PipeLens: Identifying Interventions for Resolving Malfunctioning Data Science Pipelines

Jahid Hasan
Purdue University
hasan89@purdue.edu

Stanley Jiang
Cornell University
sj466@cornell.edu

Tejendra Singh
Purdue University
sing1100@purdue.edu

Sainyam Galhotra
Cornell University
sg@cs.cornell.edu

Romila Pradhan
Purdue University
rpradhan@purdue.edu

Divesh Srivastava
AT&T Chief Data Office
divesh@research.att.com

ABSTRACT

Data is a critical component of modern decision-making systems; system malfunctions (e.g., performance degradation and module failure) can often be traced back to a mismatch between the properties of the data and the assumptions of the system modules that process the data. For example, with the increasing use of open-source libraries to develop data science pipelines, common causes of system malfunctions include inappropriately configured data processing libraries for data cleaning tasks such as entity resolution or missing value imputation. Our objective is to resolve malfunctioning pipelines and improve their utility; we introduce PipeLens, a framework that leverages successful and unsuccessful runs of past pipelines for fixing pipeline malfunctions. PipeLens uses an acyclic graph representation of the pipeline and performs causal reasoning through interventions: when a system malfunctions with a given dataset, PipeLens modifies the pipeline (by changing its structure or the parameters of its modules) and observes the impact of this intervention on system behavior. To focus on useful interventions, we learn a proxy function that approximates the pipeline's utility over a dataset and guides the search for the best intervention. Unlike traditional observational analysis that reports correlations between system parameters and their behavior, we provide causally verified root causes and suggest pipeline modifications that rectify malfunctions. Empirical evaluation on four data science tasks over four real-world datasets demonstrates that PipeLens consistently outperforms baselines in terms of interventions performed to repair malfunctions while maintaining practical running times.

PVLDB Reference Format:
Jahid Hasan, Stanley Jiang, Tejendra Singh, Sainyam Galhotra, Romila Pradhan, and Divesh Srivastava. PipeLens: Identifying Interventions for Resolving Malfunctioning Data Science Pipelines. PVLDB, 19(11): 3188 - 3201, 2026.
doi:10.14778/3836663.3836682

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jahid674/PipeLens.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836682

1 INTRODUCTION

The ubiquity of data science pipelines has made them indispensable for modern decision-making systems. The lifecycle of data — acquisition, curation, integration, transformation, modeling, and evaluation — is increasingly mediated by complex, end-to-end pipelines that translate raw, heterogeneous inputs into actionable knowledge. Data flows from one stage in the pipeline to the next, and therefore, the quality and characteristics of the input data play a crucial role in determining their behavior and performance [11, 36]. Variations in data over time can lead to unpredictable performance, particularly when the original assumptions of the pipeline implementation do not align with the properties of the incoming dataset [8, 20, 22, 26].

Examples of system malfunction due to a mismatched assumption include: (i) *Machine learning (ML) pipeline quality degradation due to selection bias* [7, 13, 15, 29]: an ML model often assumes that the training data reflects a true random sample of the underlying distribution. However, if the data excludes individuals from specific regions or demographics, it can introduce bias and degrade quality metrics, e.g., F1 score. (ii) *Data visualization pipeline assuming uniqueness of records:* the data integration component of the pipeline may assume that all incoming records are unique, but the presence of duplicates can lead to double-counting in generated visuals, resulting in misleading conclusions. Other common issues arising from a mismatch of assumptions include unfairness in decisions, excessive running times, pipeline crashes, etc. With the advent of advanced software paradigms, often termed Software 3.0, software libraries increasingly rely on pre-trained models, automated optimization, and self-correcting algorithms to improve system performance [41]. Although traditional code-level bugs remain relevant, system malfunctions are increasingly driven by data-related failures, including improper pipeline configuration, inefficient data-flow design, poorly defined stage dependencies, and inconsistent data formats [42, 66]. Consequently, malfunction resolution is shifting from code repair toward addressing data inconsistencies and optimizing pipeline configurations.

We describe the challenges of dealing with these issues and resolving malfunctioning pipelines with the following example.

Example 1.1 (E-commerce website: Entity matching pipeline). Imagine an e-commerce platform that aggregates books and novels from multiple vendors (Figure 1). Each vendor provides the title, author, year, and number of pages for each book. The e-commerce platform uses an entity matching pipeline consisting of two components – blocking and pair matching – to deduplicate products and list variations of the same entity together. The blocking component groups

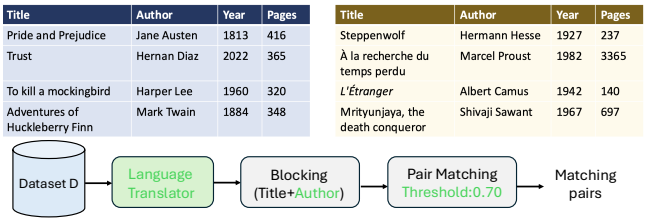


Figure 1: Example pipeline to perform entity matching; blue: passing dataset, yellow: failing dataset. Green components resolve pipeline malfunction on failing dataset.

books into candidate blocks based on shared title tokens; identified blocks are used to enumerate candidate pairs which are processed by the pair matching component to identify duplicates (labeling products whose descriptions have a Jaccard similarity higher than 0.83 as matches). The pipeline is evaluated by computing F1 score over output duplicates and ground truth; the dataset is considered *passing* if the score is more than the pre-defined threshold of 0.87. Suppose a new vendor starts listing novels in different languages (shown in the yellow colored dataset). With this new dataset, the original pipeline fails to identify several novels in different languages as duplicates and leads to an F1-score of 0.8, and the dataset is deemed as *failing*. Reasons for this failure include: (a) *language assumption:* because the pipeline assumed all product titles and descriptions to be in the same language, variants of a novel in different languages are not compared as their tokens do not overlap. (b) *blocking configuration:* since the blocking phase relies only on book titles, the translated titles may not match original English titles due to differences in phrasing or interpretation e.g., "À la recherche du temps perdu" translates to "In Search of Wasted Time," which may not match exactly with any existing English title (Remembrance of Things Past) in the dataset. (c) *low similarity scores:* the pairwise Jaccard similarity between the descriptions of non-English novels and their English counterparts is generally low because translations can vary significantly; hence, using a high similarity threshold (= 0.83) causes the pipeline to miss these matches.

This malfunction occurred even though the developer of the original pipeline had tested it extensively on the initial dataset, which only included English-language books. To resolve this malfunction, the following modifications and additions are necessary (changes are indicated in green): (i) insert a language detection and translation component; (ii) modify the blocking phase to use additional attribute tokens, such as author names, which are less likely to vary across translations; and (iii) adjust the similarity threshold. By incorporating these changes, the pipeline can effectively match international book listings with their English counterparts, ensuring accurate deduplication and aggregation of product information.

Limitations of existing tools. Prior data debugging techniques have primarily focused on identifying individual tuples or cell values [4] that are responsible for such malfunction. Recently, BugDoc [36] focused on adjusting pipeline parameters to resolve system malfunctions, but did not consider data transformations as a means to resolve such issues; therefore, it would not have identified that a new language translation component needs to be inserted into the pipeline. DataPrism [20] identified data-profile transformations to resolve system malfunctions. Data profiles are treated as explanation units, and the repair space is limited to input data transformations; therefore, it does not cater to pipeline-level repairs (e.g., inserting a new module, changing component strategies, reordering modules). Simultaneously, software debugging [45, 63, 67] focuses on failure localization to determine parts of an input program that cause failure. In contrast, data science pipelines are configurable workflows that include black-box components, where failures arise from interactions between data properties and pipeline configurations. The complex issues in Example 1.1 require not only data transformations or error localization but also modifications to the pipeline's structure and components, which are not adequately addressed by existing tools.

Given a pipeline that malfunctions on an input dataset D_{fail} and passes on some historical dataset D_{pass}, we aim to identify changes to the original pipeline that would resolve the malfunction on D_{fail}. Different from pipeline optimization, our goal is to identify a minimal intervention on the original pipeline (e.g., module insertion/deletion/reordering or parameter changes) that would satisfy the desired utility threshold on D_{fail}. Effectively traversing the search space of pipeline parameters and data transformations is the key challenge in efficiently resolving the malfunction. We identify several novel insights to address this problem. First, we recognize the causal dependencies between pipeline components and datasets, modeling the pipeline as a causal directed acyclic graph (DAG) where a component's parameters and its input data affect pipeline utility. Second, we use data profiles [2, 20] to identify dataset properties that are correlated with pipeline utility e.g., the impact of missing values on model accuracy. Data profiles help rank interventions by examining similarities in data distributions and their influence on utility. Third, we utilize execution history from the development environment to learn proxy functions (similar to a Bayesian optimization technique [47]) that guide how different interventions will impact pipeline utility, allowing us to prioritize effective changes and ignore irrelevant ones. Our proposed solution, PipeLens, integrates these insights to reduce the huge search space of potential modifications to resolve malfunctioning pipelines. Note that while the identified fix may (or may not) worsen the pipeline utility for the passing dataset on D_{pass}, PipeLens can also model additional requirements to ensure high utility for all datasets.

Solution sketch. At a high level, PipeLens first learns a proxy function to estimate the importance of feasible pipeline interventions on system utility (a general metric to capture performance, accuracy, fairness, or any other requirement). Then, PipeLens operates in two phases: *optimistic* search and *exhaustive* search. The optimistic phase tries to iteratively choose the best intervention (modifying the parameter of a component or modifying the pipeline structure) to improve the overall utility. If unsuccessful, the second phase explores the search space systematically, with the aim of quickly identifying the best intervention. We consider two different pipeline setups: (i) in the *opaque-box* setup, the intermediate output of each component is not accessible, but pipeline modifications are made by validating the effect of interventions on the final output. (ii) in the *glass-box* setup, the pipeline provides complete access to the intermediate outputs of each of its individual components.

Contributions. In this work, we make the following contributions:

- We study the novel problem of resolving malfunctioning data science pipelines through interventions. A pipeline is formalized as a directed acyclic graph, where each node denotes a pipeline component and edges capture data flow. This formulation is applicable across various domains and pipeline structures.
- We consider two scenarios to identify the best intervention that improves system utility: (i) opaque-box, where the internal mechanics of pipeline components are hidden, and (ii) glass-box, where component internals are accessible. Our approach leverages the causal relationship among pipeline parameters, data profiles, and their impact on utility.
- We evaluate PipeLens on several data science tasks over four real-world datasets and compare it with prior techniques. Compared with the baselines, PipeLens reaches the desired utility with fewer interventions, lower runtime, and effective scalability.

2 PRELIMINARIES

We formalize the notion of a data science pipeline and its constituent components (or *modules*), pipeline malfunction, and pipeline intervention to modify the pipeline. We then define the explanation (cause and corresponding fix) of pipeline malfunction and formulate the problem of modifying the pipeline to resolve the malfunction.

2.1 Data Processing Modules and Pipeline

A data-processing pipeline processes an input dataset $\mathbf{D}_{in}$ through a sequence of modules, such as data integration, normalization, cleaning, model training, and analysis. Each module applies a transformation to its input data using a selected configuration and produces an intermediate output that is consumed by downstream modules. Modules may also impose input and output specifications, such as required data types, attributes, or output formats. Formally,

Definition 1 (Module). *A pipeline module M is characterized by a tuple consisting of: (a) a list of parameters $\boldsymbol{\theta}$, (b) a transformation function f, (c) input specifications $\mathbf{I}$ denoting constraints that the input dataset must satisfy, and (d) output specifications $\mathbf{O}$ denoting constraints that the output dataset satisfies. Module M takes a dataset $\mathbf{D}_{in}$ as input and returns a dataset $\mathbf{D}_{out} = f(\boldsymbol{\theta}, \mathbf{D}_{in})$ whenever $\mathbf{D}_{in}$ satisfies the input specifications. Transformation function f maps the inputs $\boldsymbol{\theta}$ and $\mathbf{D}_{in}$ to the output $\mathbf{D}_{out}$ and guarantees that $\mathbf{D}_{out}$ satisfies the output specifications.*

We model a data science pipeline as a linear DAG, reflecting the structure of most data science pipelines [10]. The pipeline organizes a set of modules $\mathbf{M}$ as a data-flow graph, where each module $M \in \mathbf{M}$ is a node, and edges denote the flow of intermediate data. Each node stores information on the module type, transformation function, and its parameters. The edge set is initialized as $\mathbf{E} = \{(M_i, M_{i+1})\}_{i=1}^{k-1}$, such that an edge denotes the data flow i.e., intermediate output produced by module M_i and consumed by module M_{i+1}. We assume a global module collection $\mathbf{T}$ with $\mathbf{M} \subset \mathbf{T}$, and each module M has a finite strategy set S_M representing its available transformation functions.

Definition 2 (Data Processing Pipeline). *A data processing pipeline is an s-t graph $G(\mathbf{M}, \mathbf{E})$ consisting of the modules $\mathbf{M}$ as the nodes and directed edges $e \in \mathbf{E}$ between modules. The pipeline takes a collection of datasets $\mathbf{D}_{in}$ as input and returns an output $\mathbf{D}_{out}$ after processing $\mathbf{D}_{in}$ through the graph, i.e., $\mathbf{D}_{out} = G(\mathbf{M}, \mathbf{E})(\mathbf{D}_{in})$. Note that the input dataset $\mathbf{D}_{in}$ must satisfy the input specifications of the source node s in G.*

Figure 1 shows the DAG of an example pipeline with each node representing a module. In our implementation, module specifications are enforced via lightweight compatibility checks on the intermediate data object, including data type, required attributes, and output format, mostly using standard Python libraries (Table 2). For instance, numerical preprocessing modules require numerical attributes and return a tabular dataset, while entity-matching modules operate on textual attributes or candidate record pairs.

Pipeline Utility and Malfunction. We quantify the quality of a pipeline through its utility score and specify conditions for its malfunction as defined below:

Definition 3 (Utility Score and Pipeline Malfunction). *Given a pipeline $G(\mathbf{M}, \mathbf{E})$, the utility score of a dataset D over the pipeline G, denoted by $u_G(D) \in \mathbb{R}$, is a real value that quantifies the quality of the analysis performed by the pipeline over the input D. A pipeline G over dataset D is considered to malfunction if $u_G(D)$ is less than a desired utility threshold τ.*

The utility function is a user or application-defined scoring function over the pipeline output, analogous to an interestingness test for deciding whether the output satisfies the application goal. With ground truth, this score can be instantiated using standard supervised evaluation metrics, such as F1 score for entity matching, accuracy, precision, recall, or fairness metrics for classification, and mean squared error for regression [48]. In non-ML data-processing pipelines, utility can be defined through task-specific validation checks, such as schema-validity rate, constraint satisfaction, runtime, or aggregate error against trusted reference statistics [49]. For visualization pipelines, utility can measure visualization interestingness, such as deviation from a reference view as in SeeDB [62]. A dataset is *failing* if its utility falls below the desired threshold and *passing* otherwise; this thresholded-utility function is analogous to the pass/fail oracle commonly used in automated software debugging, which classifies inputs or executions as passing or failure-inducing [45, 67]. We assume access to historical executions consisting of pipelines and their corresponding utility scores.

2.2 Pipeline Intervention

We define an intervened pipeline G' as a modified version of G obtained through an *intervention*. A pipeline intervention consists of: (i) a **structural intervention**, which changes the pipeline structure, and (ii) a **parameter intervention**, which specifies modules' configuration. Concretely, interventions may replace specific nodes with alternative modules, encompassing operations such as parameter updates, module substitution, insertion, or swapping.

Definition 4 (Pipeline Intervention). *Given pipeline $G(\mathbf{M}, \mathbf{E})$, a* pipeline intervention *is a composite transformation*

$$I = \sigma \circ \Pi : G \longrightarrow G'(\mathbf{M}', \mathbf{E}')$$

that jointly modifies the structure and parameters of the pipeline.

(i) Structural Intervention. *$\Pi : \mathbf{M} \to \mathbf{M}'$ transforms the module sequence by inserting, deleting, replacing, swapping, or retaining*

modules, such that $\mathbf{M}' \subseteq \mathbf{T}$. *Specifically,* $\mathbf{M}' \subset \mathbf{M}$ *for deletion,* $\mathbf{M} \subset \mathbf{M}'$ *for insertion,* $\mathbf{M} = \mathbf{M}'$ *(same multiset, but different order) for swapping, and* $|\mathbf{M}'| = |\mathbf{M}|$ *where* $\exists i : \mathbf{M}'_i \neq \mathbf{M}_i$ *for replacing modules.*

After each structural change, the edge set $\mathbf{E}$ *is updated through the* Update-Edge *operation to maintain structural consistency. In the sequential pipelines, the DAG is implemented as an ordered list of executable modules with edges between consecutive modules.* Update-Edge *then rewires only the affected neighboring edges—connecting* $\mathbf{M}$ *between its neighbors for insertion or replacement, linking adjacent nodes for deletion, or exchanging connections for swapping—so that the resulting graph* $G(\mathbf{M}', \mathbf{E}')$ *preserves acyclicity and execution order.*

(ii) Parameter Intervention. *A mapping* $\sigma : \mathbf{M}' \rightarrow \bigcup_{M \in \mathbf{M}'} S_M$ *assigns parameter configurations to pipeline modules, where* S_M *denotes the parameter space associated with module* M.

Depending on the intervention type, the resulting pipeline $G' = I(G)$ encodes modifications to the structural topology, the parameter configuration, or a combination of both. The set of modified modules is defined as $\textsc{Modified}(I) = G \triangle G'$, i.e., the symmetric difference between the original and intervened pipelines.

Definition 5 (Minimal Explanation of Pipeline Malfunction). *Given a pipeline* G, *desired utility* τ, *module search space* $\mathbf{T}$, *and failing dataset* D_{fail}, *an intervention* I *is a minimal explanation of the pipeline malfunction if* $u_{I(G)}(D_{fail}) \geq \tau$ *and there exists no intervention* I' *such that* $\textsc{Modified}(I') \subset \textsc{Modified}(I)$ *and* $u_{I'(G)}(D_{fail}) \geq \tau$.

Note that the explanation of a malfunction aims to identify minimal interventions that improve the utility of D_{fail}; the same intervention may worsen the utility of another dataset, which may itself be either passing or failing under the original pipeline. A minimal explanation is a utility-driven repair set. Since utility may not capture all domain-specific correctness requirements, the returned intervention may be inspected through a human in the loop for correctness before deployment. The definition of a malfunction can be extended to ensure that the utility of a set of datasets should be higher than τ; for simplicity, we focus on a single failing dataset.

2.3 Problem Setting

Given a pipeline G, we define the pipeline repair discovery problem that aims to identify a minimal set of interventions that improves the overall utility.

Problem 1 (Discovering Pipeline Repair). *Given a malfunctioning pipeline* G *on a failing dataset* D_{fail} *such that* $u_G(D_{fail}) < \tau$, *identify a minimal explanation of pipeline malfunction* I *such that* $u_{G'}(D_{fail}) \geq \tau$ *where* $G'(\mathbf{M}', \mathbf{E}') = I(G)$.

We consider two extreme settings that capture varied visibility into the pipeline internals: opaque-box and glass-box. While the glass-box setting provides access to intermediate outputs or data after each pipeline stage, the opaque-box setting provides no additional information about the pipeline structure.

3 PIPELENS ALGORITHM

We now describe our algorithm PipeLens to identify the best intervention that explains the system malfunction on D_{fail}. We go through an example scenario in Section 3.1, provide key insights in Section 3.2 and the algorithm details in Section 3.3 and 3.4.

Race	Gender	Age	Type	Income	Lien status	Action taken
White	Male	35-44	Conventional	28,000	First lien	Not approved
White	Female	25-34	Conventional	191,000	First lien	Approved
Black	Male	35-44	-	141,000	Not secured	Not approved
White	Male	45-54	VA	35,000	Subordinate	Approved

Race	Gender	Age	Type	Income	Lien status	Action taken
Black	Female	35-44	-	77,000	First lien	Not approved
White	Male	25-34	Conventional	264,000	First lien	Approved

Figure 2: An ML classification pipeline configured for low classification error on the passing (blue) dataset; the configuration has a higher error for the failing (yellow) dataset.

3.1 Example Scenario

We now illustrate the PipeLens framework through a small classification example from our experiments in Section 4, using the glass-box setting while also discussing the corresponding opaque-box setup. As shown in Figure 2, consider a machine learning classification pipeline that takes a dataset D_1 as input and learns a model to predict loan approval or denial for an individual. The pipeline consists of a missing value imputer, followed by a normalizer and a classifier, and evaluates the accuracy. The imputer module considers dropping those instances with missing values, followed by a MinMax [46] scaling approach, and with a logistic regression model it achieves 87% accuracy. We seek to identify potential ways to resolve the pipeline for a different dataset D_2, following the same schema, which has an accuracy of 0.82. In the following, we illustrate how PipeLens operates, primarily in the glass-box setting:

(Step 1) Given historical executions, PipeLens constructs data profiles for the passing dataset under pipeline G, as defined below:

Definition 6 (Data Profile). *Given a pipeline* G *and dataset* D, *a* data profile *is a fixed-length summary of the dataset's characteristics (or of the dataset after some pipeline stages). We define the profile generator as* $\phi(D)$, *or* $\phi(G; D)$ *when conditioned on a pipeline* G, *which maps a dataset* D *and an ordered sequence of preprocessing modules to a fixed-length data profile vector:* $\phi(G; D) \rightarrow \mathbb{R}^p$.

Importantly, ϕ operates only on preprocessing modules, enabling efficient characterization of intermediate data properties without full pipeline execution. Profile selection follows a general principle: each task is assigned the profiles that best capture its data characteristics. We define reusable profile families based on standard data profiling and fair ML practices [2, 39], requiring only minimal task-specific mapping (refer to Table 1 for the profile assignment). This mapping can be largely automated via task metadata, making profile selection systematic and lightweight. A glimpse of profiles for the passing dataset D_1 is shown in Figure 3. A profile consists of a triplet of profile type, attribute, and profile value. For

Passing Data: $D_{pass} \equiv D_1$	Failing Data: $D_{fail} \equiv D_2$
⟨Missing, lien_status, 0.05⟩	⟨Missing, lien_status, 0.1⟩
⟨% of outliers, applicant_age, 0.03⟩	⟨% of outliers, applicant_age, 0.17⟩
⟨Corr with output, applicant_age, .29⟩	⟨Corr with output, applicant_age, 0.33⟩

Figure 3: A glimpse of classification pipeline's profiles for D_1 and D_2 based on the example scenario in Section 3.1.

example, ⟨Missing, lien_status, 0.05⟩ means that the attribute lien_status has 5% missing values.

(Step 2) PipeLens takes into account the causal transitive relationship in which pipeline parameters and module executions influence the intermediate data distribution—captured through data profiles—which, in turn, affects the final utility. To determine the appropriate intervention for a malfunctioning pipeline G_f, PipeLens ranks the available intervention space, which includes both structural and parameter interventions using several ranking strategies: (i) *similarity-based ranking*, which ranks an intervention i by the cosine similarity between the passing profile $\phi(G_f; D_1)$ from Step 1 and the post-intervention failing-data profile $\phi(G_f^i; D_2)$, with intervention i; (ii) *utility prediction ranking*, which ranks interventions using a regression model trained on historical executions. In the glass-box setting, each execution is encoded with module strategies and data profiles, e.g., $x_j = [S^j_{M_1}, S^j_{M_2}, \ldots, S^j_{M_k}, \phi(G; D_j)]$, where $S^j_{M_i}$ is the strategy of module M_i in the j-th execution and $\phi(G; D_j)$ is the corresponding profile vector; the observed utility is the target. PipeLens trains a regression $\hat{u}(x_j)$ to estimate the utility of candidate interventions, from its module choices and post-intervention profile. In the opaque-box setting, intermediate profiles are unavailable, so x_j contains only exposed component strategies; or (iii) a *fusion ranking* that combines the two rankings. The choice of ranking method is adaptively determined from historical execution characteristics, guided by historical lookup to ensure effective, data-driven interventions. The ranking leverages the intuition that the most effective intervention is the one that minimizes the divergence between the resulting and passing profiles, i.e., the profile causally linked to improved downstream utility. The resulting ranking then guides the intervention process. In this example, the available intervention space includes modifying the parameters of existing modules in the malfunctioning pipeline as well as inserting new modules—such as OutlierDetector or Deduplicator—from the module catalog (see Table 2) under various strategies. Following the above-mentioned ranking method, the top-ranked interventions are (OutlierDetector, LocalOutlier Factor(k=20), 3), followed by (OutlierDetector, 2×IQR, 3), suggesting that an outlier-detection module should be inserted immediately after the Normalizer module (index 3) in the malfunctioning pipeline. In other words, the ranking indicates that the absence of an outlier-handling component is likely contributing to the malfunction, and thus suggests an insertion. Since the search space also includes different subsets and re-orderings of the current pipeline, the ranking may similarly prioritize deleting, replacing, or reordering an existing module when it harms utility. Once the intervention ranking is identified, the subsequent step evaluates utility following this order.

(Step 3) As shown in Figure 3, the column applicant_age contains 17% outliers in the failing dataset D_2 compared to only 3% in the passing dataset. Accordingly, PipeLens prioritizes interventions that transform the failing data profile to more closely resemble that of the passing dataset. Before executing an intervention, PipeLens verifies that the candidate module can consume the current intermediate output and produce an output compatible with the next module; incompatible candidates are removed from the intervention space. Next, following the top-ranked intervention, the insertion of LocalOutlierFactor($k = 20$) at the identified position increases classification accuracy to 0.85. In a subsequent step, PipeLens further refines the outlier detector with $2 \times$ IQR, continuing to align the data profile with the passing distribution.

(Step 4) The second intervention raises the utility of the failing dataset above the target threshold $\tau = 0.87$. The resulting pipeline is ⟨ MissingValue:Drop → Normalization:MinMax → OutlierDetector: $2 \times$ IQR → Model:LogisticRegression ⟩. Thus, $G' = I(G)$ combines module insertion and parameter refinement, with Modified(I) = {OutlierDetector : $2 \times$ IQR}. Because intermediate profiles and module order are hidden, the opaque-box setting supports only parameter interventions guided by utility prediction. For this example, it requires five interventions, compared with two in the glass-box setting. In both cases, the repair reaches the target without exhaustive search. Following GRASP [18], PipeLens may additionally explore nearby configurations for further improvement.

3.2 Key Insights

We now present observations, key insights used by PipeLens, and assumptions required to guarantee effectiveness of our algorithms.

(I1) Using data profiles for intervention discovery: Existing research on data profiling [2] aims to identify dataset properties which are predictive of outcome performance. For example, a higher fraction of missing values in a dataset can lead to a lower F1-score for the model. Additionally, datasets with similar distributions are likely to have similar model performance. Using these intuitions, we leverage data profiles as properties of datasets to identify configurations to resolve system malfunction. Although data profiles are generally useful to model the relationship between input datasets and output utility, the signals from a data profile may be noisy. Therefore, PipeLens executes in two phases: the first phase relies on the profile to rank the best intervention, and the second phase systematically explores the overall space, reducing its reliance on the individual intervention signaled by the profile.

(I2) Causal dependencies: The different pipeline modules and datasets are causally dependent. Consider a module M which takes D as input and produces D' as the output. The data profiles of D' are causally dependent on the parameters and choice of modules M and the profiles of the input D. We can therefore represent a data science pipeline as a causal DAG where module parameters and input data profiles impact output data profiles. These insights do not apply in the opaque-box setting, where module impact is modeled directly on utility, bypassing intermediate profiles.

(I3) Use of historical data to learn a proxy function: Generally, a data science pipeline is extensively tested in the development environment before deployment. Availability of this historical execution data is helpful to infer how different modules impact the overall utility of the pipeline. The proxy behavior of the pipeline learned from historical executions can be used to estimate how different

Algorithm 1: PipeLens (Glass-Box)

Input: Pipeline $G(\mathbf{M}, \mathbf{E})$, Failing Dataset D_{fail}, Search Space $\mathbf{T}$, Historical Executions $\mathbf{H}$, Desired Utility τ
Output: Intervened Pipeline G'

1 $\mathbf{P} \leftarrow$ Rank-Single Intervention($\mathbf{H}$, G, $\mathbf{T}$)
2 ISize $\leftarrow 1$
3 **while** $u_G(D_{fail}) < \tau \wedge ISize \leq |\mathbf{M}|$ **do**
/* Phase 1: Optimistic Search */
4 **if** $ISize = 1$ **then**
5 **for** $(M, S_M, i) \in \mathbf{P}$ **do**
6 **if** $M \in \mathbf{M}$ **then** $S_M \leftarrow \sigma(M)$;
7 **else if** $M \in \mathbf{T} \setminus \mathbf{M}$ **then**
8 $S_M \leftarrow \sigma(M)$
9 $\mathbf{M} \leftarrow \mathbf{M}[0{:}i] + (M) + \mathbf{M}[i{:}]$ /* +:insertion */
10 $\mathbf{E} \leftarrow$ Update-Edge(E)
11 $G' \leftarrow G(\mathbf{M}, \mathbf{E})$
/* Phase 2: Exhaustive Search */
12 **else if** $ISize > 1$ **then**
13 $\mathbf{C} \leftarrow$ Get-Conflict-Free-Comb($\mathbf{P}, G$, ISize)
14 $\mathbf{Q} \leftarrow$ Rank-Intervention-Comb($\mathbf{C}$)
15 **for** $((M, S_M, i)_k) \in \mathbf{Q}$ **do**
16 **if** $M_k \in \mathbf{M}$ **then** $S_{M_k} \leftarrow \sigma(M_k),\ \forall\, k \in [2, \text{ISize}]$;
17 **else if** $M_k \in \mathbf{T} \setminus \mathbf{M}$ **then**
18 $\mathbf{M} \leftarrow \mathbf{M}[0{:}i_k] + (M_k) + \mathbf{M}[i_k{:}],\ \forall\, k \in [2, \text{ISize}]$
19 $S_{M_k} \leftarrow \sigma(M_k),\ \forall\, k \in [2, \text{ISize}]$
20 $\mathbf{E} \leftarrow$ Update-Edge(E)
21 $G' \leftarrow G(\mathbf{M}, \mathbf{E})$
22 **if** $u_{G'}(D_{fail}) \geq \tau$ **then** break;
23 **if** $u_{G'}(D_{fail}) > u_G(D_{fail})$ **then** $G \leftarrow G'$;
24 ISize $\leftarrow$ ISize + 1
25 **return** G'

module interventions would behave in the deployed environment. To automate resolving pipeline malfunctions, the repository of past executions can be maintained as a rolling log: every time a pipeline is run in development or production, its profiles and utility can be appended to the history. The surrogate can then be periodically updated using a sliding window. Note that historical data is expected to be highly useful when the data used for development and deployment satisfy the same distribution. This assumption may fail in practical scenarios, but historical data is expected to still de-prioritize interventions that are irrelevant.

3.3 PipeLens in glass-box setting

In practice, the utility metric depends on the profiles of the input dataset, which are themselves influenced by module parameters. In glass-box setting, PipeLens takes into account this causal relationship, and ranks structural and parameter interventions individually by their counterfactual scores and then applies them sequentially following the ranking. The first phase follows an optimistic strategy, applying interventions individually in descending order of their ranking scores to evaluate their isolated impact on pipeline performance. The second phase then explores combinations of these individual interventions, systematically identifying interventions.

Line 1 begins by constructing the compatible atomic intervention set from the module catalog. **Lines 3–24** iterate over every single intervention until the desired threshold is met or the intervention space is exhausted. **Lines 4– 11** evaluate optimistic search one intervention at a time, where each intervention is represented as a tuple (M, S_M, i) where module M with strategy S_M is at the i-th pipeline position. For example, (`Imputer`, `Mean`, 3) specifies a structural intervention that inserts a mean-imputation module at the beginning of the pipeline. **Line 6 Action: Parameter Change.** If the intervened module already exists in the pipeline, its strategy is updated. If a module is suggested for deletion, map its strategy to `None` and treat the module as obsolete. **Lines 7-10 Action: Structural Change.** If the intervened module does not exist in the original pipeline, it inserts the module with its corresponding strategy at the optimal position of the pipeline. Consequently, it updates the edges of DAG. **Lines 11** updates pipeline with the new set of modules and edges. **Lines 12-21** initialize the second phase if the first one fails. **Lines 13-14** iterate all non-conflicting intervention combinations of size ($ISize$) and rank them by the average of their individual scores discussed in Step 2, Section 3.1. Each intervention is represented as a nested tuple of $ISize$ entries in the form (M, S_M, i), where M denotes the module, S_M the strategy, and i the insertion position. For example, for $ISize = 2$, ((`Imputer`, `Mean`, 3), (`Scaling`, `Min` − `Max`, `None`), the first tuple is a structural intervention inserting an imputation module at the beginning of the pipeline, and the second tuple is a parameter intervention changing the scaling module to Min–Max. In **Lines 16-21**, the intervention either changes a module or updates the pipeline structure. The pipeline and its edges are then updated. **Lines 22-24** checks whether an intervention improves the utility. If it does, the intervened pipeline is stored for subsequent iterations. Iterate until the utility goal is achieved or the set of interventions is exhausted. This approach ensures that the local changes to the pipeline contribute toward the final solution.

Helper Functions. 1. Rank-Single Intervention ranks the intervention space using the ranking methods discussed in Step 2, Section 3.1 for glass-box setting. For each candidate, PipeLens computes the post-intervention profile of the failing data and assigns a ranking score. **2. Get-Conflict-Free-Comb** filters the full intervention combination space of size $ISize$ and returns only combinations that are conflict-free—i.e., no module is targeted more than once, and no strategy appears redundantly within a combination. **3. Rank-Intervention-Comb** rank all conflict-free intervention combinations by averaging the constituent single-intervention similarity scores obtained in Rank-Single- Intervention; higher averages indicate stronger combinations.

Complexity. Algorithm 1 has an overall runtime complexity approximated by $T = O\big(S\,C_{\text{prof}} + S \log S + K\,C_{\text{eval}}\big)$, where S is the number of candidate single interventions, C_{prof} denotes the cost of computing a data profile, C_{eval} is the cost of evaluating an intervention, and K represents the number of candidates evaluated before achieving the target utility.

3.4 PipeLens in opaque-box setting

In the opaque-box setting, internal module order and intermediate outputs are inaccessible. Thus, instead of using data profiles, PipeLens learns a surrogate from historical executions, where each

Algorithm 2: PipeLens (Opaque-Box)

Input: Pipeline $G(\mathbf{M}, \mathbf{E})$, Failing Dataset D_{fail}, Search Space $\mathbf{T}$, Historical Executions $\mathbf{H}$, Desired Utility τ
Output: Intervened Pipeline G'

```
1  P ← Rank Parameter(H, G, T)
2  u ← u_G(D_fail)                          /* Failing data utility */
3  ISize ← 1                  /* Determines the intervention size */
4  while u < τ ∧ ISize ≤ |M| do
5      if ISize = 1 then
6          for (M, S_M) ∈ P do
7              S_M ← σ(M)              /* Pick the top parameter */
8              G' ← Modify strategy of M to S_M
9              u' ← u_G'(D_fail)       /* Post intervention u */
10     else if ISize > 1 then
11         C ← Get-Conflict-Free-Comb(P, G, ISize)
12         Q ← Rank-Intervention-Comb(C)
13         for (M, S_M)_k ∈ Q do
14             S_Mk ← σ(M_k), ∀ k ∈ [2, ISize]
15             G' ← Intervene on the modules in G
16             u' ← u_G'(D_fail)
17     if u' > u then G ← G'; u ← u'    /* Update Pipeline */;
18     if u ≥ τ then break;
19     ISize ← ISize + 1
20 return G'
```

execution is represented by exposed module strategies and observed utility (Step 2, Section 3.1). Following the Bayesian-optimization intuition (I3), the surrogate assigns importance scores to individual modules and parameters. Additionally, since the pipeline structure is opaque, PipeLens_O is limited to parameter interventions only.

We now describe PipeLens_O, the opaque-box variant of PipeLens. It follows the same optimistic-then-exhaustive search structure as Algorithm 1, but differs in its ranking strategy and restricts the search space to parameter interventions. In **Lines 1–3**, PipeLens_O trains a surrogate regression from historical executions and ranks module–strategy pairs by their impact on predicted utility. **Lines 4–9** perform optimistic search by applying one ranked parameter intervention at a time and re-evaluating the failing dataset. If no single intervention reaches the target, **Lines 10–16** evaluate conflict-free combinations of ranked parameter interventions. Finally, **Lines 17–19** update the current pipeline whenever the utility improves and terminate once the desired threshold is achieved.

Helper Functions. 1. Rank-Parameters takes historical executions, pipeline, and search space as input; trains a regression surrogate $\hat{u}(x_j)$ to predict utility from strategy choices (as discussed in Step 2, Section 3.1). Finally, it learns module importance from the model coefficients and ranks the parameter intervention by impact.

Complexity. Algorithm 2 first trains a surrogate from historical executions, with cost C_{sur}. If the intervention phase evaluates at most K candidates, the runtime complexity is $T = O(C_{\text{sur}} + K\,C_{\text{eval}})$, where C_{eval} is the cost of running an intervened pipeline on D_{fail}.

4 EXPERIMENTAL EVALUATION

In this section, we seek to answer the following research questions: **RQ1:** How effective is PipeLens in identifying interventions for resolving malfunctioning data science pipelines compared to existing baselines? (Section 4.2); **RQ2:** How do the initial pipeline structure and utility goals affect the performance of PipeLens? (Section 4.3, Figure 8); **RQ3:** How robust are PipeLens and its variants to the availability of historical data of pipeline executions? (Section 4.3, Figure 9); **RQ4:** How well does PipeLens scale as the search space grows? (Section 4.3, Figure 10); **RQ5:** How does the efficiency of PipeLens compare to other baselines? (Section 4.3, Figure 11).

4.1 Experimental Setup

We evaluate four data science tasks—entity matching, classification (fairness and accuracy), and regression—across multiple datasets using diverse modules and performance metrics to assess PipeLens.

4.1.1 Global module catalog. We define a global module catalog that is compatible with the modality and pipelines comprising 46 components applicable to the ML, image, and NLP pipelines (Table 2). Each module supports multiple strategies (e.g., *drop*, *mean*, *median*, *mode*, *k*-nearest Neighbour for imputation), enabling structural and parameter interventions. Detailed catalog and strategy definitions are available in the source code link: PipeLens.

4.1.2 Base pipeline structure and data profiles. For each application, Table 1 specifies the task-specific base data-processing pipeline structure and module sequence, along with the data profiles, learning model, and utility metric. We optimize this base pipeline on the passing dataset, while the remaining modules in Table 2 constitute the candidate module space from which PipeLens may insert appropriate new modules to resolve a malfunctioning pipeline.

4.1.3 Datasets and Tasks. **DBLP-ACM [30]** contains selected publication entities from two CS bibliographic data sources, DBLP and ACM digital library. Task: find entities referring to the same publication from both datasets (entity matching). **Home Mortgage Disclosure Act (HMDA)** contains publicly available loan-level information on 3,075 mortgage applications. Task: predict loan approval/denial (classification). **Adult Census Income [5]** contains demographic and financial information of 48,842 individuals. Task: predict whether an individual earns more than 50K annually (classification). We use this dataset to evaluate fairness of outcomes with gender as the sensitive attribute. **Housing prices [14]** contains information on the characteristics and listed price of 1,460 houses. Task: predict price of a house given its characteristics (regression).

We split each dataset into *passing* and *failing* datasets. We begin by optimizing the task-specific base data-processing pipeline structure in Table 1 and its strategy on the passing data using sampled grid search, and store the resulting historical execution logs separately for each PipeLens variant. To systematically evaluate pipeline repair, we convert the remaining split into failing datasets by injecting controlled perturbations, following data-cleaning benchmarks that corrupt clean data to obtain instances with known ground truth, e.g., BART [3]. Similar to prior ML data-cleaning benchmarks [1, 6], we inject common data-quality violations, including invalid values, missing entries, outliers, duplicates, class imbalance,

Table 1: Data-profile assignment and base data processing pipeline across different tasks

Task	Data Profiles	Base Pipeline	Model & Metric
Entity Matching	Dataset size; stopword distributions; similarity distributions of candidate pairs; stopwords per candidate pair; proxy F1 on a 10% subsample.	⟨ Blocking, Block Cleaner, Comparison Cleaner, Matcher ⟩	Matcher; F1 score of the predicted matching pairs against the ground-truth matches.
Core ML	Missing-value fraction; outlier fraction after each module; outliers per column; input–target correlation.	⟨ Sampling, InvalidValue, TextCleaner, MissingValue, Encoding, FloatingPoint, UnitConverter, DistributionShape, Multicollinearity, Normalization, Outlier, FeatureSelection, DimensionalityReduction, Model ⟩	
▹ *Classification*	Core ML + class-imbalance ratio.		Neural Network/LR; Accuracy
▹ *Fairness*	Core ML + class-imbalance; sensitive-attribute ratio; base rate; Cov(Target, Sensitive).		NN/LR; Statistical Parity [39]
▹ *Regression*	Core ML + output median.		Regression; RMSE

and feature-correlation corruption. These perturbations model realistic pipeline assumption violations, such as domain violations, incomplete records, noisy measurements, repeated entities, and shifted class distributions. The corruption process is used only to construct benchmark failures and is not exposed to PipeLens.

4.1.4 Competing methods. We consider four families of baselines: optimization methods, data-driven debugging methods, software-debugging methods, and PipeLens variants:

GridSearch-ES searches randomly over the set of all combinations of parameters and stops as soon as the goal utility is achieved.

Learn2Clean [6] uses reinforcement learning to determine the best ML pipeline for a given dataset and a quality performance metric. We modified the initial reward matrix to reflect our pipeline structure, and use the default hyperparameter settings.

BugDoc [36] explores different system parameter configurations to determine the root cause of system failure. We consider an optimized version that identifies a cause in the smallest possible budget.

DataPrism [20] is a debugging technique that identifies data profiles that are root causes of performance degradation and proposes modifying the data profiles to improve system behavior.

ProbDD [63] extends delta debugging [67] by localizing candidates with suspiciousness probabilities. We treat the input data as candidates and conduct a probabilistic search to find a minimal error-inducing subset.

DStar [65] ranks suspicious program statements from passing and failing executions. We adapt it by treating modules and strategies as statements, computing their suspiciousness over the test space, and intervening in ascending order of rank.

Table 2: Global module catalog

Module Category
Numerical Data: (count=15) Sampling, Imputer [46], Standardizer [46], Outlier Detector [46], Deduplicator, Encoder, DistributionShapeCorrector [46], Binning, FloatingPointStabilization, InvalidValueRepair, UnitConverter, MulticollinearityDetection [46], DimensionalityReduction [46], NonLinearFeatureTransformer, FeatureSelector.
Entity Matching & NLP: (count=22) Blocking, BlockCleaner, ComparisonCleaner, Matcher, StopwordRemover [9], PunctuationRemover, LanguageDetector, LanguageTranslator, SpellChecker [35], Tokenizer, Embedding [46], Lemmatizer [9], Stemmer [9], DateSeparator, VocabularyPruning, SpecialCharacterRemover, QueryNormalizer, Reranker, Retriever, LanguageWrapper, ContextBuilder, Lowercaser.
Image Data: (count=4) Format Converter[60], Cropper[60], Resizer[60], Rotator[60].
Models: (count=5). Logistic Regression [46], Linear Regression [46], SVM [46], Random Forest [46], Neural Network [46].

PipeLens is our glass-box solution (Algorithm 1) that uses data profiles to guide the search space of system parameters.

PipeLens_O is our opaque-box solution (Algorithm 2) that ranks system parameters based on their correlation with the output and utilizes the ranking to select parameter combinations.

PipeLens_P is a combined variant of PipeLens that integrates PipeLens_O. Specifically, parameter interventions are ranked using PipeLens_O, structural interventions are ranked using PipeLens.

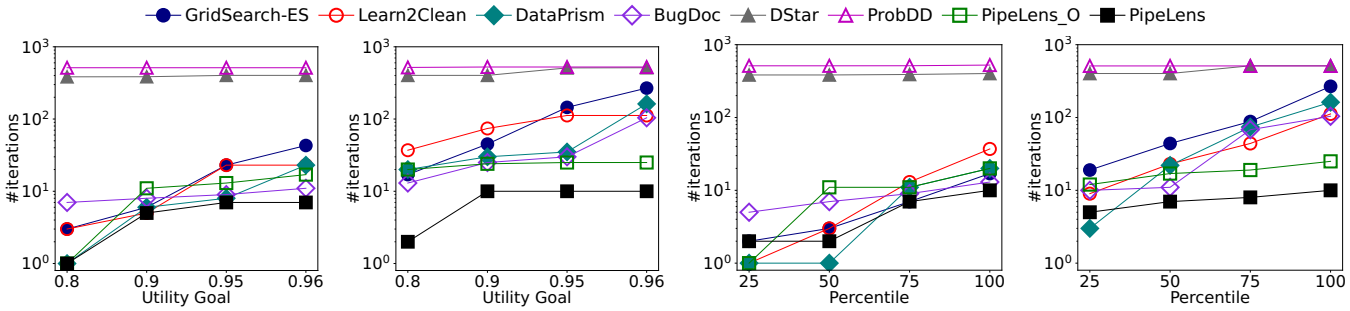
4.1.5 Performance metrics. We sample a set of malfunctioning pipeline configurations and evaluate each method by the number of interventions (i.e., model training), wall-clock time, and peak memory overhead to reach utility targets on the failing dataset.

4.2 End-to-end System Performance

In this set of experiments, we answer **RQ1** by comparing the performance of our solution PipeLens with the corresponding baselines for each data science application.

4.2.1 Performance analyses. Given a set of malfunctioning pipeline configurations, we compare methods by the number of iterations, i.e., model trainings, required to reach a target utility; fewer iterations indicate better performance. Figures 4–7 report results across tasks. Subfigures (a) and (b) show the median (50th percentile) and worst-case (100th percentile) iteration counts, respectively, while subfigures (c) and (d) show percentile-based iteration distributions for an easier goal met by approximately 80% of configurations and a stricter goal met by approximately 5%.

Entity matching. Figure 4 shows that PipeLens consistently requires fewer interventions across utility goals. In the median case (Figure 4a), PipeLens reaches the target within 1–7 iterations, while the baselines require up to 43 for GridSearch-ES, 23 for DataPrism and Learn2Clean, 11 for BugDoc, and hundreds for DStar and ProbDD. PipeLens_O also remains competitive, requiring at most 17 iterations despite operating in the opaque-box setting without intermediate profiles. The gap is larger in the worst case (Figure 4b): PipeLens needs at most 10 iterations and PipeLens_O at most 25, whereas the remaining baselines require up to 104–268 iterations, with DStar and ProbDD exceeding 500 iterations. The percentile plots confirm that PipeLens is more stable across failing configurations. For the easier goal (Figure 4c), DataPrism is competitive in lower percentiles but grows to 20 iterations, while PipeLens remains within 2–10. Under the stricter goal (Figure 4d), DataPrism increases to 162 iterations, whereas PipeLens stays within 5–10



(a) Median-case analysis (b) Worst-case analysis (c) Easier F-score Goal: 0.80 (d) Strict F-score Goal: 0.96

Figure 4: Comparison of number of iterations for the entity matching pipeline.

Table 3: Entity matching case study. ** indicates the passing pipeline. (BC: BlockCleaner; CC: ComparisonCleaner)

Iteration	Blocking	Block Cleaning	BC threshold	CC weight	Matching threshold
initial	0	0	0.35	2	0.65
1	5	0	0.35	2	0.65
2	5	0	0.95	2	0.65
3**	5	0	0.05	2	0.65

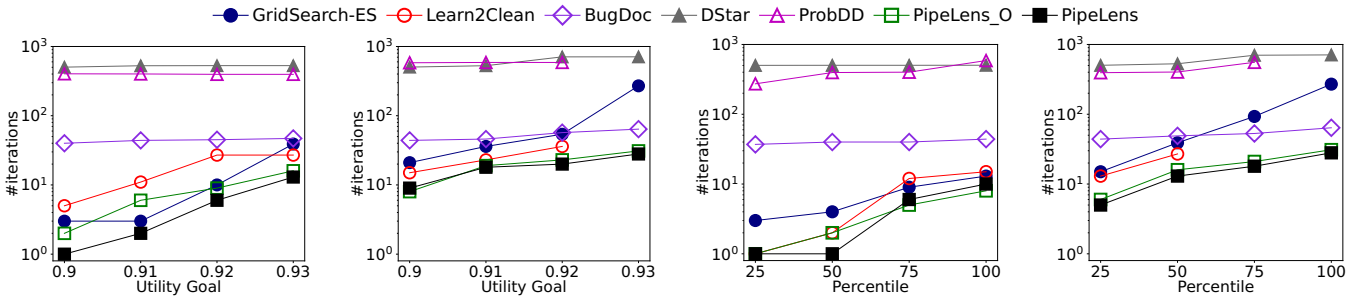
across all percentiles. ProbDD and DStar remain costly because they repeatedly retrain models and evaluate pipelines only to rank suspicious elements, without providing repair guidance; after localization, we still use PipeLens's repair search mechanism.

We further analyze a malfunctioning pipeline as a case study in Table 3, which details the repair trajectory. The initial parameters achieve an F1-score of 0.95105 on the passing dataset but only 0.0491 on the failing dataset. Starting from this configuration, PipeLens reaches the target F1-score of 0.93 in 3 iterations, compared to 6 iterations for GridSearch-ES in the median case (best: 3, worst: 11). Following the ranked profiles, PipeLens first modifies the blocking parameter (Iteration 1) to reduce average similarity, then adjusts the block-cleaning threshold (Iteration 2) to improve similarity distribution statistics. Since higher-order similarity profiles are already optimized, the final step (Iteration 3) targets the sample F1-score directly, resulting in an F1-score of 0.9316 and early termination.

Classification (accuracy). We allow both structural and parameter interventions, enabling methods to insert, reorder, or modify modules and their strategies. Figures 5a and 5b show that PipeLens consistently requires fewer interventions in both median and worst-case settings. In the median case, PipeLens reaches the target within 1–13 iterations, compared with up to 16 for PipeLens_O, 39 for GridSearch-ES, 27 for Learn2Clean, 47 for BugDoc, and hundreds for DStar and ProbDD. In the worst case, PipeLens requires at most 28 iterations and PipeLens_O remains close with at most 31, while the other baselines require substantially more iterations, reaching up to 270 for GridSearch-ES, 64 for BugDoc, 709 for DStar, and 588 for ProbDD. BugDoc becomes costly because structural interventions and module reordering expand its combinatorial design space. Learn2Clean explores module parameters through Q-learning, but its reward-driven search may overlook effective configurations in the larger action space, especially for stricter goals. Notably, BugDoc and Learn2Clean mostly fail to reach the stricter goal within the iteration limit. GridSearch-ES can occasionally find good configurations for easier goals, but lacks search guidance and therefore requires many iterations. ProbDD and DStar remain costly because they first localize failure-inducing inputs by repeatedly evaluating each split or suspicious subset, rather than directly searching for repairs. PipeLens_O improves over these methods by using historical execution signals in the opaque-box setting, while PipeLens further leverages the intermediate profile to rank interventions more precisely. The percentile plots confirm the same trend. For the easier goal (Figure 5c), PipeLens reaches the target within 1–10 iterations across percentiles, and PipeLens_O remains similarly efficient within 1–8. Under the stricter goal (Figure 5d), the gap widens: PipeLens requires only 5–28 iterations, while PipeLens_O stays within 31, GridSearch-ES reaches 270, and BugDoc requires 44–64 iterations.

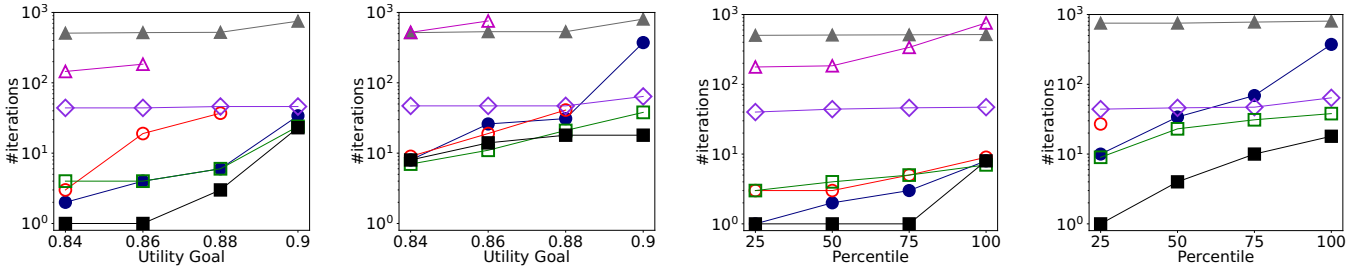
Classification (fairness). PipeLens shows a consistent advantage on the fairness task because the utility goal depends strongly on distributional and subgroup-sensitive profiles. By using these profile signals, PipeLens prioritizes interventions that directly improve fairness rather than broadly exploring the pipeline space. As shown in Figures 6a and 6b, PipeLens reaches the target with few interventions in both median and worst-case settings, while PipeLens_O remains competitive but needs more iterations because it lacks intermediate profiles. The percentile plots show the same trend with one notable exception: for the easier fairness goal (Figure 6c), both PipeLens_O and GridSearch-ES also perform well, since many configurations easily satisfy this relaxed threshold. However, under the stricter goal (Figure 6d), the gap becomes clear: PipeLens remains within 1–18 iterations, while PipeLens_O reaches 38, GridSearch-ES grows substantially, and BugDoc requires more evaluations due to its larger combinatorial search space. Learn2Clean and ProbDD fail to cover all percentiles under the strict goal, while DStar remains similarly costly.

Regression. The regression task differs from classification because utility is driven by continuous prediction error rather than discrete decision quality. In Figure 7, we report normalized RMSE utility, where 1.00 corresponds to the stricter RMSE target (= 156) and 0.85 to the easier target (= 180). PipeLens performs especially well



(a) Median-case analysis (b) Worst-case analysis (c) Easier accuracy goal: 0.90 (d) Strict accuracy goal: 0.93

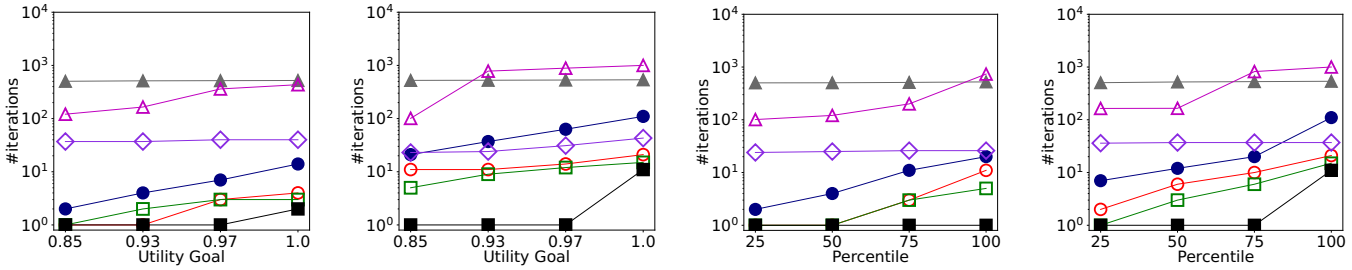
Figure 5: Comparison of number of iterations for the classification pipeline (accuracy; neural network).



(a) Median-case analysis (b) Worst-case analysis (c) Easier fairness goal: 0.84 (d) Strict fairness goal: 0.90

Figure 6: Comparison of number of iterations for the classification pipeline (fairness; neural network)

.



(a) Median-case analysis (b) Worst-case analysis (c) Normalized RMSE goal: 0.85 (d) Normalized RMSE goal: 1.0

Figure 7: Comparison of number of iterations for the regression pipeline

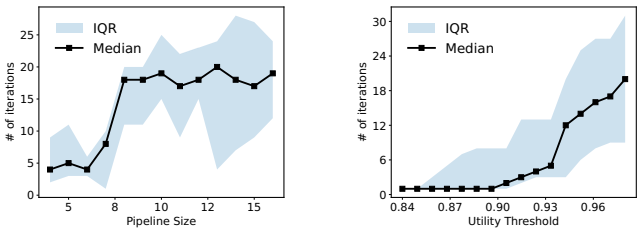
.

in this setting: in the median case (Figure 7a), it reaches all targets within 1–2 interventions, and in the worst case (Figure 7b) remains within 11. This indicates that the profile-guided ranking quickly identifies the numerical preprocessing changes that most affect RMSE. The percentile plots further show that regression repairs are highly concentrated for PipeLens. For the easier goal (Figure 7c), PipeLens needs only one intervention across all percentiles. Under the stricter goal (Figure 7d), it remains bounded within 1–11 interventions, while PipeLens_O reaches 15 and the other baselines grow more substantially, with GridSearch-ES reaching 110 and BugDoc requiring 36–37. ProbDD and DStar remain much higher because their repeated localization-style evaluations do not directly target RMSE-improving pipeline repairs.

4.3 Ablation Analysis

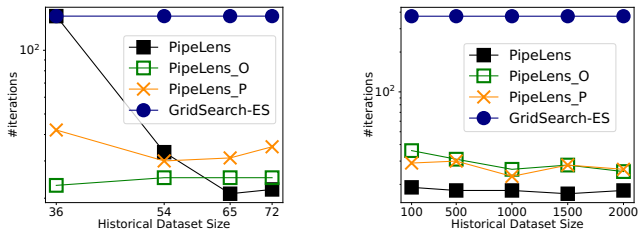
In this set of experiments, we answer **RQ2** to **RQ5** by reporting the performance of PipeLens in different settings.

Varying pipeline size and utility goal. In this section, we address **RQ2** by varying the initial configuration, including base pipeline size and goal utility. In Figure 8a, we evaluate PipeLens on the fairness task (Adult Income Dataset) with logistic regression for a goal utility of 0.95 while varying the base pipeline size from 4 to 16 modules. From a pool of 25 modules—comprising the base modules in Table 1 and additional components from Table 2—we gradually increase the number of modules in the initial pipeline, with the remaining modules forming the new module space for insertion.



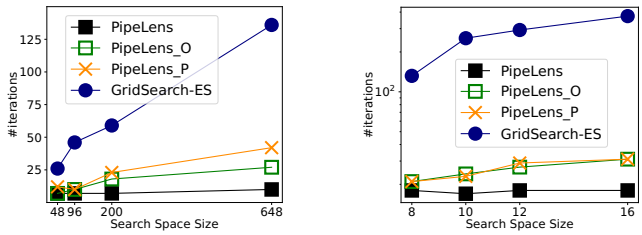
(a) Varying base pipeline size (b) Varying goal utility

Figure 8: Effect of initial pipeline and target goal.



(a) Entity matching (goal=0.96) (b) Fairness (goal = 0.95; LR)

Figure 9: Effect of historical dataset size.



(a) Entity matching (goal=0.96) (b) Fairness (goal = 0.95; LR)

Figure 10: Effect of search space size.

When the base pipeline is small, PipeLens quickly identifies passing configurations, primarily by inserting beneficial modules from new module space with appropriate parameter settings. As pipeline size increases (for example, from 6 to 10 modules), the number of required iterations rises modestly, reflecting the need for more complex actions, such as parameter adjustments or the deletion of detrimental modules along with insertions. Beyond this range, iteration counts stabilize despite further increases in pipeline size.

In Figure 8b, we examine PipeLens under progressively stricter utility goals using the same dataset. For easier utility goal, PipeLens identifies minimal interventions in only a few iterations. As the requirement tightens beyond 0.93, the iteration count increases approximately linearly, reflecting the shrinking feasible space of valid configurations. At the strictest fairness goal (0.98), which is satisfied by fewer than 1% of pipeline configurations, none of the baselines reach the target within the 1000-iteration limit. In contrast, PipeLens achieves the goal with a median of 20 iterations and an interquartile range of 9–31, demonstrating robust performance under highly constrained fairness objectives.

Effect of historical run size. Here we address **RQ3** by varying the size of the historical dataset and comparing PipeLens with the opaque-box variant PipeLens_O and the hybrid variant PipeLens_P. Since other baselines do not utilize historical data, they are excluded except for GridSearch-ES as a reference. Figure 9 reports the iterations (worst-case) required to reach the strictest utility goal for a fixed pipeline as a function of historical run size.

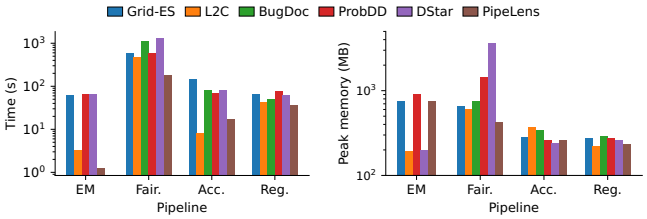
For the EM pipeline (Figure 9a), with a smaller historical dataset size, PipeLens requires far more iterations than its variants. With limited passing-profiles information, PipeLens does not learn a reliable ranking. This trend is also evident in PipeLens_P which uses profiles to obtain parameter rankings. With a random sampling of historical data, PipeLens_O is not affected by the lack of enough historical data, needing comparable iterations across all historical data sizes. Note that with a little over half of historical data, PipeLens requires drastically fewer iterations. For classification (Figure 9b), historical size has less impact on PipeLens because a single passing profile is often enough for similarity-based comparison. However, PipeLens_O and PipeLens_P depend more on historical runs for learning stable rankings, although even a small sample of about 100 instances captures useful signals and further reduces iterations.

Scalability analysis. Figure 10 addresses **RQ4** by reporting worst-case performance under the strictest utility goals while varying search space. For entity matching (Figure 10a), we vary the parameter search space while keeping the pipeline structure fixed. As the search space expands, all methods require more iterations to achieve the target. Variants of PipeLens show sub-linear growth; although PipeLens_O and PipeLens_P leverage historical data, they do not explicitly model how parameters influence intermediate data profiles and, consequently, the utility metric, leading to higher iteration counts than PipeLens.

For the fairness task (Figure 10b), where both structural and parameter interventions are allowed, the search space grows factorial-exponentially as $\sum_{\mathbf{M}\subseteq\{1,\ldots,n\}} |\mathbf{M}|! \prod_{M\in\mathbf{M}} S_M$, where n denotes the number of modules in the search space and S_M the strategy set of module M. As n increases from 8 to 16, other methods show a clear upward trend in iterations. In contrast, PipeLens maintains nearly constant performance, as it ranks interventions individually based on their proxy impact and applies them in priority order, making its effectiveness largely independent of the search space size.

Efficiency. To address **RQ5**, we evaluate efficiency by selecting one arbitrary malfunctioning pipeline from each application and optimizing it toward a moderately strict goal. Figures 11a and 11b report wall-clock time and peak memory overhead, respectively, while Figures 4d–7d provide the corresponding intervention counts.

As shown in Figure 11a, PipeLens is generally faster on larger or higher-dimensional tasks, such as classification (fairness) and regression, because its ranking-based search avoids many expensive intervention evaluations and model retrainings. Across tasks, PipeLens spends roughly 90% of its runtime on one-time intervention ranking; this front-loaded cost prioritizes the most promising interventions and makes the subsequent search phase much cheaper, requiring only about 10% of the runtime for iterative evaluation and decision making. In contrast, GridSearch-ES requires repeated



(a) Required wall-clock time (s) **(b) Peak memory overhead (MB)**

Figure 11: Efficiency comparison under stricter utility goals.

retraining over the full dataset, BugDoc suffers from a larger combinatorial search space, Learn2Clean incurs reward-update overhead, and ProbDD/DStar require many evaluations to localize failure-inducing subsets. The main exception is classification (accuracy), where Learn2Clean is faster because the dataset is lower dimensional, so PipeLens's profiling overhead is not fully amortized. Nevertheless, PipeLens remains more evaluation-efficient, which is important when training is expensive.

Figure 11b further shows that PipeLens incurs only modest peak memory overhead. This overhead comes from generating data profiles and ranked interventions, but it does not dominate execution. In most ML settings, PipeLens requires less peak memory than most competing methods, with the lowest peak memory in the classification-fairness task. In entity matching, PipeLens uses more memory than Learn2Clean and DStar due to richer matching-profile storage, but this cost accompanies a large reduction in both runtime and intervention count (Figure 4d).

Takeaway. Overall, the experiments show that PipeLens consistently identifies utility-improving interventions with fewer evaluations across tasks, goals, and search-space sizes. Its advantage comes from leveraging profile-guided ranking, which narrows the search space to promising structural and parameter interventions while maintaining practical runtime and memory overhead.

5 RELATED WORK

Debugging. Data-driven debugging mechanisms [20, 36] determine parameter settings or data transformations as root causes of pipeline failures, but do not explain the underlying reasons. While DataPrism [20] transforms the failing dataset to fix violating data profiles that cause failure, PipeLens leverages data profiles to determine changes that must be made to the pipeline to resolve the malfunction. Data debuggers such as Dagger [51] and CheckCell [4] operate on specific primitives or individual cells. Data-cleaning [50, 52] and fair ML [21, 40, 55] techniques suggest transformations yet do not determine concrete interventions for pipeline repair. Failure localization has been a longstanding focus in software debugging [17, 45, 63, 67]. Techniques such as Tarantula [27], DStar [65], and Op2 [43] rank suspicious program elements using passing and failing executions, while BIGSIFT [23] combines data provenance with delta debugging [67] to localize failure-inducing inputs. These approaches assume a fixed program and focus on identifying the origin of failures while data science pipelines are configurable workflows, often containing black-box modules, where failures arise from interactions between data properties and pipeline configurations. They identify *where* failures arise but are not sufficient to determine *how* to resolve them.

ML-based data cleaning. HoloClean [50] and HoloDetect [25] incorporate repair signals into statistical models but require manual tuning. ActiveClean [33], Raha [38], and Baran [37] rely on user-defined cleaning processes or labeled clusters to identify dirty data. Unlike these approaches, PipeLens operates over entire pipelines and automatically selects interventions that fix malfunction.

Task-dependent data cleaning. Methods such as ActiveClean [33], BoostClean [32], AlphaClean [34], CPClean [28], REIN [1] and Learn2Clean [6] leverage ML model feedback but target specific models or error types. Saga [57] enumerates candidate pipelines to improve model quality but does not use historical executions to guide search. Moreover, these techniques construct pipelines from scratch rather than intervening on an existing one.

AutoML and hyperparameter optimization. Systems such as AutoWEKA [31, 59], Auto-sklearn [19], BOHB [16], TuPAQ [58], TPOT [44], Alpine Meadow [56], DeepLine [24] integrate limited preprocessing and primarily focus on model selection and tuning. They neither generalize beyond ML tasks nor support pipeline interventions. PipeLens uses search and utility improvement as mechanisms for discovering pipeline repairs: in contrast to pipeline optimization, it starts from an existing pipeline that satisfied the application requirement on one dataset but malfunctions on another.

Explanations. Prior work explains query results [64], identifies data generation errors [64], or discovers dataset relationships [12]. Explainable AI methods [53, 54] approximate model behavior by perturbations, while others [61] analyze failure causes under known corruption mechanisms. However, none provide actionable interventions to repair pipeline malfunction, which is our focus.

6 CONCLUSION

We consider the problem of resolving malfunctioning data science pipelines through targeted interventions such as data transformations or pipeline configuration modifications. We formalize a pipeline as a directed acyclic graph, with nodes as components and edges representing data flow, making the approach broadly applicable to various domains. We consider two scenarios: (i) opaque-box, where component internals are hidden, and (ii) glass-box, where the outputs of pipeline components are accessible. Our proposed algorithm PipeLens exploits the causal relationships between pipeline parameters, data profiles, and their impact on pipeline utility, along with historical executions, to prioritize interventions. Empirical evaluation demonstrates the superior performance of PipeLens compared to several baselines. In the future, we plan to extend PipeLens to handle multi-objective optimization of data science pipelines with cycles.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable and constructive feedback. This research was supported by the National Science Foundation (NSF) under Grant No. 2237149.

REFERENCES

[1] Mohamed Abdelaal, Christian Hammacher, and Harald Schoening. 2023. REIN: A comprehensive benchmark framework for data cleaning methods in ML pipelines. *arXiv preprint arXiv:2302.04702* (2023).

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

[3] Patricia C Arocena, Boris Glavic, Giansalvatore Mecca, Renée J Miller, Paolo Papotti, and Donatello Santoro. 2015. Messing up with BART: error generation for evaluating data-cleaning algorithms. *Proceedings of the VLDB Endowment* 9, 2 (2015), 36–47.

[4] Daniel W. Barowy, Dimitar Gochev, and Emery D. Berger. 2014. CheckCell: data debugging for spreadsheets. In *OOPSLA*. 507–523.

[5] Barry Becker and Ronny Kohavi. 1996. Adult. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5XW20.

[6] Laure Berti-Equille. 2019. Learn2Clean: Optimizing the Sequence of Tasks for Web Data Preparation. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 2580–2586. doi:10.1145/3308558.3313602

[7] Jelke Bethlehem. 2010. Selection bias in web surveys. *International statistical review* 78, 2 (2010), 161–188.

[8] Bias in Amazon Hiring. https://becominghuman.ai/amazons-sexist-ai-recruiting-tool-how-did-it-go-so-wrong-e3d14816d98e

[9] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.".

[10] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large. In *Proceedings of the 44th International Conference on Software Engineering*. 2091–2103.

[11] Gabriel Cadamuro, Ran Gilad-Bachrach, and Xiaojin Zhu. 2016. Debugging machine learning models. In *ICML Workshop on Reliable Machine Learning in the Wild*.

[12] Fernando Chirigati, Harish Doraiswamy, Theodoros Damoulas, and Juliana Freire. 2016. Data Polygamy: The Many-Many Relationships Among Urban Spatio-Temporal Data Sets. In *Proceedings of ACM SIGMOD (SIGMOD '16)*. ACM, New York, NY, USA, 1011–1025.

[13] Aron Culotta. 2014. Reducing sampling bias in social media data for county health inference. In *Joint Statistical Meetings Proceedings*. Citeseer, 1–12.

[14] Dean De Cock. 2011. Ames, Iowa: Alternative to the Boston housing data as an end of semester regression project. *Journal of Statistics Education* 19, 3 (2011).

[15] Sijie Dong, Qitong Wang, Soror Sahri, Themis Palpanas, and Divesh Srivastava. 2024. Efficiently Mitigating the Impact of Data Drift on Machine Learning Pipelines. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3072–3081.

[16] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and efficient hyperparameter optimization at scale. In *International conference on machine learning*. PMLR, 1437–1446.

[17] Anna Fariha, Suman Nath, and Alexandra Meliou. 2020. Causality-Guided Adaptive Interventional Debugging. In *SIGMOD*. 431–446.

[18] Thomas A Feo and Mauricio GC Resende. 1995. Greedy randomized adaptive search procedures. *Journal of global optimization* 6 (1995), 109–133.

[19] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. *Advances in neural information processing systems* 28 (2015).

[20] Sainyam Galhotra, Anna Fariha, Raoni Lourenço, Juliana Freire, Alexandra Meliou, and Divesh Srivastava. 2022. Dataprism: Exposing disconnect between data and systems. In *Proceedings of the 2022 International Conference on Management of Data*. 217–231.

[21] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna M. Wallach, Hal Daumé III, and Kate Crawford. 2018. Datasheets for Datasets. *CoRR* abs/1803.09010 (2018). arXiv:1803.09010

[22] Google Vision Racism. https://algorithmwatch.org/en/story/google-vision-racism/

[23] Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. 2017. Automated debugging in data-intensive scalable computing. In *Proceedings of the 2017 Symposium on Cloud Computing*. 520–534.

[24] Yuval Heffetz, Roman Vainshtein, Gilad Katz, and Lior Rokach. 2020. Deepline: Automl tool for pipelines generation using deep reinforcement learning and hierarchical actions filtering. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2103–2113.

[25] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*. 829–846.

[26] Is Amazon same-day delivery service racist? 2016. The Christian Science Monitor. https://www.csmonitor.com/Business/2016/0423/Is-Amazon-same-day-delivery-service-racist

[27] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[28] Bojan Karlaš, Peng Li, Renzhi Wu, Nezihe Merve Gürel, Xu Chu, Wentao Wu, and Ce Zhang. 2020. Nearest neighbor classifiers over incomplete information: From certain answers to certain predictions. *arXiv preprint arXiv:2005.05117* (2020).

[29] Michelle E Kho, Mark Duffett, Donald J Willison, Deborah J Cook, and Melissa C Brouwers. 2009. Written informed consent and selection bias in observational studies using medical records: systematic review. *Bmj* 338 (2009).

[30] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of entity resolution approaches on real-world match problems. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 484–493. doi:10.14778/1920841.1920904

[31] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. 2017. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *Journal of Machine Learning Research* 18, 25 (2017), 1–5.

[32] Sanjay Krishnan, Michael J Franklin, Ken Goldberg, and Eugene Wu. 2017. Boostclean: Automated error detection and repair for machine learning. *arXiv preprint arXiv:1711.01299* (2017).

[33] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. 2016. Activeclean: Interactive data cleaning for statistical modeling. *Proceedings of the VLDB Endowment* 9, 12 (2016), 948–959.

[34] Sanjay Krishnan and Eugene Wu. 2019. Alphaclean: Automatic generation of data cleaning pipelines. *arXiv preprint arXiv:1904.11827* (2019).

[35] Steven Loria et al. 2018. textblob Documentation. *Release 0.15* 2, 8 (2018), 269.

[36] Raoni Lourenço, Juliana Freire, and Dennis Shasha. 2020. BugDoc: Algorithms to Debug Computational Processes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 463–478. doi:10.1145/3318464.3389763

[37] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective error correction via a unified context representation and transfer learning. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1948–1961.

[38] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Raha: A configuration-free error detection system. In *Proceedings of the 2019 International Conference on Management of Data*. 865–882.

[39] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. 2021. A Survey on Bias and Fairness in Machine Learning. *Comput. Surveys* 54, 6 (July 2021), 115:1–115:35. doi:10.1145/3457607

[40] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. 2019. Model Cards for Model Reporting. *Proceedings of the Conference on Fairness, Accountability, and Transparency* (Jan 2019).

[41] Piero Molino and Christopher Ré. 2021. Declarative machine learning systems. *Commun. ACM* 65, 1 (2021), 42–49.

[42] Mohammad Mehdi Morovati, Amin Nikanjam, Florian Tambon, Foutse Khomh, and Zhen Ming Jiang. 2024. Bug characterization in machine learning-based systems. *Empirical Software Engineering* 29, 1 (2024), 14.

[43] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.

[44] Randal S Olson and Jason H Moore. 2016. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*. PMLR, 66–74.

[45] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.

[46] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[47] Martin Pelikan. 2005. Bayesian optimization algorithm. In *Hierarchical Bayesian optimization algorithm: toward a new generation of evolutionary algorithms*. Springer, 31–48.

[48] Oona Rainio, Jarmo Teuho, and Riku Klén. 2024. Evaluation metrics and statistical tests for machine learning. *Scientific reports* 14, 1 (2024), 6086.

[49] Sergey Redyuk, Zoi Kaoudi, Volker Markl, and Sebastian Schelter. 2021. Automating Data Quality Validation for Dynamic Data Ingestion.. In *EDBT*. 61–72.

[50] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. 2017. HoloClean: holistic data repairs with probabilistic inference. *PVLDB* 10, 11 (2017), 1190–1201.

[51] El Kindi Rezig, Lei Cao, Giovanni Simonini, Maxime Schoemans, Samuel Madden, Nan Tang, Mourad Ouzzani, and Michael Stonebraker. 2020. Dagger: A Data (not code) Debugger. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*.

[52] El Kindi Rezig, Mourad Ouzzani, Walid G Aref, Ahmed K Elmagarmid, Ahmed R Mahmood, and Michael Stonebraker. 2021. Horizon: scalable dependency-driven data cleaning. *PVLDB* 14, 11 (2021), 2546–2554.

[53] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should I trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd*

ACM SIGKDD international conference on knowledge discovery and data mining. 1135–1144.

[54] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2018. Anchors: High-Precision Model-Agnostic Explanations.. In *AAAI*, Vol. 18. 1527–1535.

[55] Babak Salimi, Luke Rodriguez, Bill Howe, and Dan Suciu. 2019. Interventional Fairness: Causal Database Repair for Algorithmic Fairness. In *SIGMOD*. 793–810.

[56] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing data science through interactive curation of ml pipelines. In *Proceedings of the 2019 international conference on management of data*. 1171–1188.

[57] Shafaq Siddiqi, Roman Kern, and Matthias Boehm. 2023. SAGA: A Scalable Framework for Optimizing Data Cleaning Pipelines for Machine Learning Applications. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–26.

[58] Evan R Sparks, Ameet Talwalkar, Daniel Haas, Michael J Franklin, Michael I Jordan, and Tim Kraska. 2015. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 368–380.

[59] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 847–855.

[60] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. 2014. scikit-image: image processing in Python. *PeerJ* 2 (2014), e453.

[61] Paroma Varma, Dan Iter, Christopher De Sa, and Christopher Ré. 2017. Flipper: A systematic approach to debugging training sets. In *Proceedings of the 2nd Workshop on Human-in-the-Loop Data Analytics*. 1–5.

[62] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. 2015. Seedb: Efficient data-driven visualization recommendations to support visual analytics. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, Vol. 8. 2182.

[63] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 881–892. doi:10.1145/3468264.3468625

[64] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data X-Ray: A Diagnostic Tool for Data Errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1231–1245.

[65] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability* 63, 1 (2014), 290–308. doi:10.1109/TR.2013.2285319

[66] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. Production machine learning pipelines: Empirical analysis and optimization opportunities. In *Proceedings of the 2021 international conference on management of data*. 2639–2652.

[67] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200. doi:10.1109/32.988498