Saga: A Scalable Framework for Optimizing Data Cleaning Pipelines for Machine Learning Applications

SHAFAQ SIDDIQI, Graz University of Technology, Austria
ROMAN KERN, Graz University of Technology, Austria
MATTHIAS BOEHM*, Technische Universität Berlin, Germany

In the exploratory data science lifecycle, data scientists often spent the majority of their time finding, integrating, validating and cleaning relevant datasets. Despite recent work on data validation, and numerous error detection and correction algorithms, in practice, data cleaning for ML remains largely a manual, unpleasant, and labor-intensive trial and error process, especially in large-scale, distributed computation. The target ML application—such as classification or regression models—can be used as a signal of valuable feedback though, for selecting effective data cleaning strategies. In this paper, we introduce Saga, a framework for automatically generating the top-K most effective data cleaning pipelines. Saga adopts ideas from Auto-ML, feature selection, and hyper-parameter tuning. Our framework is extensible for user-provided constraints, new data cleaning primitives, and ML applications; automatically generates hybrid runtime plans of local and distributed operations; and performs pruning by interesting properties (e.g., monotonicity). Instead of full automation—which is rather unrealistic—Saga simplifies the mechanical aspects of data cleaning. Our experiments show that Saga yields robust accuracy improvements over state-of-the-art, and good scalability regarding increasing data sizes and number of evaluated pipelines.

CCS Concepts: • **Information systems → Data cleaning**; **Data cleaning**; • **Computing methodologies → Distributed computing methodologies**.

Additional Key Words and Phrases: Data Cleaning for ML; Linear-algebra-based Primitives; Data Cleaning Pipelines; Evolutionary Algorithms; Hyper-parameter Tuning; Data- and Task-parallel Execution

ACM Reference Format:
Shafaq Siddiqi, Roman Kern, and Matthias Boehm. 2023. Saga: A Scalable Framework for Optimizing Data Cleaning Pipelines for Machine Learning Applications. *Proc. ACM Manag. Data* 1, 3 (SIGMOD), Article 218 (September 2023), 26 pages. https://doi.org/10.1145/3617338

1 INTRODUCTION

Data integration, validation, and cleaning are old, well-studied, but complex problems [83, 99]. Sources of data quality issues are manifold, including heterogeneous data sources (e.g., syntactic or semantic heterogeneity, misalignments, outliers, missing values, and duplicates) [1, 41, 78, 79], human error[1] (e.g., typos, attribute swaps, incorrect labels) [45], or measurement and processing errors (e.g., erroneous sensors or extraction programs). These issues are traditionally addressed

*This work was partially done at Graz University of Technology, Austria.
[1]Such errors then unfortunately even propagate into highly-curated data collections such as DBLP. Interesting examples include attribute swaps in affiliations (examples meanwhile fixed), and typos in dissertation titles (e.g., link [106]).

Authors' addresses: Shafaq Siddiqi, shafaq.siddiqi@tugraz.at, Graz University of Technology, Austria; Roman Kern, rkern@tugraz.at, Graz University of Technology, Austria; Matthias Boehm, matthias.boehm@tu-berlin.de, Technische Universität Berlin, Germany.

2836-6573/2023/9-ART218
https://doi.org/10.1145/3617338

via tool-supported, but manually crafted ETL (extract, transform, load) processes [14, 96], schema matching [6], entity resolution or deduplication [18], or UI-centric data wrangling [50, 74, 82]. Orthogonal tools for data profiling [33], data validation [89, 91], and error detection and correction [25, 44, 57, 59, 63, 64, 83, 108] share the characteristics of discovering and applying constraints such as functional and inclusion dependencies; domain constraints, uniqueness, expected value ranges and distinct items; as well as validity patterns and denial constraints.

Data Cleaning for ML: Data cleaning for data science or machine learning (ML) has unique characteristics. The typical data science lifecycle is exploratory: data scientists deal with under-specified objectives, pose hypotheses, and prepare the data for predictive modeling [11]. Since the data quality strongly influences the models, most manual labor is spent on finding, integrating, and cleaning relevant datasets [98, 99]. Various data cleaning tools have been developed for reducing the manual effort [44, 59, 64, 83], gradually shifting the focus from hand-written scripts to configurations, rules, constraints, and techniques like active learning [25, 29, 50, 58, 63, 77]. Although these tools show valuable improvements, several severe usability and scalability issues remain. Existing work mostly considers data cleaning as a standalone task, independent of downstream ML applications [59, 63, 64, 83]. Previous studies have shown, however, that such oblivious data cleaning may not improve—or even degrade—model performance [58, 61, 68]. Instead, we can leverage the downstream ML applications as a signal, in terms of quantitative feedback, for selecting effective data cleaning strategies. Similar to the broader ISO 9001 quality management [3]—where products and services are monitored for their "fit-for-purpose" (fulfillment of defined requirements), the quality of ML models (e.g., accuracy) can evaluate the impact of data cleaning strategies.

Usability Challenges: In practice, data cleaning for ML faces several usability challenges. First, many tools strongly depend on user-provided inputs such as constraints, regular expressions, and dictionaries. While the experiments often look promising, such user inputs (provided upfront or via active learning) require significant domain expertise and time investments. Second, data cleaning algorithms and tools become increasingly specialized for particular types of problems or errors. This specialization is problematic because it ignores inter-dependencies among cleaning techniques, and it requires orchestrating, configuring, and tuning multiple tools. Examples are query processing with imputation [19] or deduplication [2], algorithms for missing value imputation [9, 102], entity resolution pipelines [39, 55, 81], or factor graphs for groups of constraints [83]. Together, there is still an unnecessarily large burden on the user.

Scalability Challenges: There are also unaddressed scalability challenges. First, except for special solutions for distributed data deduplication [13, 24], distributed data validation [89], and distributed rule-based cleaning [54], existing data cleaning tools are unable to enumerate and execute cleaning pipelines in a distributed manner. Relevant subproblems include task-parallelism for concurrent pipeline evaluation and data-parallel cleaning of large datasets that exceed single-node memory. Second, an increasing number of cleaning primitives and their parameters further increases the search space and thus, the need for parallelization.

Saga Design Principles: Learning from decades of work on data cleaning and related areas such as entity resolution and schema mapping[2], we aim—instead of full automation which is rather unrealistic—to simplify the mechanical aspects of data cleaning for ML. Our key design principles are *automatic pipeline enumeration* (provide effective means for enumerating and tuning pipelines and their parameters), *extensibility* (allow new types of cleaning primitives, ML applications, and

[2]Bernstein & Melnik: "Given the existence of all these tools, why is it still so labor-intensive to develop engineered mappings? To some extent, it is an unavoidable consequence of ambiguity [...] the specification of meaning is weak and the mapping must be precisely engineered, it seems hopeless to fully automate the process anytime soon." [7]

manual pipeline refinements), as well as *automatic plan generation* (generate efficient local or distributed runtime plans according to workload characteristics).

Contributions: Our primary contribution is SAGA, a scalable framework—fully integrated into open-source Apache SystemDS[3]—for finding the top-K, parameterized data cleaning pipelines for a given dataset, ML application, and optional user-provided constraints. In this context, our conceptual and technical contributions are:

- *Framework:* As framework foundations, Section 2 introduces SAGA's library of data cleaning primitives (which are implemented in linear algebra, allowing automatic parallelization), the overall problem formulation, and a clean API contract for top-K pipeline enumeration.
- *Pipeline Enumeration and Tuning Algorithms:* At the core, we introduce simple yet effective algorithms for logical pipeline enumeration (pipeline structure) in Section 3, and physical pipeline tuning (pipeline parameters) in Section 4. Further optimizations include pruning by traits of primitives (e.g., by monotonicity), and flattening nested parallelism.
- *Runtime Plan Generation:* From the perspective of scalable execution, we briefly describe alternative parallel plans, as well as compilation and runtime techniques for generating efficient local or distributed plans in Section 5.
- *Experiments:* Finally, we share the results of a broad experimental evaluation in Section 6, including real and synthetic data, different baseline cleaning tools, errors types and characteristics, ML algorithms, AutoML tools, and parallel plans.

2 SAGA FRAMEWORK

In this section, we summarize SAGA's cleaning primitives, formulate the problem of finding the top-K data cleaning pipelines for an ML application, and provide an overview of the API contract and resulting framework.

2.1 Data Cleaning Primitives

Over the last three years, we created SAGA's library of cleaning primitives, which implement—mostly known—data cleaning primitives as linear-algebra-based built-in functions. Table 1 shows selected primitives of different categories as well as their parameters and applicability to pruning by monotonicity. Related catalog information is managed as simple data frames, and thus, new primitives can be added seamlessly.

Selected Primitives: Besides data preparation (e.g., feature transformations, standardization, and dimensionality reduction) and string processing (e.g., stemming, typos, and swaps), our primitives address outliers, missing values, as well as class imbalance and/or flipped labels. First, for outlier removal, we leverage statistical primitives such as outlier removal by inter-quartile range, by standard deviation, and winsorization (based on quantiles), as well as DBSCAN [92] as a density-based clustering algorithm (remove points that are not density-reachable). Second, missing value imputation techniques include simple imputation of constants, last values (forward/backward filling), mean, and mode (most frequent item). More complex strategies are imputation by robust functional dependencies [33] (if an FD country→city applies to 90% of tuples, use it to clean the remaining 10%) and multivariate imputation by chained equations [102] (classification for categorical and regression for numerical features). Third, for handling labels we apply classical under-sampling, SMOTE [22] (synthetic oversampling), and tomekLink (under-sampling by dropping borderline points). Furthermore, we also allow label flipping and abstain (tuple removal) for instances where linear models mispredict with high confidence.

[3]Apache SystemDS [11]: http://apache.github.io/systemds/, reproducibility: https://github.com/damslab/reproducibility

Table 1. Saga's Data Cleaning Primitives.

	Category	Name	Params	Monotonic
1	Outliers	dbscan [92]	minPts, ϵ	
2		outlierByIQR	k	✓
3		outlierBySd	k	✓
4		winsorize	q_l, q_u	✓
5	MV Imputation	fillDefault	v	
6		fillForward		
7		imputeByFd [33]	P	✓
8		imputeByMean		
9		imputeByMedian		
10		mice [102]	i, P	
11	Data Preparation	encodeOneHot		
12		encodeFequency		
13		encodeDate		
14		normalize / scale		
15		pca / ppca	k	
16		weightOfEvidence		
17	Class Imbalance	smote [22]	N	
18		tomekLink		
19		underSampling	N	
20	Labels	flipLabels	P	✓
21		abstain	P	✓
22	String Handling	correctTypos		
23	(stage 0)	fixLength / fixType	**L** / **T**	✓
25		stemming / lowerCase		
27		swapValues		

Fit and Apply: Each primitive has a fit and apply function (e.g., `outlierByIQR` and `outlierByIQRApply`), where the former computes state such as the inter-quartile range (IQR = $q_3 - q_1$), and then calls the apply function to remove outliers outside the $[q_1 - k \cdot \text{IQR}, q_3 + k \cdot \text{IQR}]$ range. The fit functions are used during training, their state is kept, and used when calling the apply functions during model scoring/inference. Some primitives like `flipLabels` have no-op apply functions similar to dropout layers in DNNs.

Need for Optimization: There are several dependencies among cleaning primitives. For example, outlier methods can replace values directly, remove rows, or replace values by missing values. Thus, together outlier removal and missing value imputation are a form of error detection and correction. Accordingly, both the ordering of primitives and good parameters for these primitives are strongly data-dependent, which requires automatic optimization.

2.2 Problem Formulation

In order to provide users with the flexibility of manual refinements of cleaning pipelines and primitives, we focus on top-K cleaning pipeline enumeration. We define the problem as follows:

Notation: Let $(\mathbf{X}, \mathbf{Y})$ be a dataset composed of input features $\mathbf{X}$ ($n-1$ independent variables, categorical or numerical), and a target feature $\mathbf{Y}$ (class labels or continuous response). Each tuple $t_i \in (\mathbf{X}, \mathbf{Y})$ is a vector of frame cells $t_i = \{C_{i1}, C_{i2}, \ldots, C_{in}\}$. Furthermore, let $\omega(\mathbf{X}, \mathbf{Y})$ be an ML

model—including feature transformations—learned on $(\mathbf{X}, \mathbf{Y})$ that produces a certain training loss $\phi(\omega(\mathbf{X}, \mathbf{Y}))$ and test loss $\phi(\omega(\mathbf{X}', \mathbf{Y}'))$. We assume that some cells $C_{ij} \in (\mathbf{X}, \mathbf{Y})$ and $\mathbf{X}'$ are corrupted, wrong, or missing. Therefore, we seek to find a cleaning pipeline P that applied to $(\mathbf{X}, \mathbf{Y})$ before training and to $(\mathbf{X}', \mathbf{Y}')$ before scoring reduces the expected test loss. As a proxy for expected test loss (which is not seen by our cleaning framework), the training or validation[4] loss should be strictly improved $\phi(\omega(P(\mathbf{X}, \mathbf{Y}))) < \phi(\omega(\mathbf{X}, \mathbf{Y}))$. The target ML application is a tuple (ω, ϕ) composed of a specific regression, classification, or other learning task ω as well as a user-specific loss function ϕ (e.g., inverse accuracy, f1-measure, l2-norm, r-squared loss).

Logical and Physical Pipelines: A pipeline P is defined by its structure (primitives and their order) and its parameters (hyper-parameters of contained primitives). We explicitly distinguish between *logical pipelines* and *physical pipelines*. A logical pipelines only defines the structure and can be evaluated with default parameters. In contrast, a physical pipeline has the structure of its parent logical pipeline but assigns concrete hyper-parameter values.

Optimization Objective: We aim to find the top-K physical cleaning pipelines that satisfy the following optimization objective:

$$\begin{aligned} \mathcal{P} = \arg\min_{\mathcal{P} \in \mathcal{S}} \Sigma_{k=1}^{K} \phi\left(\omega(\mathcal{P}_k(\mathbf{X}, \mathbf{Y}), \Sigma_{\mathbf{c}})\right) \\ s.t.\ \ \forall k \in \mathbb{N}_{[1,K]} : \phi\left(\omega(\mathcal{P}_k(\mathbf{X}, \mathbf{Y}))\right) < \phi\left(\omega(\mathbf{X}, \mathbf{Y})\right) \end{aligned} \tag{1}$$

Given a dataset $(\mathbf{X}, \mathbf{Y})$, set of optional user constraints $\Sigma_{\mathbf{c}}$, a target ML application (ω, ϕ), and an integer K, the goal is to find the top-K cleaning pipelines $\mathcal{P}$ from the search space of all pipelines $\mathcal{S}$ with minimal total loss ϕ, ordered ascending by loss. In order to prevent unwanted redundancy, the returned top-K set must contain physical pipelines of distinct logical pipelines.

Search Space: A brute-force solution would enumerate every pipeline of the space $\mathcal{S}$, which requires training and evaluating $\phi(\omega(P(\mathbf{X}, \mathbf{Y})))$. The overall search space is huge and ∞ if we allow repeated cleaning primitives (unbounded pipeline length). Even the restricted search space—where every primitive appears at most once—is huge. Given p cleaning primitives with a single hyper-parameter and h parameter values each, the search space comprises

$$|\mathcal{S}| = \sum_{l=0}^{p} \binom{p}{l}! \cdot h^l = \sum_{l=0}^{p} \frac{p!}{(p-l)!} \cdot h^l \tag{2}$$

physical pipelines. For every pipeline length $l \in [0, p]$, we have a partial permutation p *permute* l, multiplied by the number of hyper-parameter configurations h^l. With $p = 25$ cleaning primitives (see Table 1) and $h = 10$, the number of pipelines is already $|\mathcal{S}| > 10^{50}$, which renders brute force enumeration infeasible.

2.3 API Contract and Saga Overview

Fitting and Applying Pipelines: Addressing the large search space requires efficient pipeline enumeration. An often overlooked aspect of such enumeration algorithms is the concrete user API contract. We define a simple API with three functions:

- `topk_cleaning()` takes the training data, metadata, constraints, and target ML application; performs top-K cleaning pipeline enumeration; and returns the top-K pipelines, their hyper-parameter values and validation scores.
- `fit_pipeline()` takes a selected or refined pipeline (with fixed primitives and hyper-parameters) and fits this pipeline on the new data, returning the internal state (e.g., quantiles for `outlierByIQR`) and cleaned data.

[4]Similar to traditional hyper-parameter tuning of ML models, we further split the train data into train and validation sets for evaluating different cleaning pipelines.

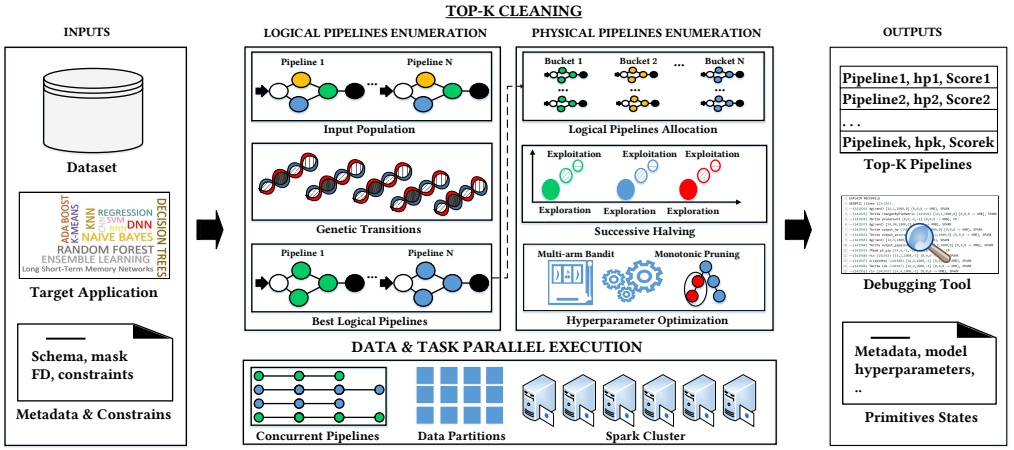


Fig. 1. Overview of SAGA Framework (Top-K Cleaning).

- `apply_pipeline()` transforms new incoming data for scoring with the given pipeline and read-only state of primitives.

This separation is important for moving top-K enumeration and training outside the critical path, and allowing manual refinements.

Framework Overview: Figure 1 gives an overview of our top-K cleaning pipeline enumeration. Besides the input data and ML target application, users may provide constraints for a more directed search, where we leverage functional dependencies, feature string lengths, and feature types (categorical or numerical). SAGA then performs all string processing and feature transformations before any cleaning on matrices. Given an input data frame and metadata, we handle typos, fix invalid values, perform stemming and normalization, and apply feature transformations. String manipulations are efficiently executed via second-order `map` functions (e.g., `map(FS,"s->s.length()")`) for which we internally perform code generation. Inspired by the optimization of data-processing workflows [27, 93, 94], and AutoML tools [71, 93, 97], we devise an effective multi-level optimization strategy of logical and physical pipelines. First, SAGA uses an evolutionary algorithm for finding the promising sequences of cleaning primitives with their default hyper-parameters (logical pipelines). Second, we adopt Hyperband [60] for tuning hyper-parameters of these logical pipelines (physical pipelines). Having implemented this top-K cleaning pipelines framework in linear algebra, allows automatically generating hybrid task- and data-parallel runtime plans of local and distributed operations. Finally, we return the top-K physical pipelines, their hyper-parameters, and ϕ-scores. SAGA also provides detailed logs of the enumeration process for debugging.

Avoiding Over-fitting: In contrast to ML-agnostic data cleaning, data cleaning for ML may be more prone to over-fitting the target ML application. We address this danger with (1) k-fold cross validation[5] for obtaining reliable scores of cleaning pipelines and models, (2) evaluating logical pipelines (structural aspects) only with default parameters, (3) starting logical pipeline enumeration from simple 1-primitive pipelines, and (4) top-K pipeline enumeration where users can modify and tune the most promising pipelines (balancing cleaning performance and pipeline complexity). Similar to the pruning of decision trees after training, selecting robust cleaning pipelines in a semi-automatic manner is interesting future work.

[5]Related work on wrapper-methods for feature selection [70] and similar approaches also tackles the danger of over-fitting primarily with cross validation.

3 LOGICAL PIPELINE ENUMERATION

We aim to find a set of candidate logical pipelines, which serve as good starting points for physical pipeline tuning. To this end, we devise a simple—yet empirically very effective—evolutionary algorithm. This algorithm aligns very well with our goal of automating the mechanical aspects of trial-and-error data cleaning, where data scientists iteratively apply combinations of data repair routines and evaluating their effectiveness on the target ML application.

3.1 Pipeline Construction

A key component of our evolutionary algorithm is the construction of new logical pipelines. We start from a set of seed pipelines $\mathcal{P}$ (by default, the primitives from Table 1 as 1-step pipelines) and then refine them via genetic transitions. The primitives for seed pipelines are defined in an extensible configuration file. We do not restrict the search space, allowing cleaning primitives to appear multiple times in a pipeline. This approach causes an unbounded search space, but enables pipelines such as missing value imputation, followed by outlier removal (which state is influenced by imputed values), and missing value imputation for correcting dropped outliers.

Genetic Transitions: From the fittest candidate pipelines, we then derive a population of new pipelines. Inspired by work on optimizing ETL workflows [94], we apply cleaning-specific transformations for refining these pipelines. In detail, we randomly pick good pipelines, and then apply a random transition to each candidate pipeline:

- **Addition:** Add a random primitive at a random position in $[1, l+1]$ to the candidate pipeline.
- **Crossover:** Concatenate variable (non-overlapping) parts of two random candidate pipelines.
- **Mutation:** Swap the order of two existing primitives at random but non-equal positions in a candidate pipeline.
- **Removal:** Remove a primitive (at a random position in $[1, l]$) from the candidate pipeline.

Additional fine-tuning includes a weighted randomization (i.e., sample selection) of candidate pipelines by their observed loss, and expected runtime of primitives.

Pipeline Dependencies: For the sake of simplicity and extensibility—by new, user-provided cleaning primitives—our evolutionary algorithm does not exploit detailed dependencies among individual cleaning primitives for a more directed search. However, we keep track of duplicates (avoid already executed pipelines), perform pruning for known categories of cleaning primitives (e.g., no MV imputation primitives directly on input data without missing values), and perform generic pruning by monotonicity (see Section 4 for details).

3.2 Enumeration Algorithm

Algorithm 1 summarizes our logical pipeline enumeration approach. The inputs are the dataset $(\mathbf{X}, \mathbf{Y})$, the seed pipeline population $\mathcal{P}$, the loss on the dirty dataset $\mathcal{D}$, a target absolute loss improvement e, the target ML application (ω, ϕ), and a maximum number of iterations max_iter (i.e., evaluated generations).

Scoring Pipelines (Line 3): In each iteration, we evaluate the scores of the population of pipelines $\mathcal{P}$ via the target ML application, sort the pipelines ascending by loss (best first), and perform successive halving (select the top-half pipeline population, halving ratio $\eta = 2$). The dynamically composed pipelines are represented as frames and then evaluated primitive-by-primitive via second-order methods.

Applying Transitions (Line 11): From the retained best pipelines, the new population $\mathcal{P}'$ is generated. We first copy over the good pipelines into the output (Line 8), and then derive new pipelines until the expected population size (by default 16 logical pipelines) is reached. Once the new population $\mathcal{P}'$ is complete, the next iteration starts and evaluates this new population.

Algorithm 1 Logical Pipeline Optimization

Input: (X, Y), $\mathcal{P}$, $\mathcal{D}$, e, (ω, ϕ), max_iter
Output: Best logical pipelines **L**

1: **L** $\leftarrow frame(0,0,0)$; $iter \leftarrow 1$; $loss\_list()$ *// initialize variables*
2: **while** (!converged & iter < maxi) **do**
3: $\mathcal{P}, scores \leftarrow$ evaluateAndOrder(**X**, **Y**, $\mathcal{P}$, $descr = F, \eta = 2$)
4: putLoss($loss\_list, scores[1]$)
5: $loss \leftarrow$ getLoss($loss\_list, start = 1, end = 3$)
6: $converged \leftarrow$ isAllDuplicate($loss$)&$loss[1] < (\mathcal{D}$ - $e)$
7: $n \leftarrow nrow(\mathcal{P})$
8: **L** $\leftarrow$ append(**L**, $\mathcal{P}$)
9: $children \leftarrow frame(rows = n)$
10: **while** $k < n$ & (!converged) **do**
11: $c1 \leftarrow$ applyTransition($top = \mathcal{P}[k], prob = random()$)
12: $\mathcal{P}' \leftarrow$ append($\mathcal{P}', c1$)
13: $k \leftarrow k + 1$
14: $\mathcal{P} \leftarrow \mathcal{P}'$
15: $iter \leftarrow iter + 1$
16: **return** **L**

Termination (Line 6): The algorithm terminates if at least one of the following conditions are met. First, if the previous three iterations did not yield any loss reduction, or the best loss already reached the target loss $\mathcal{D} - e$, we declare convergence. Second, the algorithm also terminates if the maximum number of iterations is reached, providing an upper bound on the number of evaluated populations. The algorithm finally returns an output frame of the best candidate pipelines from all iterations (Lines 8 and 16).

4 PHYSICAL PIPELINE TUNING

At the last level of multi-level optimization, we perform physical pipeline tuning for optimizing the hyper-parameters of primitives in a logical pipeline via a technique similar to Hyperband [60] (as a generalization of multi-armed bandits and random search) as well as tailor-made pruning and parallelization strategies.

4.1 Tuning Approach

Hyperband Background: Hyperband [60] is an extension of the successive halving method [46] for adaptive resource allocation of random search. Successive halving evaluates a set of configurations with R resources, keeps the best 50% of configurations and doubles their resources until only one configuration remains. In order to avoid the high-influence parameter of the number of initial configurations, Hyperband introduces the notion of brackets and calls successive halving in each bracket with different numbers of starting configurations and maximum number of successive halving iterations. The spectrum of brackets and the principled early-stopping strategy provide a good tradeoff of explorations and exploitation, and has shown substantial speedups over hyper-parameter tuning with Bayesian Optimization on selected benchmarks.

Resource Allocation: Similar to Hyperband brackets, we create buckets with different resource configurations for balancing exploration and exploitation. The resource value $\mathcal{R}$—a Saga hyper-parameter—controls the generation of physical pipelines per bucket. In contrast to Hyperband, where the number of configurations per bucket (logical pipelines here) decreases exponentially,

Table 2. Hyperband Buckets with n_i Pipelines and r_i Resources per Iteration i (with $\mathcal{R} = 50$, $\eta = 2$).

i	s = 4		s = 3		s = 2		s = 1		s = 0	
	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i
0	24	1	24	2	24	7	24	20	24	50
1	12	2	12	5	12	15	12	40		
2	6	4	6	10	6	30				
3	3	8	3	20						
4	1	16								
#Pipelines	112		228		528		960		1200	

Algorithm 2 Hyperband for Physical Pipelines Enumeration

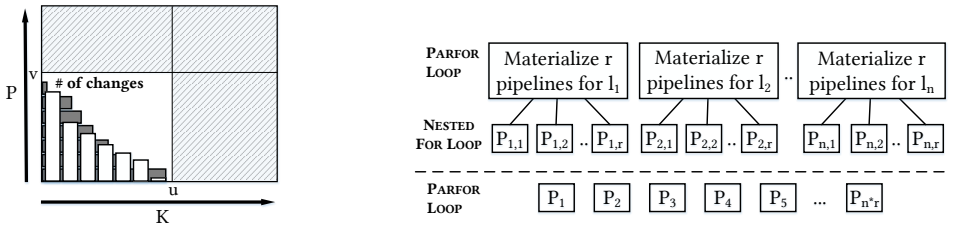
Input: R, η (default $\eta = 2$), $\mathbf{L}$, $(\mathbf{X}, \mathbf{Y})$, (ω, ϕ), config
Output: $topK$

1: $s_{\max} \leftarrow \lfloor \log_\eta(R) \rfloor$, $B \leftarrow (\max(s_{\max} + 1))R$ *// initialize variables*
2: $n \leftarrow \lceil (nrow(\mathbf{L})/s_{max}) \rceil$
3: **for** $s \in \{s_{\max}, ..., 0\}$ **do**
4: $\quad r \leftarrow W * R\eta^{-s}$
5: $\quad$ **for** $i \in \{0, ..., s\}$ **do**
6: $\quad\quad n_i \leftarrow n\eta^{-i}, r_i \leftarrow r\eta^i$;
7: $\quad\quad loss \leftarrow$ RUNWITHHYPERPARAM$((\mathbf{X}, \mathbf{Y}), (\omega, \phi, L, r_i, \text{config})$
8: $\quad\quad \mathbf{L} \leftarrow$ TOPK$(\mathbf{L}, loss, n_i/\eta)$
9: **return** $topK$

we assign the same number of logical pipelines to each bucket to enable exploration despite small populations. For example, in Table 2, we have 120 pipelines, $s\_max = 5$ buckets, and $\mathcal{R} = 50$ resources. We sort the logical pipelines by their loss, and assign 24 pipelines to each bucket s, with the most-promising pipelines in the last bucket ($s = 0$ with high starting resources). We then use the scores of the logical pipelines for a weighted distribution of resources to the buckets. The weights for each bucket are computed as follows:

$$\begin{aligned} \mathbf{weights} &= \text{seq}(1/s\_max, s\_max, 1/s\_max) \\ r &= \mathcal{R} \cdot \mathbf{weights}[(s\_max - s) + 1] \cdot \eta^{-s} \end{aligned} \tag{3}$$

Successive Halving: Per bucket, we then apply—as shown in Table 2—Hyperband's successive halving (select top-half of pipelines and increase their resources), where i is the current iteration and $\eta = 2$ is the rate of successive halving. In each iteration, we evaluate n_i logical pipelines with r_i resources (hyper-parameter combinations) and thus, $n_i \cdot r_i$ physical pipelines, and then sort these pipelines by their scores. We keep the upper half of well-performing pipelines and discard the rest. In the next iteration, the remaining pipelines are evaluated with $r + r'$ resources (i.e., more resources of physical pipelines for promising logical pipelines). SAGA keeps the top-K pipelines from each bucket to create one final set of $(s\_max \cdot K)$ pipelines, which is later capped to the top-K pipelines and filtered by the dirty score $\mathcal{D}$ (to only return useful pipelines to the user). To avoid over-fitting, by default, we evaluate the pipelines using 3-fold cross validation. Algorithm 2 provides a high-level overview of bucket creation and pipeline evaluation with input resources (R), successive halving rate (η), logical pipelines ($\mathbf{L}$), features and labels $(\mathbf{X}, \mathbf{Y})$, ML target application (ω, ϕ), and hyper-parameter grid (parameters and value ranges) for all primitives (config).



(a) Sub-space Pruning (b) Nested and Flattened Parallelism

Fig. 2. Pruning and Flattening Techniques.

4.2 Pruning by Monotonicity

Understanding the traits of cleaning primitives allows pruning dominated configurations. Primitives like `outlierByIQR`, `outlierBySd`, and `imputeByFd` have threshold parameters for value removal and imputation. Some of these parameters exhibit monotonic behavior (see Table 1, where we marked the applicable primitives). For example, `outlierByIQR` computes the interquartile range $\text{IQR} = q_3 - q_1$ and removes all values outside $[q_1 - k \cdot \text{IQR}, q_3 + k \cdot \text{IQR}]$. Given a fixed dataset, if a certain value k does not remove any values, then any values $k' \geq k$ will not remove values either. Hence, all parameter combinations with such k' can be safely pruned, which also motives ordering parameter configurations to maximize pruning. More formally, let $\mathcal{P}_q = \{p_1, p_2, \ldots, p_l\}$ be a pipeline of l primitives with one monotonic hyper-parameter $h(p_i)$ each. If we have observed that $h(p_i) = k$ does not modify any values—and assuming a homogeneous coverage of the remaining parameter space—we can safely prune $h(p_i) \geq k$ and all combinations thereof with $h(p_j)$ for $j > i$. Figure 2(a) shows an example of two monotonic hyper-parameters K and P (order is irrelevant here). If no changes are made for $P = v$ or $K = u$, then all combination with values $P \geq v$ or $K \geq u$ can be pruned. Interesting future work for logical and physical pipeline tuning include error-driven slice selection [26, 73, 86, 100], and lineage-based reuse of intermediates [76, 103].

4.3 Flattening Nested Parallelism

Pipeline tuning with Hyperband creates deep nested loops, each with few iterations. We ensure effective local and distributed task-parallelism via a simple, yet effective, flattening of nested parallelism [40] using parallel for loops (ParFor).

ParFor Background: Task-parallel ParFor loops [12, 40] have been successfully employed for large-scale machine learning and a variety of tasks such as cross-validation, hyper-parameter tuning, and ensemble learning. Given a `parfor(i in beg:end){<body>}` construct, an optimizing compiler performs dependency analysis to verify that there are no loop-carried dependencies, and then executes the `body` with a chosen parallelization strategy (from a spectrum of parallel plans, see Section 5) for every iteration.

Flattening Approach: Given nested ParFor loops of independent, expensive pipeline evaluations as well as additional state at intermediate levels, our flattening approach splits these computations into two phases. First, we perform an identical loop nest for materializing all configurations in matrices, frames, or lists. Second, we evaluate all configurations through a single, local or distributed ParFor loop (see Algorithm 3). This approach exposes more independent tasks and avoids unnecessary synchronization barriers—which simplifies loop optimization and task scheduling—and thus, improves the utilization of available parallelism, and reduces overheads for distributed setup and result aggregation. The cost for materializing the configurations is negligible here.

Algorithm 3 Flattened RUNWITHHYPERPARAM

Input: r, $\mathbf{L}$, $(\mathbf{X}, \mathbf{Y})$, (ω, ϕ), config
Output: *loss*

```
1: pipelines_with_configs ← [], loss ← []
2: while (i < nrow(L)) do
3:    param_config ← GETPARAMCONFIGS(l, r, config)          // r · l rows
4:    APPEND(pipelines_with_configs, l, param_config)
5: for i in nrow(pipelines_with_configs) do                 // parallel for
6:    score ← EVALUATE(pipelines_with_configs[i], X, Y), (ω, ϕ))
7:    APPEND(loss, score)
8: return loss
```

Flattening Example: Logical pipelines are grouped by their scores into buckets, and for each bucket, we perform successive halving. Nested ParFor loops parallelize over buckets and subsequently—within each bucket—over logical pipelines, and resources per logical pipeline (i.e., physical pipelines). Instead, the flattened ParFor—as shown in Figure 2—executes super-steps of all resources across all buckets with lockstep successive halving (five jobs across buckets in Table 2).

5 PARALLELIZATION STRATEGIES

Opportunities: Optimizing data cleaning pipelines has numerous parallelization opportunities. First, frame and matrix operations can exploit multi-threaded or distributed data parallelism. Second, several cleaning primitives (see Table 1) have intrinsic opportunities for data-or task parallelism (e.g., `mice` repeatedly trains classifiers and/or regression models). Third, logical and physical pipeline tuning forms a hierarchy of populations, hyper-parameter tuning of pipelines, as well as cross-validation and ML model training. Given these parallelization opportunities, alternative parallelization strategies can be utilized. A key design principle of SAGA is to avoid hard-coding such decisions at script level for adaptation to data, workload, and cluster characteristics.

Task-parallel Plans: We utilize task-parallel for-loops (ParFor) for coarse-grained parallelization that avoids unnecessary synchronization barriers of individual operations. Plan alternatives include:

- *ParFor in Cleaning Primitives:* A baseline approach applies task-parallelism only at the last level of cleaning primitives.
- *Local Top-Level ParFor:* Executing loops for logical and physical pipeline evaluation as local, multi-threaded loops, which works for all data sizes with low latency overhead.
- *Distributed Top-Level ParFor:* In contrast to *Local Top-Level*, this top-level ParFor can be executed as a single distributed job, potentially utilizing a higher degree of parallelism, but this approach requires that inputs and outputs fit into task memory.
- *Adaptive ParFor Hierarchies:* Combining the advantages of other plans, we utilize adaptive ParFor hierarchies, where all applicable levels include ParFor loops, and during runtime, the ParFor optimizer compiles the most appropriate parallel execution plan.

We extended the underlying SystemDS for pipeline enumeration in distributed ParFors loops, specifically with on-demand function loading and dynamic compilation in Spark executors, without interference while still reusing functions across tasks and jobs.

Data-parallel Plans: End-to-end ML pipelines can be compiled into hybrid runtime plans of local and distributed, data-parallel operations. Cleaning pipeline enumeration seamlessly leverages these capabilities, but—except for flattening at script level (which is always beneficial)—we do not yet specifically optimize such plans. An interesting direction for future work is the adaptation of

Table 3. Dataset Characteristics (excluding class labels).

Dataset	#Rows	#Cols	#Class	ω	#Cat	#Num	$\mathcal{D}$	$\mathcal{G}$
Animal [21]	26,729	8	5	class	8	0	0.62	Y
EEG [85]	14,980	14	2	class	0	14	0.63	Y
Movie [61]	9,329	7	2	class	2	5	0.74	Y
Nashville [30]	211,945	22	2	class	14	8	0.76	N
Puma [31, 53]	4,756	8	2	class	0	8	0.54	Y
Titanic [48]	891	11	2	class	5	6	0.76	Y
Cancer [32]	3,048	33	-	Reg	2	31	0.10	N
Housing [47]	1,460	79	-	Reg	46	33	0.67	N

Table 4. Summary of Errors (SE.. string errors, MV .. missing values, O .. outliers, VFD .. violated functional dependencies, IM .. imbalanced classes, NL .. noisy labels).

Dataset	SE	MV	O	VFD	IM	NL
Animal	208,436	21,322	2,083	6,470	10,572	0
EEG	0	0	209	0	1,537	854
Movie	13,828	0	937	0	2,531	129
Nashville	2,529,412	213,650	66,059	31,680	116,941	45,387
Puma	0	7,646	306	0	39	185
Titanic	1,180	866	97	743	207	0
Cancer	6,093	3,046	1,150	0	N/A	N/A
Housing	63,135	6,965	1974	124	N/A	N/A

ideas from scan sharing in hyper-parameter tuning [97], and model-hopper parallelism [66, 67] to the specifics of cleaning pipelines (e.g., partial data modifications, and pruning by monotonicity).

6 EXPERIMENTS

We study SAGA in comparison with state-of-the-art data cleaning and AutoML tools regarding the accuracy of ML applications, various micro benchmarks, and parallelization strategies.

6.1 Experimental Setting

Hardware Setup: We ran all experiments on a 1+6 node cluster, each node having an AMD EPYC 7302 CPU @ 3.0-3.3 GHz (16 physical/32 virtual cores), and 128 GB DDR4 RAM (peak performance is 768 GFLOP/s, 183.2 GB/s). The software stack comprises Ubuntu 20.04.1, Apache Hadoop 3.3.1, and Apache Spark 3.2.0. SAGA uses OpenJDK 11.0.13 with 110 GB max and initial JVM heap sizes.

ML Applications: The ML target applications are provided as linear-algebra-based UDFs. We study different classification (multinomial logistic regression, multi-class support vector machines, and decision tree) as well as regression (direct-solve ordinary least squares, and decision tree) models. All datasets are split into 70% train and 30% test. As the train/test splits are from the same dataset, they follow the same distributions. The test data labels are never passed to the cleaning frameworks, but solely used for evaluation. We further use 3-fold cross-validation on the train set, and report accuracy for classification and R^2 for regression tasks.

Datasets: We used eight real datasets from ML competitions and the UCI repository. Table 3 summaries the data characteristics of these datasets including dimensions, number of classes, target application ω, number of categorical and numeric columns, dirty accuracy $\mathcal{D}$, and the availability of

ground truth $\mathcal{G}$, which refers to "clean" accuracy as known from previous work because no actual ground truth exist. These datasets include string errors (e.g., typos, value swaps, out-of-domain values, case mismatch), missing values, outliers, violations of functional dependencies, imbalanced classes, and noisy labels. Table 4 shows the error distributions of all datasets. Given the absence of ground truth, we computed these counts in a best effort manner (e.g., missing values from NULL/NaN, outliers outside three standard deviations). Although, the Movie dataset has no missing values, the test set does because it contains strings non-existing in the train set.

6.2 End-to-End Data Cleaning for ML

We investigate the end-to-end accuracy improvements of data cleaning on all datasets and ML applications. In detail, we compare SAGA with state-of-the-art data cleaning frameworks, evaluate the impact of different models, and compare with AutoML tools. Here, we use the SAGA default parameters of 15 iterations of the evolutionary algorithm, seed population size $|\mathcal{P}| = 16$, and $\mathcal{R} = 50$ Hyperband resources. With this configuration, SAGA evaluates $\approx$ 10,000 physical pipelines (cleaning and cross-validated model training) per dataset.

Baselines: We compare SAGA with the following baselines, which all use the same ML target application. SAGA differentiates by its holistic tuning and the generation of scalable execution plans.

- **Basic Heuristics:** We hand-crafted several heuristic, commonly-used cleaning pipelines (e.g., missing value imputation by mean, outlier removal via winsorizing [0.05, 0.95]).
- **BoostClean [57]:** We compare BoostClean with b=5 (best setting mentioned in the paper) and report the accuracy on the train set. Additionally, we use the best ensemble and apply it for obtaining the test accuracy as well.
- **HoloClean [44, 83]:** HoloClean uses a factor graph for detecting and repairing errors based on denial constraints. For each dataset, we spent three hours to define meaningful constraints, and tuned the hyper-parameters.
- **Raha-Baran [63, 64]:** Similarly, Raha & Baran also perform error detection and correction without considering the target application. Raha & Baran (and similarly but slightly worse, ActiveClean [58]) use active leaning for predicting the correct and incorrect values.
- **Learn2Clean [8]:** The recent Learn2Clean uses reinforcement learning (Q-learning) to find the best cleaning strategy for a configurable ML target application, without any other hyper-parameters or extension hooks.

Results: Table 5 summarizes the train/test accuracy of the baselines and SAGA. BoostClean only supports classification and no regression (N/A). To remove the effects of randomization, each test was repeated three times and we report the average accuracy. The *basic heuristics* already work quite well for some datasets. In Movies, missing values exist only in the test split, so the heuristic pipeline does not change the training data but imputes missing values for testing using the training mean. Housing contains outliers that are winsorized, which yields an improved R^2. *BoostClean* shows generally mixed results. When changing the underlying ML application from the default decision tree (more robust to errors) to multinomial logistic regression, accuracy was negatively impacted. SAGA outperforms BoostClean because BoostClean tries to repair every misclassified tuple with statistical imputation, whereas SAGA uses a mix of statistical, rule-based, and ML-based primitives. Additionally, BoostClean also did not perform very well on the Animal dataset where all the features are categorical. For *HoloClean* and *Raha-Baran*, we observe similar accuracy on all datasets, except HoloClean runs out-of-memory (OOM) on Nashville, Raha-Baran times out (TO) on Nashville due to swapping, and HoloClean times out on Housing despite dedicated hyper-parameter tuning. For Raha-Baran, we observe that the system detects the errors correctly (Raha's F_1 score greater than 0.9), but the error correction does not improve overall accuracy. BoostClean

Table 5. Train and Test Accuracy Linear Models
(Multinomial Logistic Regression for Classification and Ordinary Least Squares for Regression*).

Dataset	Dirty		Basic Heuristics		BoostClean		HoloClean		Raha-Baran		Learn2Clean		Saga	
	train	test	train	test	train	test	train	test	train	test	train	test	train	test
Animal	0.70	0.62	0.70	0.69	0.34	0.33	TO	TO	0.34	0.60	N/A	N/A	**0.86**	0.86
EEG	0.64	0.63	0.65	0.65	0.64	0.63	0.55	0.64	0.55	0.63	0.68	0.67	**0.69**	0.68
Movie	0.75	0.74	0.75	0.84	0.69	0.70	0.63	0.62	0.53	0.64	0.76	0.76	0.78	**0.85**
Nashville	0.76	0.76	0.79	0.78	0.79	0.79	OOM	OOM	TO	TO	0.79	0.79	**0.80**	0.80
Puma	0.55	0.54	0.54	0.56	0.55	**0.57**	0.56	0.54	0.57	0.47	0.57	0.51	0.57	0.57
Titanic	0.79	0.76	0.78	0.65	0.80	0.81	0.66	0.78	0.66	0.80	0.78	0.73	0.81	**0.82**
Cancer*	0.43	0.10	0.43	0.45	N/A	N/A	0.16	0.16	0.16	0.16	**0.62**	0.61	0.51	0.52
Housing*	0.67	0.67	0.81	0.85	N/A	N/A	TO	TO	0.65	0.67	0.84	**0.89**	0.83	0.87

Table 6. Train and Test Accuracy Tree-based Models
(Decision Trees with Impurity Measure Gini-Index for Classification and Squared Loss for Regression).

Dataset	Dirty		Basic Heuristics		BoostClean		HoloClean		Raha-Baran		Learn2Clean		Saga	
	train	test	train	test	train	test	train	test	train	test	train	test	train	test
Animal	0.64	0.63	0.64	0.65	0.32	0.17	TO	TO	0.42	0.1	N/A	N/A	0.83	**0.83**
EEG	0.58	0.55	0.76	0.80	0.80	0.81	0.58	0.55	0.59	0.55	0.80	0.73	**0.84**	0.74
Movie	0.65	0.66	0.65	0.65	0.70	0.69	0.65	0.66	0.66	0.67	0.76	**0.77***	0.67	0.67
Nashville	0.79	0.78	0.80	0.80	0.75	0.77	OOM	OOM	TO	TO	0.80	0.80	0.80	0.80
Puma	0.51	0.53	0.51	0.55	0.53	0.54	0.52	0.51	0.52	0.52	0.54	0.53	0.55	**0.56**
Titanic	0.76	0.57	0.69	0.72	0.79	0.72	0.63	0.68	0.68	0.55	0.80	0.79	**0.81**	0.79
Cancer	0.34	0.37	0.37	0.39	N/A	N/A	0.34	0.37	0.34	0.37	0.39	0.37	0.36	0.39
Housing	0.64	0.74	0.65	0.73	N/A	N/A	TO	TO	0.65	0.74	0.70	**0.78**	0.69	0.76

performs generally better than HoloClean and Raha-Baran, which highlights the importance of optimization for target ML applications. *Learn2Clean* shows overall good results but failed on the Animal dataset—which has only categorical features—because the design of Learn2Clean requires at least one continuous column. We observed that primitives used in Learn2Clean proactively remove the rows from train and test dataset to get rid of outliers and invalid patterns. This approach shows improved performance on Cancer and Housing dataset but deleted 1,418 train and 595 test samples from Cancer as well as 543 train and 238 test samples from Housing. Ignoring this behavior, Learn2Clean performed better than all the previous baselines showing effectiveness of using ML signals for data cleaning but loses information by deleting the samples instead of fixing them. Furthermore, Saga consistently outperforms the basic and state-of-the-art baselines, with substantial (greater than 10%) improvements on some datasets without deleting any samples from test data and using sampling only on train datasets when under-sampling the majority class.

Generalization: In order to validate our results, we further evaluated all baselines and datasets with decision trees (table 6) and Multi-class SVM (not shown). For decision trees, we observe the same trend as with linear models for all baselines, except the Movie dataset. For Movie dataset Learn2Clean removed 2,670 rows from the train and 1,123 rows test data, yielding a big accuracy improvement. For Saga, we only remove rows from training dataset whenever a sample is required and do not remove any instances from the test data. For Multi-class SVM, we obtain similar results to multinomial regression, with differences of ±0.03. BoostClean shows one outlier where the accuracy on Titanic significantly drops from 0.81 to 0.66. We attribute this incident the strong dependence of BoostClean on the underlying target ML application. Furthermore, we found cases, where enumerated pipelines with separate primitives for outlier removal and missing value imputation helped generalization. For example, the Movies dataset is challenging because missing values only exist in the test set. Saga yields good generalization on the test set because it selected outlier

Table 7. Auto-Sklearn 1.0 Accuracy (on Dirty Data left and SAGA's Data right).

Dataset	100 mins		200 mins		100 mins		200 mins	
	Train	Test	Train	Test	Train	Test	Train	Test
Animal	0.87	0.87	0.86	0.86	0.86	0.86	**0.88**	0.86
EEG	0.96	0.97	0.96	0.97	0.96	0.97	**0.97**	0.97
Puma	0.58	0.56	0.58	0.56	**0.59**	0.56	**0.60**	0.55
Titanic	0.84	0.80	0.84	0.81	**0.94**	**0.83**	**0.95**	**0.82**

Table 8. Auto-Sklearn 2.0 Accuracy (on Dirty Data left and SAGA's Data right).

Dataset	100 mins		200 mins		100 mins		200 mins	
	Train	Test	Train	Test	Train	Test	Train	Test
Animal	NA	0.4	NA	0.4	NA	0.35	NA	0.35
EEG	0.78	0.84	0.78	0.84	0.69	0.71	**0.80**	**0.87**
Puma	0.58	0.55	0.58	0.57	**0.59**	**0.56**	**0.59**	0.55
Titanic	0.79	0.80	0.81	0.80	**0.91**	**0.81**	**0.91**	0.80

removal primitives (which dropped certain values), and thus, rendered missing value imputation during training beneficial (which then handled real missing values during scoring).

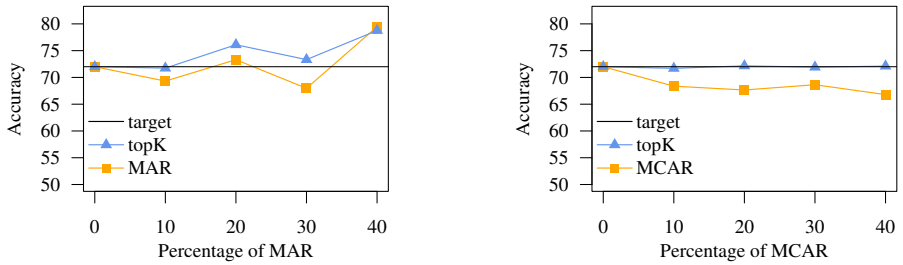
AutoML Comparison: Some AutoML tools also include basic data preparation but do not support data cleaning primitives. Table 7 (left) compares the accuracy of Auto-Sklearn [37]—as a representative AutoML tool—with different time budgets, which allows the application of a variety of feature transformations and ML models. Auto-Sklearn generally yields very good accuracy, even on dirty data. However, with data cleaned by SAGA, the accuracy improved up to 10% on instances like Titanic Table 7 (right). This result—together with another recent study [69]—motivates the integration of data cleaning into AutoML tools. For datasets with value-swap errors, Auto-Sklearn even crashed due to mixed data in categorical columns. We repeated these experiments with AutoSklearn 2.0 (experimental) [36], which builds data profiles and—similar to SAGA—uses a combination of Bayesian Optimization and Hyperband for hyper-parameter optimization and model selection. Table 8 (right) shows that Auto-Sklearn 2.0 also benefits from clean data, but overall Auto-Sklearn 1.0 still yield better accuracy.

6.3 Micro Benchmarks

In a second series of experiments, we evaluate SAGA on a variety of micro benchmarks. We use the Chicago FoodInspection dataset (39,814 × 9) [42], and leverage Jenga [91] for generating errors (data corruptions of specific types) in the train and test sets. As pre-processing steps, we drop all rows with NULL values, as well as filter the two risk categories high and medium, and the five most frequent facilities. Furthermore, we remove the institution name, constant city and state, and date columns, which resulted in a high-quality input dataset as a basis for systematic corruption.

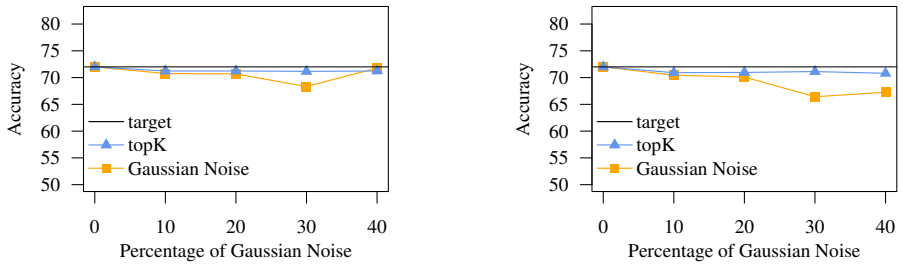
Error Generation: We vary the percentage of errors from 0% to 40% in all columns, and generate the following types of errors:

- *Missing Values:* We generate both Missing at Random (MAR) and Missing Completely at Random (MCAR).
- *Noise:* We add Gaussian noise to the Longitude and Latitude with $\mu = 0$ and $\sigma \in [1, 5]$.
- *Value Swaps:* We swap values among License Number, Facility Type, Zip, and Address.
- *Scaling Errors:* We scale the Longitude and Latitude.



(a) Missing at Random (b) Missing Completely at Random

Fig. 3. Missing Values.



(a) Sampled = MAR (b) Sampled = MCAR

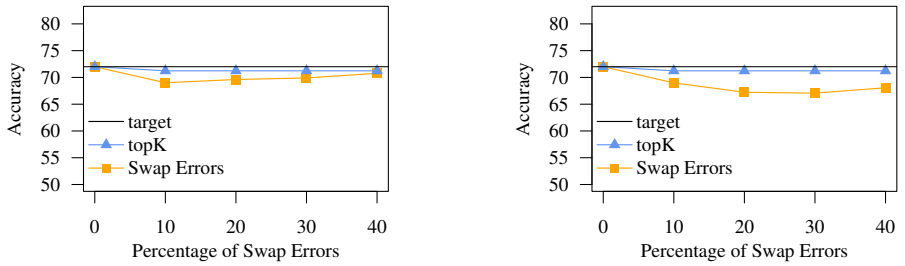
Fig. 4. Gaussian Noise.

- *Violated FDs:* We introduce violations of functional dependencies in Zip and Address.
- *Mixed Errors:* We randomly pick columns for different types of error generation (see above).

Finally, we divide the dataset into 70/30% train/test splits, run SAGA's top-K cleaning pipeline enumeration on the train split, apply the top-1 pipeline with multinomial logistic regression on the test split, and report the test accuracy on the ground-truth (target baseline), dirty data (yellow squares) and cleaned data (blue triangles).

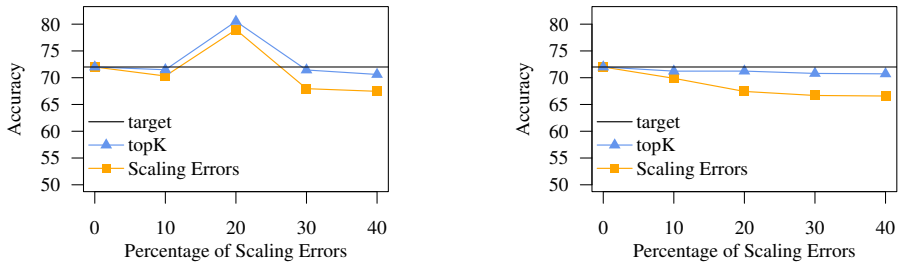
Missing Values: We experimented with two categories of missing data: Missing at Random (MAR, reconstructable from existing data) and Missing Completely at Random (MCAR). Figure 3 shows the results, where with increasing error rate, the model accuracy drops below the ground truth. In contrast, with our top-K cleaning pipeline enumeration, the clean accuracy continuously outperforms the ground-truth accuracy. The imputed values seems to act as a form of regularization. As SAGA continuously optimizes for signals of the downstream ML application, we get good generalization. The results in Figures 3(a) and 3(b) further indicate a major difference in behavior between MAR and MCAR. For MAR, both with and without cleaning, we get fluctuating results, whereas for MCAR, we see a very smooth behavior. A possible reason could be that synthetic error generation distorts the columns correlations, and thus, affects the reconstruction in ways, disproportionate to the error rate. Most importantly, SAGA shows very consistent improvements over dirty data for both types of missing values.

Noise: Gaussian Noise is only added to two of nine columns, and thus, we do not observe significant distortions in Figure 4. However, SAGA yields robust performance, comparable to the ground-truth, for all noise levels as well. Interestingly, with increasing error rate, at some point, the model and its predictions are dominated by the Gaussian noise, and thus, accuracy improves even on dirty data. We observe again larger improvements for MCAR-sampled error locations.



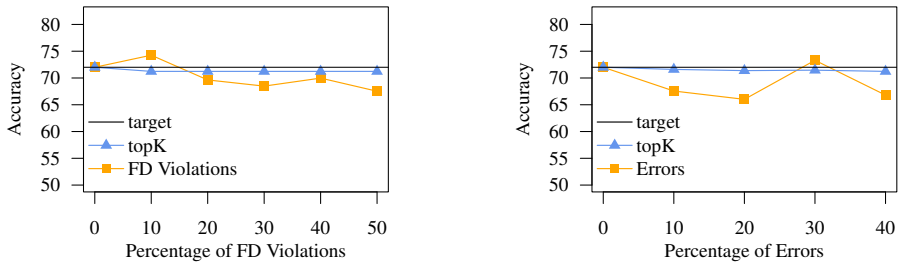
(a) Sampled = MAR (b) Sampled = MCAR

Fig. 5. Value Swap Errors.



(a) Sampled = MAR (b) Sampled = MCAR

Fig. 6. Scaling Errors.



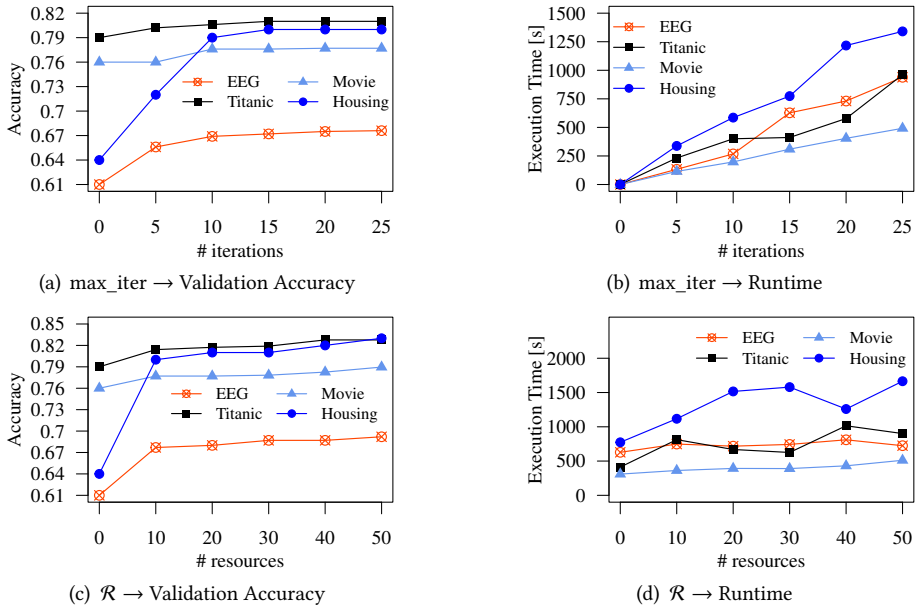
(a) FD Violations (b) Multiple Errors

Fig. 7. FD Violation and Mixed Errors.

Value Swaps: For value-swaps, we observe accuracy differences of at most 2% for MAR (Figure 5(a)) and 5% for MCAR (Figure 5(b)). Swaps with MCAR introduce more distortion in the dirty accuracy because of random swapping without correlations. SAGA provides almost constant accuracy, very close to the ground-truth.

Scaling Errors: For scaling errors, we randomly change the scaling of entire numeric columns. Figure 6 shows the resulting accuracy differences. The SAGA behavior is consistent as before, and the MAR errors (Figure 6(b)) introduce more model-affecting distortion in the dataset than MCAR (Figure 6(a)). However, there is a spurious spike for both dirty and clean data at 20%. Even in such scenarios SAGA consistently outperforms the dirty accuracy.

Violated FDs: For violations of functional dependencies (FDs), we systematically introduce inconsistencies in the Zip and Address columns. Figure 7(a) shows only slight drops in accuracy because we only introduce errors in a single column. SAGA exhibits almost constant accuracy.



(a) max_iter → Validation Accuracy
(b) max_iter → Runtime
(c) $\mathcal{R}$ → Validation Accuracy
(d) $\mathcal{R}$ → Runtime

Fig. 8. Impact of Logical/Physical Enumeration.

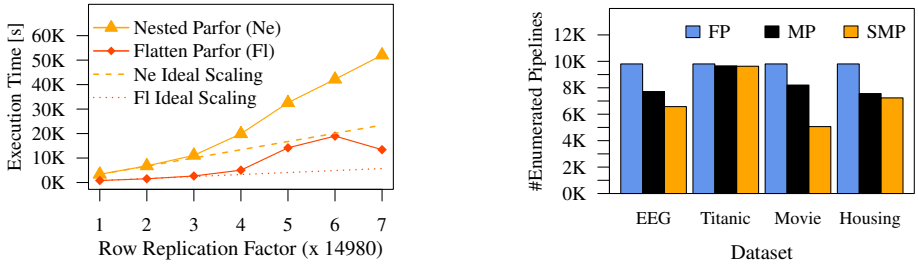
Mixed Errors: Finally, we inject a mix of errors (missing data, noise, value swaps, scaling, FD violations) into the dataset. Figure 7(b) shows that for mixed errors, there is a significant accuracy deterioration of up to 6% on dirty data. At some points the dirty accuracy catch-up with the ground truth due to significant distortion of column correlations. In contrast, SAGA's top-K cleaning also handles this more challenging case with accuracy close to the ground-truth accuracy. Overall, SAGA yields remarkably robust results outperforming dirty model accuracy on almost all error types and error rates, often very close or better than the ground-truth.

6.4 Scalability and Runtime Comparison

Finally, we evaluate various runtime properties, including parameters, pruning, runtimes of baselines, and parallelization strategies.

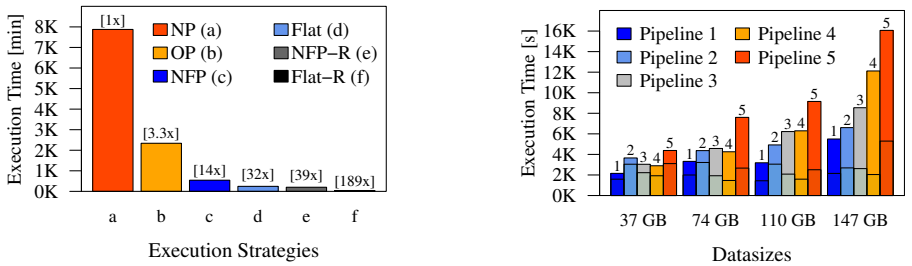
Impact of Parameters: The parameters max_iter (default 15) and resources $\mathcal{R}$ (default 50) heavily influence logical and physical pipeline enumeration. Figure 8(a) shows the top-1 accuracy for varying number of iterations (with default $\mathcal{R}$) for selected datasets. With increasing number of iterations (fixed seeds, only logical pipelines), we observe increasing accuracy that plateaus before our default 15. Figure 8(b) further shows linearly increasing runtimes when evaluating more populations. Together, converging accuracy and increasing runtime, motivates our convergence checks. Additionally, our starting heuristic yields good accuracy quickly. Figures 8(c) and 8(d) show the accuracy and runtime for varying resources $\mathcal{R}$ (with default max_iter). We observe increasing accuracy, which is higher due to full optimization. The runtime fluctuations are due to the impact of resources on parallelization and some hyper-parameters (e.g., in `dbscan` or `mice`).

Scalability with Datasize: For evaluating the scalability with increasing #rows, we replicate the EGG dataset up to 7x, which keeps other characteristics constant. Figure 9(a) shows the runtimes and ideal scaling (1x-runtime × rep factor). We compare Hyperband with basic nested-loops (over buckets/pipelines, and hyper-parameters), and our flattening approach (see Section 4.3). Since these loops are ParFor loops, flattening influences parallelization. Except for few outliers, we observe



(a) Nested & Flattened ParFor (b) Pruning Strategies

Fig. 9. Flattening and Pruning.



(a) Task-Parallel Plans (b) Data-Parallel Plans

Fig. 10. Task- and Data-parallel Execution Plans.

good scalability with increasing datasize, and—even for local, multi-threaded plans—already a >2x improvement by flattening. The moderate deterioration and outliers are due to larger intermediates and memory pressure (increased garbage collection overheads).

Pruning Strategies: We also evaluate the impact of pruning (skipping of pipelines by monotonicity). Figure 9(b) shows the number of evaluated pipelines for selected datasets without pruning (FP), pruning by monotonicity (MP), and sorted pruning by monotonicity (SMP), where we order the hyper-parameters accordingly. Pruning by monotonicity is lossless and thus, does not impact accuracy, while still reducing the number of evaluated pipelines up to 2x. However, this pruning is strongly data-dependent (e.g., Titanic vs. Movie) and applies only to a subset of primitives.

Task-parallel Plans: SAGA leverages local and distributed execution plans. Figure 10(a) compares the execution time of the following task-parallel plans, using the EEG dataset with replication factor 5 and approximately 10,000 physical pipelines.

- *No Parallelism (NP):* Single-threaded ops, and for-loops,
- *Operator Parallelism (OP):* Multi-threaded ops, and for-loops,
- *Nested Full Parallelism (NFP):* Nested-parallel, multi-threaded pipeline enum and ops,
- *Flatten Parallelism (Flat):* Flattened-parallel multi-threaded pipeline enum and ops
- *Nested Full Parallelism Remote (NFP-R):* Distributed NFP, and
- *Flatten Parallelism Remote (Flat-R):* Distributed Flat.

For local, single-node computation on the driver (16 physical/32 virtual cores), flattened, multi-threaded ParFor loops (Flat) yield improvements of 21x over NP, 6x over OP, and 2x over NFP. This characteristic is due to fewer synchronization barriers and better CPU utilization. Furthermore, for distributed computation on the 1+6 node cluster (32+192 virtual cores), flattened, distributed

Table 9. Baseline Runtime Comparison (SAGA in single node).

Dataset	BoostClean	HoloClean	Raha-Baran	Learn2Clean	SAGA
Animal	262 s	TO	33,484 s	NA	885 s
EEG	296 s	567 s	1,715 s	38 s	843 s
Movie	1,529 s	1,923 s	10,414 s	22	621 s
Nashville	41,658 s	OOM	TO	1118 s	5,961 s
Puma	39 s	404 s	8,535 s	19 s	210 s
Titanic	444 s	304 s	1,500 s	19 s	436 s
Cancer*	NA	996 s	918 s	13 s	294
Housing*	NA	TO	2,100 s	19 s	1,691 s

ParFor loops (Flat-R) yield improvements of 189x, 148x, and 13x over NP, OP and NFP. Due to fewer distributed jobs and a larger degree of parallelism, Flat-R further yields a 5x over NFP-R.

Data-parallel Plans: We further validate SAGA's scalability on large datasets, which are up to 205x larger than reported by prior work. Once an operation with its in/outputs exceeds local memory (here, 70 GB driver), we compile data-parallel plans. We scale EEG to sizes in [37 GB, 147 GB], and measure the runtime of cleaning pipelines and model training/testing (Figure 10(b)).

```
pipeline1: undersample, scale, winsorize, imputeByMean
pipeline2: imputeByFd, winsorize, outlierBySd, imputeByMean
pipeline3: forward_fill scale winsorize imputeByMean SMOTE
pipeline4: imputeByMean, winsorize, scale
pipeline5: mice, scale, winsorize
```

In these experiments, the min/max accuracies per pipeline did not vary much with increasing datasize due to replication. However, data-parallel plans ensure scalability if needed (for unknown datasets with many features); otherwise, SAGA also provides a sampling option, and techniques for estimating appropriate sampling fractions exist [72].

Baseline Comparisons: Finally, Table 9 shows the execution time for all baselines tools and datasets (see Table 5), where we run SAGA only with local, multi-threaded enumeration (Flat) and pruning. BoostClean and Learn2Clean show good runtime performance, but mixed accuracy. The most comparable approach to SAGA is Raha-Baran, which also applies multiple error detection and correction strategies without dropping train and test data. Even with multi-threaded enumeration, SAGA already shows runtime improvements of more than an order of magnitude on some datasets (e.g., Animal, Movie, Puma), and generally very robust performance characteristics. Nashville is a challenging dataset with 212,000 rows and 22 features, where HoloClean runs out of memory, Raha-Baran runs into timeouts, BoostClean takes 11.5 hours, while SAGA takes <1.7 hours.

7 ADDITIONAL RELATED WORK

Optimizing data cleaning pipelines for ML applications is related to the areas of data validation, ML-based data cleaning, data cleaning for ML, pipeline enumeration, and AutoML tools.

Data Validation: The validation of training and serving data focuses on both exploratory model training and production ML pipelines. Common strategies include checking descriptive statistics, histograms of continuous and categorical values, as well as the number of distinct and existing values for expected shapes [78]. Schelter et al. introduced a UDF library of constraints and metrics for quality checks, computed via distributed Spark operations and incremental maintenance [89]. Similarly, Google's TFX platform [4] offers via TensorFlow Data Validation [15, 20, 34] means for schema validation, statistics, validation checks, and anomaly detection. Recent work includes

model assertions [51], validation via unit tests in Deequ [88] and ease.ml/ci [52, 84], as well as tests for model robustness against data corruptions in Jenga [91] and expected prediction quality [90]. BigDancing [54] performs rule-based data validation and repairs on distributed data, as well as compiles efficient runtime plans with scan sharing and other data access optimizations. In contrast, SAGA automatically optimizes complex data cleaning pipelines—while optionally leveraging user-provided constraints—for minimizing the loss of a target ML application.

ML-based Data Cleaning: Machine learning is widely used to improve the effectiveness and efficiency of data cleaning [65, 104, 105] by detecting errors and predicting repairs. The HoloClean [83] and HoloDetect [44] systems combine qualitative and quantitative repair signals in a statistical model. However, these systems require manual hyper-parameter optimization and follow a one-shot aggregation strategy. ActiveClean [58] cleans the training data for an ML application but rely on the user to specify how to clean and featurize the dataset. Finally, Raha-Baran [63, 64] use a combination of active learning and clustering to find labels for sampled data clusters and use them to classify data as dirty or clean. SAGA yields robust accuracy improvements over these baselines, and is implemented in linear algebra to seamlessly integrate ML-based cleaning primitives, ML applications, and hyper-parameter tuning.

Data Cleaning for ML: Recent work [11, 68, 107] made a case for "fit-of-purpose" data cleaning, that is, data cleaning is not an isolated and model-agnostic task. Data that is of high quality for one purpose may be of low quality for another purpose. Signals from the downstream ML applications can be used for effective data cleaning for specific use-cases. Systems such as ActiveClean [58], BoostClean [57], AlphaClean [59], CPClean [53], Learn2Clean [8] already use these signals, but focus on specific models or types of errors (e.g., missing values). There are also models like XGBoost that are more robust to missing values because they learn default paths (dedicated sub-trees) for all tuples that miss split-feature values [23], but cannot handle disguised missing values yet (where missing values are replaced, for instance, by defaults) [80]. Li et al. conducted a large-scale study on the impact of a variety of basic data cleaning primitives on the accuracy of ML models [61]. SAGA is a natural extension of this line of work in several dimensions by optimizing data cleaning pipelines for an extensible set of scalable cleaning primitives (with high-level traits like monotonicity), different ML models, as well as local and distributed runtime plans.

Pipelines Enumeration: Apart from data cleaning, SAGA is also related to the enumeration of data preparation pipelines in general. Early work focused on optimizing ETL (extract-transform-load) workflows with similar, but ETL-specific transitions [94], as well as the evaluation of such workflows under different metrics [95, 96]. Recent work on pipeline enumeration for data preparation include data augmentation pipelines via reinforcement learning [27], explainable stock market prediction models via evolutionary algorithms [28], data preparation and model selection [93], as well as feature selection for algorithmic fairness [87]. SAGA shares similar ideas of evolutionary pipeline enumeration and physical/logical pipeline tuning, but focuses on cleaning primitives, dedicated pruning techniques, as well as local and distributed evaluation.

AutoML Systems: Optimizing data cleaning pipelines—with respect to model performance on a validation set—is also closely connected to AutoML. Although existing systems like Auto-WEKA [56, 101], Auto-sklearn [37], BOHB [35], TuPAQ [97], TPOT [71], Alpine Meadow [93] and DeepLine [43] also include basic data preparation steps, they focus primarily on model and feature selection as well as hyper-parameter tuning for non-ML-experts [5, 10, 62]. Common techniques are multi-armed bandits for model selection [17, 60, 97], Bayesian Optimization for hyper-parameter tuning [16, 37, 56, 101], evolutionary algorithms [71], and hierarchical enumeration [71, 93]. Recently, most cloud providers also offer dedicated AutoML services [38, 109]. Furthermore, neural architecture search [49, 75, 110] also leverages evolutionary algorithms for constructing new network architectures from building blocks under multiple objectives (e.g., accuracy and runtime).

In Saga, we leverage similar ideas for evolutionary algorithms and hyper-parameter tuning with a specific focus on data cleaning pipelines.

8 CONCLUSIONS

We introduced Saga, a framework for finding effective data cleaning pipelines for downstream ML applications. We utilize an evolutionary algorithm that incrementally refines logical pipelines, and Hyperband for tuning physical pipelines with their parameters. Compared to state-of-the-art, we see robust accuracy improvements with good generalization. A combination of multi-level enumeration, pruning, and parallelization also yields efficient and scalable runtime behavior. We draw two major conclusions. First, a simple and clean adoption of ideas from evolutionary algorithms and hyper-parameter tuning is able to automate the mechanical aspects of finding good data cleaning pipelines, while providing extensibility for different ML applications, cleaning primitives, and manual refinements. Second, implementing this cleaning framework in linear algebra on top of ML systems yields a seamless integration with ML-based cleaning primitives, and leverages the infrastructure for optimizing and executing linear algebra programs. Further interesting future work includes dedicated data-parallel execution strategies and HW acceleration; the integration with Auto-ML tools; and data cleaning for nested and multi-modal data.

ACKNOWLEDGEMENTS

We thank Ziawasch Abedjan for valuable comments on an earlier version, as well as our anonymous reviewers for their constructive comments and suggestions, which helped improve the paper.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. Data Profiling. In *Synthesis Lectures on Data Management*. http://sites.computer.org/debull/A18june/p3.pdf

[2] Giorgos Alexiou, George Papastefanatos, Vassilis Stamatopoulos, Georgia Koutrika, and Nectarios Koziris. 2022. QueryER: A Framework for Fast Analysis-Aware Deduplication over Dirty Data. *CoRR* abs/2202.01546 (2022). arXiv:2202.01546 https://arxiv.org/abs/2202.01546

[3] ASQ/ANSI/ISO. 2015. 9001:2015: Quality management systems - Requirements.

[4] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *KDD*. 1387–1395. https://doi.org/10.1145/3097983.3098021

[5] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *JMLR* 13 (2012), 281–305. https://doi.org/10.5555/2503308.2188395

[6] Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. 2011. Generic Schema Matching, Ten Years Later. *PVLDB* 4, 11 (2011), 695–701. http://www.vldb.org/pvldb/vol4/p695-bernstein_madhavan_rahm.pdf

[7] Philip A. Bernstein and Sergey Melnik. 2007. Model management 2.0: manipulating richer mappings. In *SIGMOD*. https://doi.org/10.1145/1247480.1247482

[8] Laure Berti-Équille. 2019. Learn2Clean: Optimizing the Sequence of Tasks for Web Data Preparation. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 2580–2586. https://doi.org/10.1145/3308558.3313602

[9] Felix Bießmann, Tammo Rukat, Philipp Schmidt, Prathik Naidu, Sebastian Schelter, Andrey Taptunov, Dustin Lange, and David Salinas. 2019. DataWig: Missing Value Imputation for Tables. *JMLR* 20 (2019). http://jmlr.org/papers/v20/18-753.html

[10] Carsten Binnig, Benedetto Buratti, Yeounoh Chung, Cyrus Cousins, Tim Kraska, Zeyuan Shang, Eli Upfal, Robert C. Zeleznik, and Emanuel Zgraggen. 2018. Towards Interactive Curation & Automatic Tuning of ML Pipelines. In *DEEM*. 1:1–1:4. https://doi.org/10.1145/3209889.3209891

[11] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*. http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf

[12] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas Burdick, and Shivakumar Vaithyanathan. 2014. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB* 7, 7 (2014), 553–564. https://doi.org/10.14778/2732286.2732292
[13] Christoph Böhm, Gerard de Melo, Felix Naumann, and Gerhard Weikum. 2012. LINDA: distributed web-of-data-scale entity matching. In *CIKM*. 2104–2108. https://doi.org/10.1145/2396761.2398582
[14] Matthias Böhm, Uwe Wloka, Dirk Habich, and Wolfgang Lehner. 2009. GCIP: exploiting the generation and optimization of integration processes. In *EDBT*, Vol. 360. 1128–1131. https://doi.org/10.1145/1516360.1516494
[15] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *MLSys*. https://proceedings.mlsys.org/book/267.pdf
[16] Eric Brochu, Vlad M. Cora, and Nando de Freitas. 2010. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. *CoRR* (2010). http://arxiv.org/abs/1012.2599
[17] Sébastien Bubeck and Nicolò Cesa-Bianchi. 2012. Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. *Found. Trends Mach. Learn.* 5, 1 (2012), 1–122. https://doi.org/10.1561/2200000024
[18] Douglas Burdick, Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang-Chiew Tan. 2019. Expressive power of entity-linking frameworks. *J. Comput. Syst. Sci.* 100 (2019), 44–69. https://doi.org/10.1016/j.jcss.2018.09.001
[19] José Cambronero, John K. Feser, Micah J. Smith, and Samuel Madden. 2017. Query optimization for dynamic imputation. *PVLDB* 10 (2017), 1310–1321. https://doi.org/10.14778/3137628.3137641
[20] Emily Caveness, Paul Suganthan G. C., Zhuo Peng, Neoklis Polyzotis, Sudip Roy, and Martin Zinkevich. 2020. TensorFlow Data Validation: Data Analysis and Validation in Continuous ML Pipelines. In *SIGMOD*. 2793–2796. https://doi.org/10.1145/3318464.3384707
[21] Austin Animal Center. 2022. Shelter Animal Outcomes competition dataset from Kaggle. https://www.kaggle.com/competitions/shelter-animal-outcomes/data
[22] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Intell. Res.* 16 (2002), 321–357. https://doi.org/10.1613/jair.953
[23] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *SIGKDD*. 785–794. https://doi.org/10.1145/2939672.2939785
[24] Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. 2016. Distributed Data Deduplication. *PVLDB* 9, 11 (2016), 864–875. https://doi.org/10.14778/2983200.2983203
[25] Xu Chu, Mourad Ouzzani, John Morcos, Ihab F. Ilyas, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: Reliable Data Cleaning with Knowledge Bases and Crowdsourcing. *PVLDB* 8, 12 (2015). https://doi.org/10.14778/2824032.2824109
[26] Yeounoh Chung, Tim Kraska, Neoklis Polyzotis, Ki Hyun Tae, and Steven Euijong Whang. 2020. Automated Data Slicing for Model Validation: A Big Data - AI Integration Approach. *IEEE Trans. Knowl. Data Eng.* 32, 12 (2020), 2284–2296. https://doi.org/10.1109/TKDE.2019.2916074
[27] Ekin D. Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. 2019. AutoAugment: Learning Augmentation Strategies From Data. In *CVPR*. 113–123. https://doi.org/10.1109/CVPR.2019.00020
[28] Can Cui, Wei Wang, Meihui Zhang, Gang Chen, Zhaojing Luo, and Beng Chin Ooi. 2021. AlphaEvolve: A Learning Framework to Discover Novel Alphas in Quantitative Investment. In *SIGMOD*. 2208–2216. https://doi.org/10.1145/3448016.3457324
[29] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. 2013. NADEEF: a commodity data cleaning system. In *SIGMOD*. 541–552. https://doi.org/10.1145/2463676.2465327
[30] Data.Nashville.gov. 2020. Nashville Traffic Accidents Dataset. https://data.nashville.gov/Police/Traffic-Accidents/6v6w-hpcw
[31] Delve Datasets. 2022. Puma Dataset. https://www.cs.toronto.edu/~delve/data/datasets.html
[32] data.world. 2016. OLS Regression Challenge - Cancer. https://data.world/nrippner/ols-regression-challenge
[33] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System. (2017). http://cidrdb.org/cidr2017/papers/p44-deng-cidr17.pdf
[34] Mike Dreves, Gene Huang, Zhuo Peng, Neoklis Polyzotis, Evan Rosen, and Paul Suganthan G. C. 2020. From Data to Models and Back. In *DEEM@SIGMOD Workshop*. https://doi.org/10.1145/3399579.3399868
[35] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *ICML*. 1436–1445. http://proceedings.mlr.press/v80/falkner18a.html
[36] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. 2020. Auto-Sklearn 2.0: The Next Generation. *CoRR* (2020). https://arxiv.org/abs/2007.04074
[37] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. Auto-sklearn: Efficient and Robust Automated Machine Learning. In *Automated Machine Learning - Methods, Systems, Challenges*. 113–134. https://doi.org/10.1007/978-3-030-05318-5

[38] Nicoló Fusi, Rishit Sheth, and Melih Elibol. 2018. Probabilistic Matrix Factorization for Automated Machine Learning. In *NeurIPS*. 3352–3361. https://proceedings.neurips.cc/paper/2018/file/b59a51a3c0bf9c5228fde841714f523a-Paper.pdf
[39] Sainyam Galhotra, Donatella Firmani, Barna Saha, and Divesh Srivastava. 2021. BEER: Blocking for Effective Entity Resolution. In *SIGMOD*. 2711–2715. https://doi.org/10.1145/3448016.3452747
[40] Gábor E. Gévay, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. The Power of Nested Parallelism in Big Data Processing - Hitting Three Flies with One Slap. In *SIGMOD*. 605–618. https://doi.org/10.1145/3448016.3457287
[41] Joachim Hammer, Michael Stonebraker, and Oguzhan Topsakal. 2005. THALIA: Test Harness for the Assessment of Legacy Information Integration Approaches. In *ICDE*. 485–486. https://doi.org/10.1109/ICDE.2005.140
[42] Chicago health services. 2022. Chicago Food Inspection Dataset. https://data.cityofchicago.org/Health-Human-Services/Food-Inspections/4ijn-s7e5/data
[43] 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 *KDD*. 2103–2113. https://doi.org/10.1145/3394486.3403261
[44] Alireza Heidari, Joshua McGrath, Ihab F. Ilyas, and Theodoros Rekatsinas. 2019. HoloDetect: Few-shot learning for error detection. *SIGMOD* (2019), 829–846. https://doi.org/10.1145/3299869.3319888
[45] Christoph Hube, Besnik Fetahu, and Ujwal Gadiraju. 2019. Understanding and Mitigating Worker Biases in the Crowdsourced Collection of Subjective Judgments. In *CHI*. ACM, 407. https://doi.org/10.1145/3290605.3300637
[46] Kevin G. Jamieson and Ameet Talwalkar. 2016. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *AISTATS (JMLR Workshop and Conference Proceedings, Vol. 51)*. 240–248. http://proceedings.mlr.press/v51/jamieson16.html
[47] Kaggle. 2022. House Prices - Advanced Regression Techniques. https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques/data
[48] Kaggle. 2022. Titanic Dataset. https://www.kaggle.com/competitions/titanic/data
[49] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, and Eric P. Xing. 2018. Neural Architecture Search with Bayesian Optimisation and Optimal Transport. In *NeurIPS*. 2020–2029. https://proceedings.neurips.cc/paper/2018/hash/f33ba15effa5c10e873bf3842afb46a6-Abstract.html
[50] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *CHI*. 3363–3372. https://doi.org/10.1145/1978942.1979444
[51] Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. 2020. Model Assertions for Monitoring and Improving ML Models. In *MLSys*. https://proceedings.mlsys.org/book/319.pdf
[52] Bojan Karlas, Matteo Interlandi, Cédric Renggli, Wentao Wu, Ce Zhang, Deepak Mukunthu Iyappan Babu, Jordan Edwards, Chris Lauren, Andy Xu, and Markus Weimer. 2020. Building Continuous Integration Services for Machine Learning. In *KDD*. 2407–2415. https://doi.org/10.1145/3394486.3403290
[53] Bojan Karlas, 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. *PVLDB* 14, 3 (2020), 255–267. https://doi.org/10.5555/3430915.3442426
[54] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. BigDansing: A System for Big Data Cleansing. In *SIGMOD*. 1215–1230. https://doi.org/10.1145/2723372.2747646
[55] Pradap Konda, Sanjib Das, Paul Suganthan G. C., AnHai Doan, Adel Ardalan, Jeffrey R. Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeffrey F. Naughton, Shishir Prasad, Ganesh Krishnan, Rohit Deep, and Vijay Raghavendra. 2016. Magellan: Toward Building Entity Matching Management Systems over Data Science Stacks. *PVLDB* 9, 13 (2016), 1581–1584. https://doi.org/10.14778/3007263.3007314
[56] 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. *J. Mach. Learn. Res.* 18 (2017), 25:1–25:5. http://jmlr.org/papers/v18/16-261.html
[57] Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, and Eugene Wu. 2017. BoostClean: Automated Error Detection and Repair for Machine Learning. *CoRR* abs/1711.01299 (2017). http://arxiv.org/abs/1711.01299
[58] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. 2016. Activeclean: Interactive data cleaning for statistical modeling. *PVLDB* 9, 12 (2016), 948–959. https://doi.org/10.14778/2994509.2994514
[59] Sanjay Krishnan and Eugene Wu. 2019. AlphaClean: Automatic Generation of Data Cleaning Pipelines. (2019). http://arxiv.org/abs/1904.11827
[60] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.* 18 (2017), 185:1–185:52. http://jmlr.org/papers/v18/16-558.html
[61] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2021. CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks. In *ICDE*. 13–24. https://doi.org/10.1109/ICDE51399.2021.00009

[62] Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. 2018. Ease.Ml: Towards Multi-Tenant Resource Sharing for Machine Learning Workloads. *PVLDB* 11, 5 (2018), 607–620. https://doi.org/10.1145/3187009.3177737
[63] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning. *PVLDB* 13, 11 (2020), 1948–1961. https://doi.org/10.14778/3407790.3407801
[64] Mohammad Mahdavi, Samuel Madden, Ziawasch Abedjan, Mourad Ouzzani, Nan Tang, Raul Castro Fernandez, and Michael Stonebraker. 2019. Raha: A configuration-free error detection system. *SIGMOD* (2019), 865–882. https://doi.org/10.1145/3299869.3324956
[65] Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. 2010. ERACER: a database approach for statistical inference and data cleaning. In *SIGMOD*. 75–86. https://doi.org/10.1145/1807167.1807178
[66] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2019. Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems. In *DEEM@SIGMOD Workshop*. https://doi.org/10.1145/3329486.3329496
[67] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *PVLDB* 13, 11 (2020), 2159–2173. https://doi.org/10.14778/3447689.3447691
[68] Felix Neutatz, Binger Chen, Ziawasch Abedjan, and Eugene Wu. 2021. From Cleaning before ML to Cleaning for ML. *IEEE Data Eng. Bull.* 44, 1 (2021), 24–41. http://sites.computer.org/debull/A21mar/p24.pdf
[69] Felix Neutatz, Binger Chen, Yazan Alkhatib, Jingwen Ye, and Ziawasch Abedjan. 2022. Data Cleaning and AutoML: Would an Optimizer Choose to Clean? *Datenbank-Spektrum* 22, 2 (2022), 121–130. https://doi.org/10.1007/s13222-022-00413-2
[70] Uchechukwu Njoku, Besim Bilalli, Alberto Abelló, and Gianluca Bontempi. 2023. Wrapper Methods for Multi-Objective Feature Selection. In *EDBT*. 697–709. https://doi.org/10.48786/edbt.2023.58
[71] Randal S. Olson and Jason H. Moore. 2019. TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning. In *Automated Machine Learning - Methods, Systems, Challenges*. 151–160. https://doi.org/10.1007/978-3-030-05318-5_8
[72] Yongjoo Park, Jingyi Qing, Xiaoyang Shen, and Barzan Mozafari. 2019. BlinkML: Efficient Maximum Likelihood Estimation with Probabilistic Guarantees. In *SIGMOD*. 1135–1152. https://doi.org/10.1145/3299869.3300077
[73] Eliana Pastor, Elena Baralis, and Luca de Alfaro. 2023. A Hierarchical Approach to Anomalous Subgroup Discovery. In *ICDE*. 2647–2659. https://doi.org/10.1109/ICDE55515.2023.00203
[74] Dessislava Petrova-Antonova and Rumyana Tancheva. 2020. Data Cleaning: A Case Study with OpenRefine and Trifacta Wrangler. In *QUATIC*, Vol. 1266. 32–40. https://doi.org/10.1007/978-3-030-58793-2_3
[75] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. 2018. Efficient Neural Architecture Search via Parameter Sharing. In *ICML*. 4092–4101. http://proceedings.mlr.press/v80/pham18a.html
[76] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *SIGMOD*. ACM, 1426–1439. https://doi.org/10.1145/3448016.3452788
[77] Clément Pit-Claudel, Zelda E. Mariet, Rachael Harding, and Samuel Madden. 2016. Outlier Detection in Heterogeneous Datasets using Automatic Tuple Expansion. In *Technical Report MIT-CSAIL-TR-2016-002*. http://hdl.handle.net/1721.1/101150
[78] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data Management Challenges in Production Machine Learning. In *SIGMOD*. 1723–1726. https://doi.org/10.1145/3035918.3054782
[79] Abdulhakim A. Qahtan, Ahmed Elmagarmid, Raul Castro Fernandez, Mourad Ouzzani, and Nan Tang. 2018. FAHES: A Robust Disguised Missing Values Detector. (2018), 2100–2109. https://doi.org/10.1145/3219819.3220109
[80] Abdulhakim Ali Qahtan, Ahmed K. Elmagarmid, Raul Castro Fernandez, Mourad Ouzzani, and Nan Tang. 2018. FAHES: A Robust Disguised Missing Values Detector. In *SIGKDD*. 2100–2109. https://doi.org/10.1145/3219819.3220109
[81] Kun Qian, Lucian Popa, and Prithviraj Sen. 2019. SystemER: A Human-in-the-loop System for Explainable Entity Resolution. *PVLDB* 12, 12 (2019), 1794–1797. https://doi.org/10.14778/3352063.3352068
[82] Vijayshankar Raman and Joseph M. Hellerstein. 2001. Potter's Wheel: An Interactive Data Cleaning System. In *VLDB*. 381–390. http://www.vldb.org/conf/2001/P381.pdf
[83] Theodoros Rekatsinas, Xu Chuy, Ihab F. Ilyasy, and Christopher Ré. 2017. HoloClean: Holistic data repairs with probabilistic inference. *PVLDB* 10 (2017), 1190–1201. https://doi.org/10.14778/3137628.3137631
[84] Cédric Renggli, Bojan Karlas, Bolin Ding, Feng Liu, Kevin Schawinski, Wentao Wu, and Ce Zhang. 2019. Continuous Integration of Machine Learning Models with ease.ml/ci: Towards a Rigorous Yet Practical Treatment. In *MLSys*. https://proceedings.mlsys.org/book/266.pdf
[85] UCI Repository. 2013. EEG Eye State Dataset. https://archive.ics.uci.edu/ml/datasets/EEG+Eye+State
[86] Svetlana Sagadeeva and Matthias Boehm. 2021. SliceLine: Fast, Linear-Algebra-based Slice Finding for ML Model Debugging. In *SIGMOD*. 2290–2299. https://doi.org/10.1145/3448016.3457323
[87] Ricardo Salazar, Felix Neutatz, and Ziawasch Abedjan. 2021. Automated Feature Engineering for Algorithmic Fairness. *PVLDB* 14, 9 (2021), 1694–1702. https://doi.org/10.14778/3461535.3463474

[88] Sebastian Schelter, Felix Bießmann, Dustin Lange, Tammo Rukat, Philipp Schmidt, Stephan Seufert, Pierre Brunelle, and Andrey Taptunov. 2019. Unit Testing Data with Deequ. In *SIGMOD*. 1993–1996. https://doi.org/10.1145/3299869.3320210

[89] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Bießmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *PVLDB* 11, 12 (2018), 1781–1794. https://doi.org/10.14778/3229863.3229867

[90] Sebastian Schelter, Tammo Rukat, and Felix Bießmann. 2020. Learning to Validate the Predictions of Black Box Classifiers on Unseen Data. In *SIGMOD*. 1289–1299. https://doi.org/10.1145/3318464.3380604

[91] Sebastian Schelter, Tammo Rukat, and Felix Biessmann. 2021. JENGA - A Framework to Study the Impact of Data Errors on the Predictions of Machine Learning Models. In *EDBT*. 529–534. https://doi.org/10.5441/002/edbt.2021.63

[92] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.* 42, 3 (2017), 19:1–19:21. https://doi.org/10.1145/3068335

[93] 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 *SIGMOD*. 1171–1188. https://doi.org/10.1145/3299869.3319863

[94] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. 2005. Optimizing ETL Processes in Data Warehouses. In *ICDE*. 564–575. https://doi.org/10.1145/2463676.2465247

[95] Alkis Simitsis, Kevin Wilkinson, Malú Castellanos, and Umeshwar Dayal. 2009. QoX-driven ETL design: reducing the cost of ETL consulting engagements. In *SIGMOD*. 953–960. https://doi.org/10.1145/1559845.1559954

[96] Alkis Simitsis, Kevin Wilkinson, and Petar Jovanovic. 2013. xPAD: a platform for analytic data flows. In *SIGMOD*. 1109–1112.

[97] 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 *SoCC*. 368–380. https://doi.org/10.1145/2806777.2806945

[98] Michael Stonebraker, Daniel Bruckner, Ihab F. Ilyas, George Beskales, Mitch Cherniack, Stanley B. Zdonik, Alexander Pagan, and Shan Xu. 2013. Data Curation at Scale: The Data Tamer System. In *CIDR*. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper28.pdf

[99] Michael Stonebraker and Ihab F. Ilyas. 2018. Data Integration: The Current Status and the Way Forward. *IEEE Data Eng. Bull.* 41 (2018), 3–9.

[100] Ki Hyun Tae and Steven Euijong Whang. 2021. Slice Tuner: A Selective Data Acquisition Framework for Accurate and Fair Machine Learning Models. In *SIGMOD*. 1771–1783. https://doi.org/10.1145/3448016.3452792

[101] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *KDD*. 847–855. https://doi.org/10.1145/2487575.2487629

[102] Stef van Buuren and Karin Groothuis-Oudshoorn. 2011. mice: Multivariate Imputation by Chained Equations in R. *Journal of Statistical Software* 45, 3 (2011), 1–67. https://www.jstatsoft.org/index.php/jss/article/view/v045i03

[103] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. 2018. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *PVLDB* 12, 4 (2018), 446–460. https://doi.org/10.14778/3297753.3297763

[104] Mohamed Yakout, Laure Berti-Équille, and Ahmed K. Elmagarmid. 2013. Don't be SCAREd: use SCalable Automatic REpairing with maximal likelihood and bounded changes. In *SIGMOD*. https://doi.org/10.1145/2463676.2463706

[105] Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. 2011. Guided data repair. *PVLDB* 4, 5 (2011), 279–289. https://doi.org/10.14778/1952376.1952378

[106] Matei A. Zaharia. 2013. *An Architecture for and Fast and General Data Processing on Large Clusters*. Ph. D. Dissertation. University of California, Berkeley, USA.

[107] Amrapali Zaveri and Anisa Rula. 2019. Data Quality and Data Cleansing of Semantic Data. (2019). https://doi.org/10.1007/978-3-319-63962-8_289-1

[108] Aoqian Zhang, Shaoxu Song, Jianmin Wang, and Philip S. Yu. 2017. Time series data cleaning: From anomaly detection to anomaly repairing. *PVLDB* 10, 10 (2017), 1046–1057. https://doi.org/10.14778/3115404.3115410

[109] Hantian Zhang, Luyuan Zeng, Wentao Wu, and Ce Zhang. 2017. How Good Are Machine Learning Clouds for Binary Classification with Good Features? *CoRR* abs/1707.09562 (2017). http://arxiv.org/abs/1707.09562

[110] Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. In *ICLR*. https://openreview.net/forum?id=r1Ue8Hcxg

Received January 2023; revised April 2023; accepted May 2023