions filtering. In *KDD '20*, page 2103–2113, 2020.
[12] Z. Jin, Y. He, and S. Chauduri. Auto-transform: Learning-to-transform by patterns. *Proc. VLDB Endow.*, 13(12):2368–2381, jul 2020.
[13] S. Kandel, R. Parikh, A. Paepcke, J. M. Hellerstein, and J. Heer. Profiler: Integrated statistical analysis and visualization for data quality assessment. In *AVI '12*, 2012.
[14] OpenAI. Gpt-4 technical report, 2023.
[15] C. Scaffidi, B. Myers, and M. Shaw. Intelligently creating and recommending reusable reformatting rules. In *IUI '09*, page 297–306, 2009.
[16] streamlit. Streamlit. https://streamlit.io/, 2024. Accessed: April 12, 2024.
[17] C. Wang, A. Cheung, and R. Bodik. Synthesizing highly expressive sql queries from input-output examples. In *PLDI 2017*, page 452–466, 2017.
[18] C. Yan and Y. He. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *SIGMOD '20*, page 1539–1554, 2020.
[19] J. Yang, Y. He, and S. Chaudhuri. Auto-pipeline: Synthesizing complex data pipelines by-target using reinforcement learning and search, 2021.
[20] 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 *ICSE '19'*, pages 783–794.