Generalizable Data Cleaning of Tabular Data in Latent Space

Eduardo Reis
eduardo.reis@cs.tu-darmstadt.de
Technical University of Darmstadt

Mohamed Abdelaal
mohamed.abdelaal@softwareag.com
Software AG

Carsten Binnig
carsten.binnig@cs.tu-darmstadt.de
Technical University of Darmstadt & DFKI

ABSTRACT

In this paper, we present a new method for learned data cleaning. In contrast to existing methods, our method learns to clean data in the latent space. The main idea is that we (1) shape the latent space such that we know the area where clean data resides and (2) learn latent operators trained on error repair (Lopster) which shift erroneous data (e.g., table rows with noise, outliers, or missing values) in their latent representation back to a "clean" region, thus abstracting the complexities of the input domain. When formulating data cleaning as a simple shift operation in latent space, we can repair all types of errors using the same method which makes it more robust than other methods. Importantly, with our method, we can handle errors that are unseen during the training of our error repair model. We do not rely on an external error detection method as seen in the state-of-the-art, instead, we handle both detection and repair within the Lopster framework. In our evaluation, we show that our approach outperforms existing cleaning methods even when trained on only a subset of the errors that occur in the dirty data.

PVLDB Reference Format:
Eduardo Reis, Mohamed Abdelaal, and Carsten Binnig. Generalizable Data Cleaning of Tabular Data in Latent Space. PVLDB, 17(13): 4786 - 4798, 2024.
doi:10.14778/3704965.3704983

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

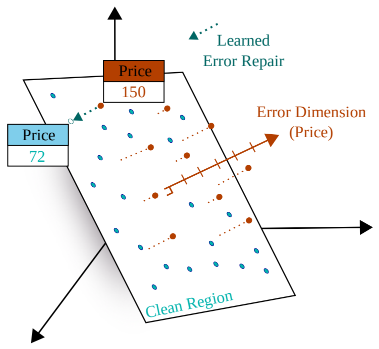


Figure 1: The main idea of Lopster is to learn data cleaning in the latent space. Key to our method is that we shape the latent space such that all clean data resides in an identifiable region. For cleaning the data, we first project a tuple into the latent space, then with the help of learned latent operators, we move dirty tuples back to the clean region and decode this cleaned representation into a clean tuple.

1 INTRODUCTION

Data quality is important. Nowadays companies rely on data acquired through third parties or open sourced, typically of lower quality than the small curated datasets collected locally. Even within the company, data errors and missing values are common due to human oversight or missing constraints for guaranteeing data quality. Moreover, as Machine Learning (ML) models become more and more ubiquitous, one of the main factors for enabling highly accurate models is train data quality. Having dirty data in the training set severely biases the model towards wrong inferences [18].

High overheads for data cleaning. Much of the manual efforts of ML engineers are spent today on data engineering, e.g., preparing data for data analytics. Within the data engineering pipeline, the initial steps typically concern detecting and repairing errors in the data, also known as data cleaning. Unfortunately, tabular data can be highly specialized, has sparse and heterogeneous errors and error types overlay each other [8, 28]. Hence, data cleaning is a cumbersome process done by experts, and although solutions to partially automate it are available, choosing the best method for each use case is hard and time consuming.

Learned approaches to the rescue. Automating data cleaning is thus a goal shared by both the ML and database communities. A recent approach in this direction is to solve data cleaning using ML. To be more precise, ML models are trained to learn both error detection and error repair for the dataset at hand. While these ML-based cleaning methods show promising results, they can only clean error types seen during training. As such, many errors stay undetected and thus might have a negative impact on the downstream ML model training.

How do learned approaches work? Existing ML-based approaches for data cleaning are mostly based on the supervised learning paradigm, where a training dataset requires labeled clean and dirty data. The standard pipeline for learning a ML-based cleaning method is as follows: in the training data, each cell of a table tuple is labeled as dirty or clean which allows us to train an error detection model; i.e., a model that decides whether a table cell is erroneous or not. Next, for each cell that contains an error, the error type (e.g. missing values, noise) must be identified, and a correspondent error repair function applied.

Low generalizability of existing approaches. Given the heterogeneity of error types in tabular data, it is hard to have a training set that covers all possible errors that can occur on all the tuples in the domain of even a single dataset. To aggravate the issue, most

Proceedings of the VLDB Endowment, Vol. 17, No. 13 ISSN 2150-8097.
doi:10.14778/3704965.3704983

error detection methods are typically tied to a set of predefined error types [22], thus, any error repair method that depends on a detector is also limited to this subset. Ideally, data cleaning models have to tackle errors not seen during training, which is a case of "out-of-distribution generalization" [9]. In such cases, the ML models have to rely on inductive biases learned during training that are specific to the set of errors presented in the dirty data, which may not generalize to new unseen error types.

Data cleaning in latent space. We strive to break this paradigm by solving any error type with the same repair method. The limitation of having different functions, or models, for different error types arises from attempting data cleaning in the input space. In this paper, we propose Latent OPeratorS Trained on Error Repair (Lopster) as a data cleaning solution with generalization for error types unseen during training, and no need for an external error detection method. The main idea is to move the data cleaning task to the latent space.

Shaped latent spaces. To avoid having different functions to solve different error types, we project the data to a "shaped" latent space with structurally separated clean and dirty regions. Next, we train our Lopster model to map dirty data to a specific direction of the "shaped" latent space, "orthogonal" to the region used for clean data. In our model, if the input tuple is clean it will be projected into the clean region, and if it is dirty it will be projected somewhere along the error dimension learned by our model, as shown in Figure 1. Moreover, error repair is simplified to the process of moving this projection along the error dimension back to the point where it intersects with the clean region. A key advantage is that error types unseen during training can be repaired by also projecting them back alongside this error dimension.

Highly promising results. To evaluate Lopster, we train our model in a simple set of errors, then compare our results against other approaches on dirty data generated by a new and unseen set of error functions, provided by the published data cleaning benchmark REIN [1]. Through our extensive evaluation in Section 5, we show that our approach is not only more robust but can indeed generalize to unseen error types.

Outline of the paper. In Section 2, we first present an overview of Lopster. Next, we discuss details of the training and inference procedures in Section 3 and Section 4. In our evaluation in Section 5, we show that our method is more general than the existing approaches while having competitive performance on numerical columns across all datasets, and superior performance on categorical columns. Also, we evaluate the scalability of our method and the sensitivity to each hyperparameter. Finally, we iterate over related works in Section 6 and provide concluding remarks in Section 7.

2 LATENT OPERATORS OVERVIEW

In this section, we first explain the model architecture used for Lopster, and then its training and inference procedures.

2.1 Model Architecture

Figure 2 illustrates an overview of how Lopster can be used to repair errors in tabular data. The main idea is to first encode rows of a table to a latent representation, then apply the error repair in latent space. Since tabular data can have a varying number of columns,

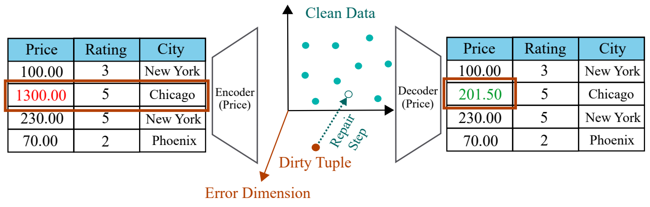


Figure 2: Our encode-repair-decode architecture. The encoder maps a dirty tuple into latent space, then a shift function moves this erroneous latent representation back to the clean region. Next, the decoder reconstructs the cleaned tuple.

we chose to train one Lopster model per column. In turn, to capture all important data characteristics of the entire tuple in the latent space, each instance of the Lopster model for a particular column still receives the whole tuple as input. However, the model repairs errors only in this particular column. In other words, a Lopster model for the *Price* column in Figure 2 receives as input the full tuple (1300.00, 5, *Chicago*) where the apartment price is an outlier, and yields the repaired value only for the attribute *Price*.

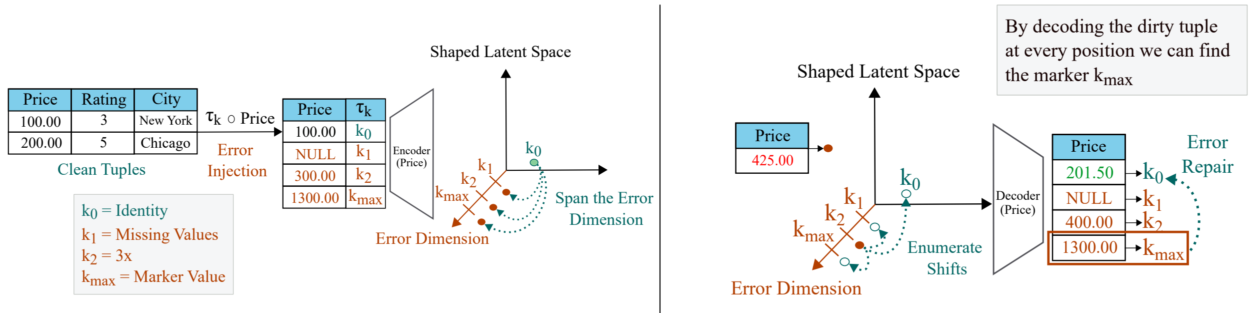
The importance of using the whole tuple (i.e., all attributes) for one instance of the Lopster model is to capture correlations in the input data. For example, apartment prices in New York are in general much higher than in Phoenix, thus, simply projecting apartment prices to the latent space would not capture this information. The same process must then be repeated for the other columns (i.e., *City* and *Rating*) as well, using their respective Lopster instances. To enable repairs in latent space, we propose a novel *encoder-repair-decoder* architecture. While the encoder-decoder architecture is prominent in ML, we extend it by a *repair* step after the encoder. The repair is a simple shift along the error dimension in latent space back to the region where the clean data is mapped to [1].

2.2 Training and Inference Procedures

Training. The main idea to enable such an encoder-repair-decoder architecture is that we need to "shape" the latent space, such that the regions in latent space with clean and dirty data are orthogonal, and the dirty region can represent a wide spectrum of errors. Therefore, we follow our training procedure as shown in Figure 3a, where we start with a clean sample of tabular data. Noteworthy, this clean sample is provided by the user for training, but we do not assume every tuple to be perfectly clean. Also, as we show in our evaluation, a small sample of around 15% of clean tuples from the target dataset is sufficient to train the Lopster models.

Based on the clean sample, we then create dirty tuples by systematically injecting errors into the clean tuples, as shown in Figure 3a. For injecting errors, we use a procedure based on weak supervision [27] that uses a set of user-defined transformation functions $\mathcal{T}^d = \mathcal{T}_1^d, ..., \mathcal{T}_k^d$ where each function injects a different error type. For example, to inject missing values the function might replace cell values by NULL, or for outliers we can use functions that multiplicative increase or decrease cell values (e.g., by 2×, 3×, etc.). We further discuss our set of transformation functions in Section 3.

[1]The clean region (blue data points) is shown in Figure 2 as a 2D-plane for simplicity. In practice, Lopster latent space is higher dimensional to enable rich representations.



(a) Training Procedure (b) Inference Procedure

Figure 3: An overview of training and inference with a Lopster model. (a) For training, we start with the clean tuples and inject errors using the set of user-defined transformation functions $\mathcal{T}_k^d$, creating a set of dirty tuples. Next, we train our model to span up the error dimension by learning from the latent representations of both clean and dirty tuples. (b) For cleaning a dirty tuple, we first encode the dirty tuple in latent space and shift the representation to every available position in latent space along the error dimension, then decode them all. We use a marker value (k_{max}) to decide which representation is the clean one, since the distance from k_{max} to k_0 (e.g., 201.50 in the example) is known.

After generating clean and dirty tuples, we train our *encoder-repair-decoder* architecture as follows: we randomly use pairs (x, x') of clean x and dirty tuples x', where a function $\mathcal{T}_k^d$ is applied to inject an erroneous value into x to create the dirty tuple x'. Next, we map x into latent space in the clean region, apply the shift equivalent to error type k in the error dimension, reconstruct this shifted representation, and compare it to the dirty tuple x'. Consequently, the encoder-repair-decoder learns to inject errors in latent space using shift operations, indexed by k.

Important is that we enforce equivariance for the input domain and the latent space, so we can repair a dirty tuple by simply encoding x' with the encoder, applying a shift in the "inverse" direction of the ones used during training, to move the latent representation back to the clean region, then decode the repaired representation x.

Inference. Once the Lopster models latent space is "shaped" to have an orthogonal error dimension, the task of data cleaning is reduced to first projecting the tuple into the shaped latent space, then shifting the representation to the clean region. However, one question remains: how to identify the distance of the shifts required for the representations to intersect with the clean region? Since the distance of the shift is unclear in higher dimensional latent spaces, and dirty tuples might also be mapped to positions in between errors on the error dimension. Figure 3b depicts our procedure to identify how much to shift in latent space. The basic idea is that we quantize the error dimension by the number of error types K used during training, and we make sure that individual errors are placed at a fixed number of shifts away along the error dimension. Then, during inference, we iteratively shift the latent representation step by step back to the clean data region. The non-trivial part is identifying how many shifts away from the clean region the encoded tuple representation is. For that, we developed a method called *index discovery trick* which relies on a marker value k_{max} that is explained in Section 4.1.

In cases where the representation lands in between our indexed positions, which is true for error types unseen during training, the best representation available in our latent space is the closest we can get to the clean region. It ends up amounting to the same number of shifts than the closest known position. For example, for an unseen error 2.6×, if the closest error (e.g. 3×) we have is indexed at $k = 2$, the model will apply the same number of shifts for both cases. Hence, our encoder-repair-decoder architecture can repair a dirty tuple even if the error was unseen (i.e., it will pick a representation in latent space that is closest to a known error). In contrast, existing data cleaning methods [26, 28] need to learn specific error repair functions for every error type, while Lopster can apply the same error repair method universally, and is thus error type agnostic.

3 LEARNING A LOPSTER MODEL

In this section, we detail the key concepts behind the column-wise encoder-repair-decoder architecture. First, we explain the details of how we enable the shaped latent space that separates clean from dirty data, and the need for equivariance in our model. Next, we discuss the details of learning data cleaning in latent space, and explain the training procedure and goal.

3.1 Why is Equivariance Needed?

Equivariance is a central research in machine learning today. The idea is to train a ML model where transformations in the input space can be replicated directly in the latent space [7, 9, 11, 13, 31]. For example, image-based transformations such as image rotations (or inverse rotations) can be expressed in latent space to augment images. While equivariant models have been intensively studied for such modalities, they have not been applied to tabular data and we believe that they provide important properties, in particular for data cleaning as we show in this paper.

Figure 4 shows the basic idea of an equivariant model used in Lopster for the cleaning of tabular data. In our model error injection in input tuples is mapped to simple shift operations in latent space using a shift matrix as explained in Section 3.2, while inverse shifts are used for cleaning a tuple. As such, a tuple can be cleaned by encoding it with our encoder, applying the inverse shift to the

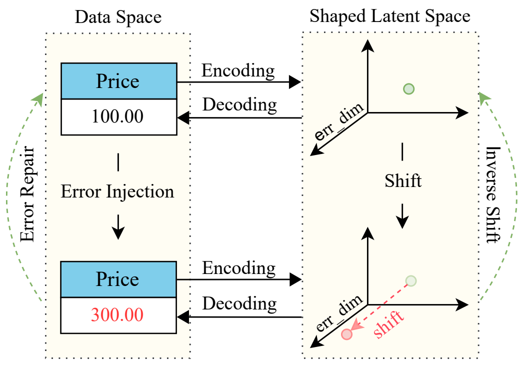


Figure 4: Equivariance in Lopster models. Once the tuple is encoded, the *Inverse Shift* of its representation back to the clean region (within our shaped latent space), is equivalent to any arbitrary *Error Repair* in data space.

encoded latent space representation (multiple times as explained in Section 2), and then decoding the cleaned tuple. The question now is how do we benefit from equivariance? To answer the question, let us contrast our approach to existing learned cleaning approaches that do not rely on equivariance.

In existing learned data cleaning, models learn the error repair by using training data which includes clean and dirty tuples directly on the input data space. However, since error repair functions can be quite diverse, and they heavily depend on the actual error type (i.e., we need a very different function for repairing a missing value and an outlier), the models do not generalize to unseen errors. This is different from cleaning with our equivariant model, where all repairs are multiples of the same inverse shift operation in latent space, i.e., all errors are repaired with the same function. In other words, this equivariant property allows our model to generalize better, as shown in our evaluation.

3.2 Learned Cleaning with Equivariance

We consider tabular data as a set D composed of N tuples with C columns each. Our goal is to transform every dirty tuple $x'_i \in D$ back to the clean tuple x_i. To do so, we chose to first project the input tuples x'_i into a representation z'_i in the latent space of our encoder-repair-decoder architecture, and then solve the data cleaning task in latent space using inverse shifts. Unfortunately, the equivariance between tabular input and latent space transformations is non-trivial for an ML model to learn [31]. Therefore, we first define equivariance in our context and then explain how we achieve it through our training procedure.

In our context, equivariance can be conceptualized as the scenario in which a transformation in the input space, and the corresponding transformation in latent space, are commutative [13]. The formal definition follows [31]: A function $g : D \longrightarrow Z$ is equivariant under a group of transformations $\mathcal{T}$ if for any transformation $\mathcal{T}_k$, the following expression holds for any $k \in K$:

$$\mathcal{T}_k^Z \circ g(x) = g \circ \mathcal{T}_k^D(x) \quad (1)$$

where this equivariance between input D and latent space Z under $\mathcal{T}$ is our model training goal.

As explained before, this property of equivariance is used while training a Lopster model to span up the latent space, where we use a set of K error injection functions $\mathcal{T}_k^d$. More precisely, for spanning up the latent space, we chose a simple set of linear scale transformations (i.e., shifts) during training, where different error types are mapped to different positions along the error dimension. For example, in Figure 5 (upper part), a possible set of injected errors is shown which consists of five different error types defining the a circular group $\mathcal{T}_K^d = \{I, 0.33\times, 0.66\times, 1.66\times, MV, Marker\}$ mod 6, where I stands for the identity transformation, followed by different degrees of outliers which simply multiply a value with a scalar factor, MV for missing or null values, and a marker position at the last index k_{max}.

Figure 5 (lower part) also shows how one instance of our Lopster model can be used for cleaning one column of a table tuple with inverse shifts in latent space. A tuple is first projected into latent space and then shifted. Shifts can be simply implemented in latent space using *shift matrices* (i.e., the SUP) which have shown to be adequate for mapping affine transformations from the input space [5]. The shift matrix SUP we use for a latent space with K error types is defined as follows:

$$SUP = \begin{bmatrix} 0 & 1 & 0 & \dots \\ 0 & 0 & 1 & \dots \\ \vdots & & & \\ 0 & 0 & \dots & 1 \\ 1 & 0 & 0 & \dots \end{bmatrix}_{M \times K}$$

As we see in Figure 5, the encoded representation of a tuple can be seen as the $M \times K$ matrix (Z_1), where $M \times K$ is the dimensionality of the tuple representation in our latent space. Important is that the matrix is structured in K rows each having M columns. Using the shift matrix SUP, the entire encoded representation is shifted up by one row in this matrix. After training, fixing different errors can be implemented as a sequence of k applications of the inverse shift using the transposed matrix SUP^T.

When encoding a tuple, we do not know its position in our error dimension, the output of the encoder is just a matrix where all K rows are filled, which is why we need the *index discovery trick*. In Figure 5, we however visualize the row with a dirty representation as a dark green row to make the idea of the inverse shift visible. In fact, what the inverse shift matrix is doing is that it shifts all rows of the encoded tuple up by one and also wraps around; i.e, the upper row of the encoded input (the $M \times K$ matrix) appears as the bottom row after one shift. As such, to find out how many shifts are needed to clean a tuple with our *index discovery trick*, we need to apply the inverse shift k-times.

To decide how many inverse shifts are needed, we use the marker value on the error dimension to move the dirty representation to row 0 in the $M \times K$ matrix. To be more precise, if we need i shifts to move the latent representation to the marker value, then the clean tuple will be reached after $k_{max} - i$ inverse shifts. In our example in Figure 5, we apply two inverse shifts which brings us to the marker. As such, applying one inverse shift is needed to clean the tuple in our example as shown by the matrix Z_0.

3.3 Training Goal

The orbit of a group of transformations $\mathcal{T}_k$ over the input space D is all the elements created by applying the different error injection functions $\mathcal{T}_k$ on one column of an input tuple $t \in D$. For example, the column *Price* of a tuple with a value of 100 would span the orbit $\{100, 33, 66, 166, NULL\}$, under the set of transformations (i.e., the error dimension) depicted in Figure 5. Ideally, after training an equivariant Lopster model, the orbit of the group of transformations in latent space $h \circ \mathcal{T}_k^z(Z)$ matches the orbit of $\mathcal{T}_k^d(D)$ in input space, where h is our decoder.

In other words, the training goal is for the model to be equivariant to the transformation $\mathcal{T}_k^d$ applied to the input x, and the shift $\mathcal{T}_k^z$ applied to the latent vector z. Hence, the inverse shift $\mathcal{T}_k^{-z}(z)$ is approximately equivalent to applying the inverse transformation $\mathcal{T}_k^{-d}$ to x (i.e., applying the cleaning of a column of the tuple). We include the identity transformation as $\mathcal{T}_0^d$ and $\mathcal{T}_0^z$ in the orbit to allow the model to represent clean inputs. Borrowing the formulation from Dupont et al. [7], given an encoder model g:

$$\mathcal{T}_k^{-d} \circ \mathcal{T}_k^d(x) = \mathcal{T}_0^d(x) = x \tag{2}$$

$$g \circ \mathcal{T}_k^d(x) \approx \mathcal{T}_k^z \circ g(x) \tag{3}$$

following:

$$\mathcal{T}_k^{-z} \circ g \circ \mathcal{T}_k^d(x) \approx g(x) \tag{4}$$

therefore, once the Lopster model is trained we can move back to the clean input position $\mathcal{T}_0^d$ by applying the inverse transformation $\mathcal{T}_k^{-z}$ in z, with the proper index k, and then decoding the latent representation. Notice that the transformations in the latent space are shifts using the shift matrix SUP as discussed before, and the inverse shift is the transposed shift matrix SUP^T.

Finally, it is important to note that the quality of an equivariant model for data cleaning depends on how many error types k we use to span up the latent space and to define the equivalent circular group. However, our model generalizes to different k. If we increase to $k = 8$ errors, the equivalent circular group could be $\mathcal{T}_k^d = \{I, 0.25\times, 0.50\times, 0.75\times, 1.25\times, 1.50\times, 1.75\times, MV\}$ mod 8. In our evaluation, we show the sensitivity of our model to k in an ablation study and evaluate the capacity of our model to generalize to errors that are not in our set of input transformations $\mathcal{T}_k^d$.

3.4 Training Procedure

In the following, we explain the training procedure for training a Lopster model. To enforce equivariance between input tuple errors and shifts of our model latent space, we first generate a dirty pair (x, x') for each tuple, where $x' = \mathcal{T}_k^d(x)$ is the dirty version of the clean tuple x, transformed by $\mathcal{T}_k^d$. The choice of which error transformation to apply to x is based on a random draw from the set $\mathcal{T}^d$, allowing for the Identity where $x = x'$.

Once we have generated several (x, x') pairs, we train the encoder-repair-decoder model using these pairs of clean and dirty tuples. The idea is that we run a clean tuple x through the encoder, shift it by k positions (for the k which was used to inject the error), and then decode the tuple which gives us a new version x''. To enforce that the shift operation in latent space represents the error injection function in the data space, we need to minimize the difference between x' (the dirty tuple in the training data) and x'', which is

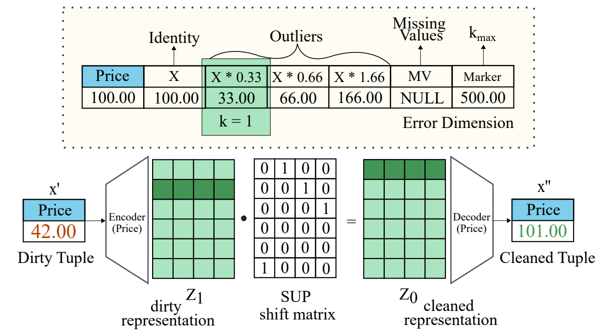


Figure 5: An equivariant Lopster model for the column *Price* and a set of six error types, learned based on the following set of error injection functions: $\mathcal{T}_k^d$ = $\{I, 0.33\times, 0.66\times, 1.66\times, MV, Maker\}$ mod 6, where I stands for Identity (i.e., no error injection), the multiplication factors stand for outliers of different amplitudes, and MV stands for missing values. In addition, we use a marker value in the error dimension. The encoder task is to project the input to one of the Z_k indexed positions (Z_1 in the example). Next, we can move this representation along the error dimension using a shift matrix (SUP) resulting in Z_0. Decoding Z_0 results in the cleaned tuple. The dark green row is for visual clarity.

produced by applying k shifts in latent space, then decoding. Formally, the loss function that is minimized during training is defined as follows:

$$Loss = \sum_{i=1}^{D} ||x' - h \circ \mathcal{T}_k^z \circ g(x)|| \tag{5}$$

where, the functions g and h are our encoder and decoder, respectively. Moreover, k refers to the index of the transformation $\mathcal{T}_k^d$ drawn to transform x into x' for the tuple at hand. In other words, the reconstruction error is the difference between the decoded dirty tuple x', and the decoded tuple x'', coming from the representation z' shifted k times.

As we also include the identity function as part of the error injections, we thus also learn the clean region in the latent space by the same loss. In other words, when the identity function is applied, our model acts as an autoencoder.

3.5 Important Design Considerations

On the order of error transformations. Since our latent space transformations are modular (thanks to the shift matrices), the mapping of the input errors is also cyclic, such as when the maximum index (i.e., k_{max}) is reached, it starts back at position $k = 0$. Moreover, we defined an ascendant sequence of transformations to force the model to fill in the intermediary representations, similar to the temporal coherence explored in [13].

Sorting the indexes of our error injection functions is a way to guide the training procedure into mapping intermediary positions in between the set of transformations $\mathcal{T}_k^z$. For example, $0.5\times$ should be indexed before $1.5\times$ for the model to map intermediary positions such as $0.7\times$ or $1.2\times$. In turn, if the transformations were unordered, each position of our error dimension is learned as a static transformation, and in-between representations yield noise.

On disentanglement. Another common goal is for the transformations in the latent space to be disentangled [11]. Without proper disentanglement between the latent space dimensions, the attributes of the input tuple are all spread throughout multiple latent dimensions, which is not the desired behavior for our method. For example, if we trained our encoder-repair-decoder model without a disentangled operator, as we moved around our error dimension for the column *Price*, it would be entangled with other attributes, such as *Age* or *Income*, and the repair step would affect these attributes unexpectedly. In contrast, the dependence between attributes in the input domain should be preserved. Our error dimension is disentangled from other attributes, but the dependencies learned from data are kept in the non-disentangled dimensions of our latent space, which also helps our model to generalize. A theoretical foundation for disentangled latent operators can be found in [5].

How to set k_{max}?: When setting the marker value k_{max} it is important that we normalize all values to the interval $[0, 1]$, even the categorical columns. Without this normalization, setting k_{max} would indeed be problematic. For example, in a personal information dataset, $k_{max} = 150$ works for the column age, but for the column income it must be much higher. Therefore, to enable the *index discovery trick*, we set k_{max} after normalizing the values to avoid having a marker value that is dependent on the range of values of each column. To be more precise, in all experiments shown in Section 5, k_{max} was set to 3.0 for all columns in all datasets. Noteworthy, in this case if a clean column value at test time is 3× higher than the maximum value seen during training (e.g., for an outlier), the model would mistreat the value as potentially dirty. However, although k_{max} is the last index of our error dimension, it is in no way the max value that our decoder model can yield, as long as the other columns are also higher than seen during training, such that the learned correlation holds.

4 DATA CLEANING USING LOPSTER

In the following, we discuss how we can use a trained Lopster for cleaning errors (i.e., how inference with our models works).

4.1 Error Detection Procedure

Current data cleaning methods also use learned approaches [2]. These methods are typically split into error detection and error repair. The error detection is framed as a binary classification task to flag each attribute of a tuple as either dirty or clean. Afterward, the error is repaired by a separate cleaning model. In contrast to these methods, in Lopster error detection is implicit in our process of indexing tuples in latent space, since any input tuple that indexes $k \neq 0$ is considered dirty. In other words, if the tuple representation does not intersect with the clean data region, the tuple contains errors. Therefore, error detection is abstracted as the task of finding the position to which our encoder projected the input on the error dimension. This index discovery task is non-trivial, as discussed in Section 2. In the following, we explain the procedure for determining k in detail.

As a first step, a tuple is mapped to latent space using our encoder. To be more precise, the encoder $g : D \to Z_k$ yields a latent vector Z_k. The problem is that the index position k in the error dimension is non-trivial to determine. We can not simply measure the "distance" of the latent representation to the clean region, since this is a manifold in high-dimensional space. As such, to find the index k, we adopt an *index discovery trick*: we introduced the marker value k_{max} in the error dimension, mapped to the very last position. Next, we decode every position available in our error dimension. By analyzing each decoded value, we can find in which position the maximum decoded value is, and thus determine which latent representation stands for the marker. Consequently, we also have the distance to the first index $k = 0$, which encodes a clean tuple. Formally, our *index discovery trick* can be expressed as:

$$k_{max} = \arg\max_{k} \{h(Z_k), h(Z_{k+1}), h(Z_{k+2}), ..., h(Z_{k+K})\} \quad (6)$$

where h stands for our decoder model, k is the index of the decoded value, Z_k is the first representation projected by our encoder model, and K is the total number of error injection functions. Discovering the k_{max} position suffices to find the actual position k that the encoder projected the latent representation Z, since the number of shifts from any transformation to the k_{max} is known. For example, if k_{max} is at position 5 during training, and we found it at index 3 on our set, we started at index 2, therefore $k = 2$.

4.2 Error Correction Procedure

Next, given that we know how to determine the index k of the dirty tuple representation Z_k in latent space, we can apply the correspondent inverse shift $\mathcal{T}_k^{-z}$ to return the representation to index $k = 0$, i.e., the clean data region. As a caveat, the indexing process must be applied once per encoder model, i.e., once per column of the tuple. Fortunately, each Lopster model is small by current ML standards (< 10k parameters)[2], and its size is independent of the number of tuples available in the dataset that needs to be cleaned.

Figure 6 shows the complete inference procedure for a single input tuple. In Figure 6 we also depict how our model can handle both categorical and numerical values under the same framework. For *Price* and *Rating* the shift to the clean region is straightforward, by applying the inverse shift $\mathcal{T}_k^{-1}$. In contrast, for the categorical column *City*, values such as New York or Toronto are first mapped to a scalar in the input space using an arbitrary order defined during training. Once mapped to a scalar, the representation of the categorical values can then be encoded and shifted as any numerical value. The same holds for binary columns. In our example, different columns need different number of shifts and *City* is detected to be a clean tuple, so the Identity function is applied.

4.3 Why Lopster works for Unseen Errors?

A Motivating Example. To illustrate Lopster repairing unseen errors, lets look into the missing values repair task. In general, missing values can assume three main forms: missing completely at random (MCAR), missing conditioned on observed data (MAR), or missing conditioned on unobserved data (MNAR). If the type of missing value is known, imputation is considerably easier. For instance, MCAR cells could be removed without introducing bias or increasing the error. Unfortunately, in practice it is hard to determine the error type of how a missing value was generated [32].

[2]We provide more details about the model size in the evaluation in Section 5.

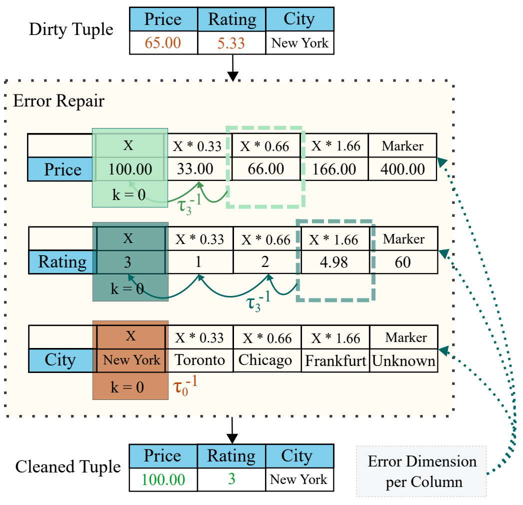


Figure 6: Example of our inference process for a full tuple. For error repair, one Lopster model per column is used to shift the latent representation back to position $k = 0$. Our framework supports numeric, binary and categorical data.

Thus, in Lopster, we clean a tuple containing missing values by reconstructing them from all available signals of that tuple. As long as the majority of input values in a tuple are clean, we have enough signal to restore the clean tuple that most likely represents the input tuple with all values restored. As such, in our framework, all types of missing values can be handled as the same reconstruction task, in which MCAR and MAR cells use information from the other cells seen during training, and MNAR relies on the model having been presented similar tuples during training to extrapolate a value.

Discussion. We now provide a more formal discussion of why Lopster works on error types unseen during training. As stated before, we assume a fixed number of error types during training (similar to other learned methods). However, we first map a tuple into latent space where all types of errors, even unseen ones, can be repaired by a simple shift operation.

The reason is that, in contrast to existing learned methods, within the Lopster framework the latent representation of a tuple with an unseen error still corresponds to the best latent representation available for that tuple; i.e., the encoder will still map the tuple on the error dimension to the latent representation Z_k of the input transformation $\mathcal{T}_k^d$ that is the closest to the unseen error. Hence, the unseen error can be fixed by the same procedure, i.e., inverse shifting k-times.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate Lopster on both cleaning numerical and categorical data, and also evaluate our model in a setting where error types are unseen during training. For the evaluation, we use the data cleaning benchmark REIN [1]. In addition, we show an ablation study where we measure the impact of the parameterization of our model. All numbers shown in this section are the average over five runs, without changing any of the hyperparameters to avoid weight initialization noise.

Table 1: Datasets chosen for our evaluation. Each dataset contains a different challenging property for Lopster: mixed numerical and categorical data in Adult and Soccer, uncorrelated attributes in Bikes and Nasa, time series in HAR, and a large range of numerical values in Smart Factory, including negative ones.

Dataset	#Rows	#Columns	#Categorical	Numerical
Adult	45223	14	8	6
Bikes	17378	15	0	13
HAR	70000	4	0	4
Nasa	1503	6	0	6
Smart Factory	23645	19	0	15
Soccer	180228	37	5	32

5.1 Experimental Setup

Benchmark data. The REIN benchmark serves as a comprehensive public suite for the assessment of data-cleaning methodologies. It encompasses an array of datasets from various sectors, including but not limited to sports, healthcare, and financial records. These datasets are comprised of diverse data types, such as numerical and categorical/binary (i.e., boolean) data. Realistic errors are introduced by an error generator[3] and the BART tool[4].

Our evaluation method in this paper, however, differs from that of the REIN benchmark in that it remains neutral to the specific application that may utilize the cleaned data. While REIN originally focuses on using the cleaned data for training a ML downstream model, we focus on a direct evaluation of the data quality after cleaning (i.e., we do not consider a downstream application). As such, we report the Root Mean Square Error (RMSE) for numerical columns and the F1-score for categorical and binary columns. We provide more details on the used datasets in Table 1.

Error types & methodology. REIN provides a clean dataset and a method for error generation as discussed before, with adjustable parameters such as error types and the percentage of data to be dirty. The errors contain explicit missing values (e.g., empty, null, none), outliers, value swapping (randomly permuting values among cells), and implicit missing values (e.g., 999999, −1). Specifically, implicit missing values refer to invalid replacements for empty cells, usually manually added by the user to avoid having no value, due to software constraints. For the evaluation, we used a subset (i.e., explicit missing values and outliers) of the error types generated by REIN for learning the Lopster models while others were excluded and used only during the evaluation.

Clean training data. We have two main assumptions on the clean data samples that are used for training the Lopster models: (i) We assume the dataset to have at least more than 50% of the rows clean in the sample used for training, and (ii) the data we see during training is IID, i.e., the ranges of values in the data do not change too much from training to inference.

Baselines. In our evaluation, we explore a wide array of data-cleaning methods as baselines which are included in the REIN benchmark. Within REIN, we examined a total of 32 distinct error detection and repair methods, ranging from simple to advanced

[3]https://github.com/BigDaMa/error-generator, last access on 30/10/2024.
[4]https://github.com/dbunibas/BART, last access on 30/10/2024.

methods. The error detection methods have been categorized into non-learning and ML-supported detectors. The former category leverages predefined resources such as user-provided knowledge bases, business rules, integrity constraints, or statistical measures, e.g., dBoost, KATARA, and HoloClean. Each of these methods is typically tailored to tackle particular error types, e.g., duplicates, outliers, or missing values. The second category comprises the methods, e.g., ED2, and RAHA, which conceptualize error detection as a classification problem, utilizing algorithmic learning to classify errors within the data. The REIN benchmark is designed to assess these methods, not only in terms of their intrinsic effectiveness but also their influence on subsequent predictive tasks.

In this paper, we rigorously test every possible combination of error detection and error repair methods. For clarity, we categorize these methods into two groups in our analysis: learned methods (denoted by ML) and non-learned methods (denoted by SI). This categorization is visually represented in Figure 7, where we illustrate the permutations tested. It is important to note that certain methods are exclusively applicable to numerical data, while others can handle both numerical and categorical data. To address this, we introduce the term *numerical − categorical* repair methods to describe those that are versatile in their application. For instance, the abbreviation MF-datawig denotes a hybrid approach that employs MissForest (MF) for numerical error correction and Datawig [4] for categorical data issues, which is a learned method. To ensure clarity and ease of reference throughout our paper, we employ concise abbreviations for the various error repair methods under consideration. For example, we use BR to refer to Bayesian Ridge, MF to denote MissForest, and DT to signify Decision Tree.

Training and inference. Important is that our Lopster models have been trained only on a subset of these error types used in REIN while it has been tested on all of them. This is different from the baseline models which have been trained in a supervised fashion on training data that contains all error types. For inference, we applied our Lopster to the datasets shown in Table 1, containing a mixed set of column types, data domains, and value ranges. Not all error detector and repair combinations are shown across all datasets, due to either being too resource intensive (e.g. Baran [19] out of memory errors) or having worst results that are too far away from the distribution of the remaining methods. Moreover, we do not show non-applicable methods (e.g., methods that do not work on categorical values are not shown on the categorical evaluation).

Setup and Hyperparameters. The code for Lopster has been implemented in Python using Tensorflow as ML backend. The REIN benchmark includes a set of Python scripts for injecting, detecting, and repairing errors using the examined methods. PostgreSQL is used as a data repository, and the FDX profiler [5] to automatically generate functional dependencies for the baselines. Moreover, RAHA expects the user to provide labels. We set the labeling budget to 20× the number of columns, and these user labels are obtained from the ground truth version of the dataset.

The training procedure of Lopster is similar to other learned methods w.r.t. hyperparameters tuning. One addition is the $\mathcal{T}_k^d$ set of input transformations that are going to be equivariant to shifts in the latent space. As aforementioned, the set $\mathcal{T}_k^d$ used across all experiments comprises a series of predefined scalar increases and decreases of values, missing values, and the identity operation, in a circular group. For instance, $\mathcal{T}_k^d = \{I, 0.33\times, 0.66\times, 1.66\times, MV\} mod\ 5$ for $k = 5$, or $\mathcal{T}_k^d = \{I, 0.25\times, 0.5\times, ..., 1.50\times, 1.75\times, MV\} mod\ 8$ for $k = 8$. Unless specified, we kept $k = 12$ (i.e., 12 error types) for all models, which seems to be a good tradeoff between fine- and coarse-grained error sets across all datasets. In our ablation study, we do show an evaluation when varying k.

Independent of the k parameter, we defined a fixed learning rate of 0.001 and 100 epochs for all models. Furthermore, our encoder model contains a single linear layer with z neurons, and our decoder has one hidden layer with 128 neurons and ReLU activation, totaling around 2,000 parameters per model. The scalar z refers to the dimensionality of our latent space, kept as 120 for the REIN experiments. Notice that we have one model per column. Our goal is to show that a very small set of parameters is enough to reach generalizable data cleaning, by applying an equivariant training procedure as in Lopster.

5.2 Numerical Data Cleaning

The quality of numerical columns can be measured by the RMSE between the original clean dataset and the repaired version. Importantly, we do not check for errors in the original data and use them as is from the provided sources as ground truth. Lopster only uses a small sample of clean data (15% of the training data). In contrast to the other baselines, which use the full training data set and require labeled training data for error detection and repair.

Adult. For the first experiment on the Adult dataset (Figure 7a), Lopster achieves the lowest RMSE across all methods, even though it has been trained on fewer clean tuples than the baselines. It is a detection-bound task for this dataset, and we outperform the best alternative dBoost (i.e., Lopster has an average RMSE of 0.70 while the best baseline has an average RMSE of 0.82).

Soccer. For the next experiment, we evaluated our method on the numerical columns of the Soccer dataset, as seen in Figure 7b. This dataset is as challenging as the Adult dataset, with both categorical and numerical columns. In addition, it is the largest dataset with 180,000 tuples and 37 columns. Remarkably, even on this more challenging dataset, Lopster still has the lowest RMSE. Furthermore, as we report in the next Section, it is the only method that can handle the categorical data robustly with the same model architecture. Showing high performance on the large Soccer dataset, while using only 15% of clean data for training, clearly shows the generalization capabilities of our model.

HAR. Next, we evaluated Lopster on the HAR (Human Action Recognition) dataset depicted in Figure 7c. The HAR dataset consists of a set of time series of human actions (e.g. sitting, walking, running). In this dataset, the values are highly correlated to the timestamp. As expected, for a dataset with high column correlations Lopster reaches the lowest RMSE by a margin.

Nasa, Bikes and Smart Factory. We experimented also with three challenging numeric datasets for Lopster: Nasa, Bikes and Smart Factory. Nasa is challenging for learned methods for having only 1,500 tuples. As shown in Figure 7d, Lopster also performs consistently well on Nasa, but using RAHA for error detection shows slightly better performance. However, RAHA only achieves

[5]https://github.com/sis-ethz/Profiler-Public, last access on 30/10/2024.

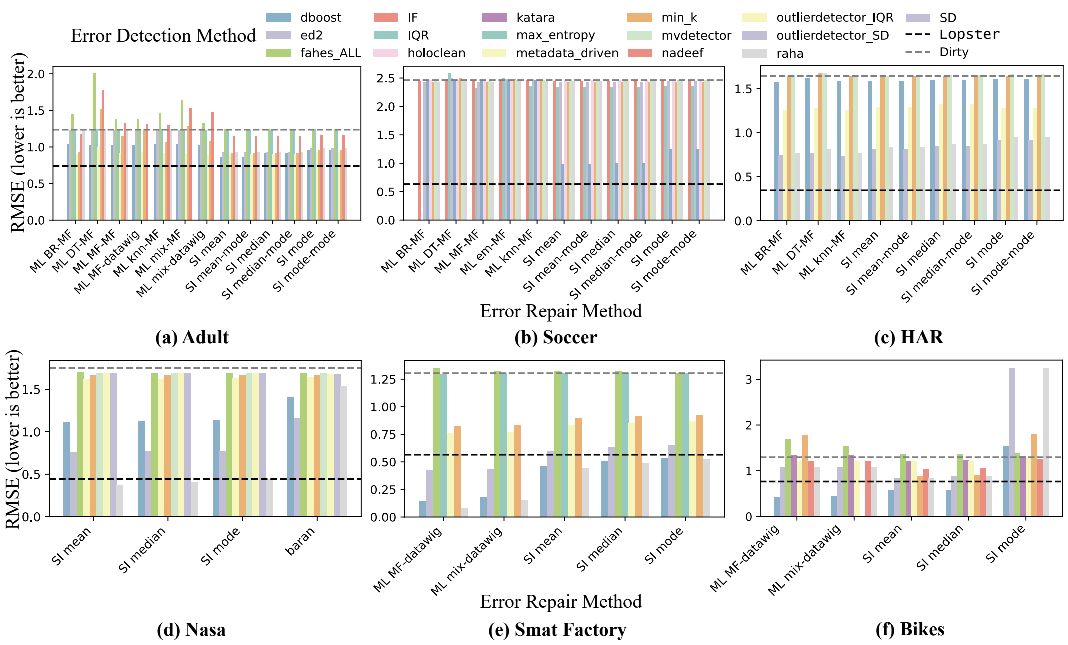


Figure 7: Evaluation on the numerical columns. RMSE (lower is better) of the data cleaning baselines on the numerical columns of all six datasets. Notice that Lopster consistently achieves a low RMSE across all datasets. Moreover, Lopster does not require labeled training data in contrast to all other baselines.

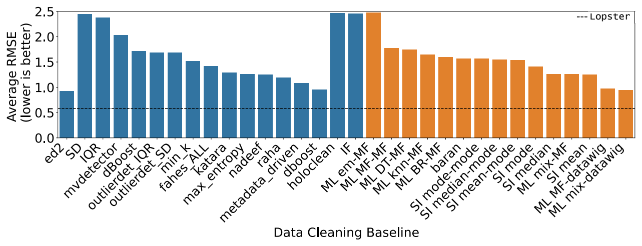


Figure 8: Average RMSE across all datasets (numeric data) for all baselines vs. Lopster. For the baselines, we combine each error repair with all detector methods (blue bars) and combine each detector with all error repair methods (orange bars). The average for Lopster is considerably lower, supporting our domain agnostic claims.

good results when combined with some of the cleaning methods. In Figure 7e we show our results on the Smart Factory numeric dataset, composed of electrical sensor data that varies largely in range. Overall, on this dataset, Lopster is comparable to many of the baselines while some baselines (e.g., raha+datawig) outperformed our model. In turn, different from the baselines, Lopster has not seen all errors and uses a smaller clean sample for training. Finally, on the Bikes dataset shown in Figure 7f, Lopster achieved better performance than most baseline methods. The only exception on this dataset is based on dBoost as the error detector.

Takeaway: Lopster can consistently provide high performance across a wide spectrum of datasets. This is also shown in Figure 8, which averages the results across datasets and clearly shows that Lopster provides robust performance while not requiring labeled training data as the others. Ultimately, Lopster never increases the RMSE compared to the dirty data, which happens to some of the baselines. This consistency is important for practical use, since one cannot verify the improvement provided by the data-cleaning step.

5.3 Categorical Data Cleaning

In this experiment, we show the performance of Lopster vs. baselines on the two datasets that contain categorical data: Soccer and Adult. In contrast to numerical columns, performance on categorical columns is measured by F1-score, comparing clean and repaired versions of the dataset cell by cell. Moreover, there is no notion of order in categorical data. The order used for the error indexing of Lopster is the order of appearance in the dataset.

Therefore, it is a hard scenario for our model that relies on a sequence of shifts to repair the values, since any additional shift may move the representation far enough to change the decoded category and yield a mismatch. Fortunately, the shaping of our latent space induced by our equivariant training procedure suffices, as seen in our results in Figure 9. The reported results are the average across all categorical columns. The same trained models used for the numerical evaluation of Soccer and Adult datasets were used for repairing the categorical columns. Hence, both types of repair models were trained jointly, without any additional hyperparameter tuning. Interestingly, Lopster is the best overall for Soccer, with F1-score of 0.85 against only 0.55 from the second best method HoloClean. Moreover, HoloClean was the only baseline that was able to detect categorical errors in this dataset at all. Our hypothesis is that the error detectors fail to detect the player name, since they

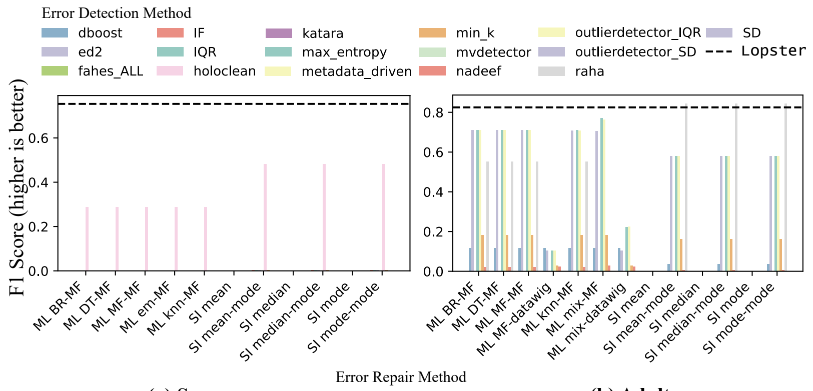


Figure 9: Evaluation on the categorical columns. Average F1-score of Lopster on all categorical columns of the (a) Soccer and (b) Adult datasets. Higher is better. Baselines that are not shown have F1-score close or equal zero, due to the error detection method being unable to detect the dirty tuples. Lopster is the only method to achieve high F1-scores on both datasets.

have to match names to the statistics of each player, which is a hard task. Also, the other two categorical columns are challenging for having weak correlation to the other statistics (e.g. attacking work rate, preferred foot).

Takeaway: Although categorical columns are especially challenging to Lopster, due to the lack of a semantic order between categories, we reached the second-best F1-score in Adult, and the highest one in Soccer by a margin. Also, we used the same models from the numerical evaluation, without any hyperparameter tuning or retraining, showing how general our training procedure is.

5.4 Ablation Study

In the following, we discuss our ablation study for the important hyperparameters (i.e., the number of error transformations K and the latent space size) of Lopster models and show the effects on runtime and accuracy. All other hyperparameters were fixed for this experiment: we used 80 training epochs, 0.001 as the learning rate, and the Adam optimizer to study the effect of the important hyperparameters in isolation.

Number of Error Types During Training. The main hyperparameter for Lopster is the number of error types available during training, defined in the set $\mathcal{T}_k^d$ of size K. In this experiment, we vary K between 4 (few error classes) and 64 (fine-grained error classes). The minimal value is $K = 4$ to allow for the two default transformations, Identity and marker value, in addition to at least one scalar increase and one decrease. Otherwise, the model is not able to map the orthogonal error dimension to an ordered sequence of shifts, it is a necessary inductive bias for "sorting" the transformations. Unfortunately, a single transformation in each direction (reductions and increases) does not suffice, since the model tends to map them as a static transformation, i.e. as jumps from one position to the next in the latent space, without intermediary representations.

The results on REIN for the Adult dataset are shown in Figure 10a. Interestingly, when looking at the RMSE (turquoise line) in Figure 10a, we can see that with only a few error transformations (i.e. $K = 12$) we reach already a low RMSE, while with $K \leq 30$ we see diminishing returns. These diminishing returns are expected, since increasing K above a certain threshold makes the error transformation too fine-grained, i.e., the error types are too similar and Lopster models start overfitting. Moreover, we see that training time (brown line) in Figure 10a increases minimally with larger K.

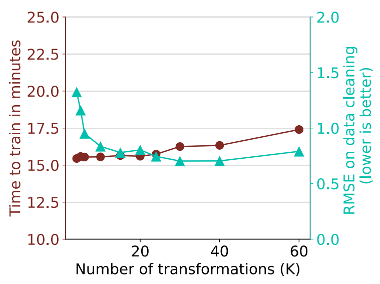
Takeaway: Overall, we argue that a small K such as $K = 12$ is sufficient for Lopster to learn models that provide high accuracy and low training overhead. As such, we have used $K = 12$ throughout our evaluation.

Latent space dimensionality. As a second experiment in the ablation study, we varied the dimensionality of the latent space from 12 up to 480 (per column) as depicted in Figure 10b. While Lopster does not overfit with higher dimensionality since the error transformations add regularization to our training procedure, it also does not benefit significantly from it. For example, going from 120 dimensions to 500 only minimally reduces the error (turquoise line). We argue this because a reasonably-sized model in terms of dimensions suffices to represent the data correlations, and - as reported in other papers [20] - the model "discards" the additional dimensions by encoding redundant information. Furthermore, the training time linearly increases with the dimensionality of the latent space which is also a known effect [20].

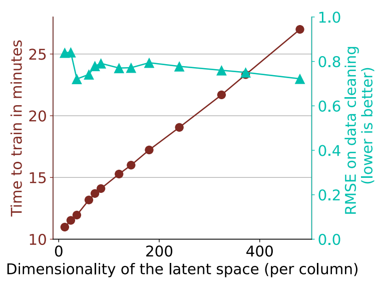
Takeaway: The latent space dimensionality has much less of an impact than K on the error while training time significantly increases. In our evaluation, we thus use 120 dimensions per column throughout all experiments since it provides high accuracy with reasonable runtime overheads.

Inference overhead. Finally, we evaluate the inference runtime overheads vs. RMSE in the Adult dataset (Figure 10c), while varying K. As we see the inference time increases with larger K. This is due to our *index discovery trick*. For each column, we must decode every one of the K available indexes in $\mathcal{T}_k^d$. Once k_{max} is reached we can infer the index of the current tuple in our error dimension. Moreover, the impact of increasing K is much less pronounced than expected, due to the marker value k_{max} being reached on average after half of the transformations.

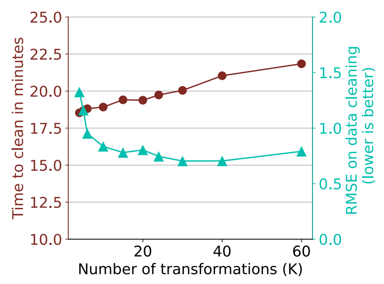
Takeaway: Inference time is clearly impacted by K, but since Lopster models do not require large K, the impact is not significant.



(a) K vs. RMSE & Training Time



(b) Latent space vs. / RMSE & Training Time



(c) K vs. RMSE & Inference time

Figure 10: Ablation studies on the Adult dataset. RMSE and training time in minutes, lower is better. (a) training time vs. Lopster models trained on increasing sizes K of the error transformation group $\mathcal{T}_k^d$. (b) training time vs. Lopster models trained on increasing latent space sizes (per column). Clearly, the hyperparameter that impacts the RMSE the most is K, while the training time slows down with larger latent space sizes. (c) inference time vs. increasing K. Notice that the impact on inference time is lower than expected due to the way we implement the *index discovery trick.*

Clean tuples available for training. To verify if Lopster is indeed more efficient during training, i.e., requires less clean training tuples to learn the error repair than all other baselines, we used clean training samples of different sizes. Figure 11 depicts our results against the best baseline in the Adult dataset numerical evaluation (Figure 7a), the Metadata-driven error detection method. Metadata-driven is a semi-supervised error detection tool that relies on metadata of the input to construct its feature vector. In this paper, we only show the numerical comparison of the Adult dataset, but we see a similar behavior across all datasets, and also for categorical data. As Figure 11 shows, with only 5,000 training tuples out of the 45,000 available, Lopster achieves an improvement over the best baseline of 50% (0.70 vs 1.50). Tuple efficiency during training is a huge advantage of Lopster in practical applications, since acquiring a set of clean representative tuples causes high overheads.

Takeaway: Lopster is more efficient concerning the number of clean training tuples required. In addition, the training time is almost linearly reduced by using fewer tuples.

Qualitative evaluation of the disentanglement claims. We conducted a qualitative evaluation for Lopster, again for the Adult dataset. The goal is to check if our error dimension is indeed disentangled from all the other latent dimensions. Hence, we have defined a procedure that allows us to show that shifts along the error dimensions of individual columns do not affect the values of other columns. The procedure for the qualitative evaluation is composed of two steps: (1) a simulated cleaning step and (2) a step to check disentanglement. In (1) we first encode a given tuple to its latent representation, then shift along our error dimension on a subset of columns (i.e., two in this experiment), simulating a cleaning procedure. Next, on (2) we decoded this shifted tuple back to data space and fed it again to Lopster, this time without applying any shift. The second step is used to test the disentanglement (i.e. if the shift of step 1 on individual columns affects the latent representation of other dimensions in Lopster).

Figure 12 shows results by comparing the distribution (mean and quartiles) for 7 columns of the Adult dataset (due to visual clarity). The same behavior was observed in the full dataset, as well as in the Smart Factory dataset. The green boxes are the distribution of values for each column of the original tuples before applying the procedure. The orange boxes show the distribution of the same columns for the reconstructed tuples (after our two-step procedure). Notice that, as expected, the shift on the last two columns along the error dimension did not affect the other columns, i.e., the distribution of the other columns remains almost unchanged. In contrast, if the representations of these columns were entangled to the error dimension of the two shifted ones, this shift would result in a change of distributions of other columns as well.

Takeaway: Our qualitative evaluation clearly shows the disentanglement between column representation and the error dimension, otherwise the non-shifted columns' distribution would not be unchanged in Figure 12.

6 RELATED WORK

In the following, we discuss related work on learned data cleaning and the origins of equivariant models in computer vision.

Learned Error Detection. Flagging tuples as dirty is the base task for data cleaning, and can heavily influence its quality. In RAHA and Baran [19], the authors propose a semi-supervised approach to error detection called RAHA: it applies a collection of error detection models with different configurations, encodes and clusters cells with similar errors, and then asks the user to label the center point of each cluster. RAHA is the best baseline in our evaluation by a margin, but Lopster is overall better, and requires no user supervision. In contrast to this clustering-based approach, more recent error detection methods, e.g., ED2 [23], HoloDetect [8], and SAGED [2], exploit active learning, data augmentation, and meta-learning to reduce the labeling budget. For instance, HoloDetect, and also TabReformer [22], report on the imbalance between dirty and clean training tuples for training, and thus rely on data augmentation techniques. Closer to our learned method is the Transformer-based PicketNet [18], which applies self-supervision by masking some attributes of the tuple and training the model to fill the gaps. In contrast to PicketNet, Lopster adopts weak supervision by defining the transformations that compose $\mathcal{T}_k^d$.

Learned Error Repair In the deep learning-dominated state-of-the-art, models can tolerate noisy data through regularization, but

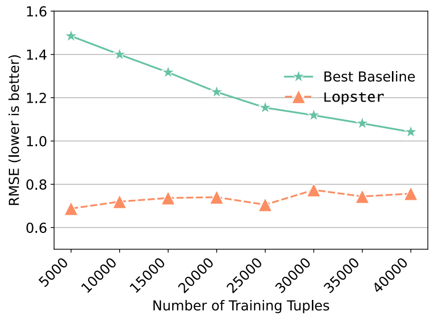


Figure 11: Lopster vs best REIN's baseline on increasing the number of clean training tuples (Adult dataset). Lower is better. We verify the low data hypothesis: Lopster shows lower RMSE even with only 5,000 tuples (9% of the dataset).

only to some extent [17]. Hence, error repair is still a key challenge, which can be extremely time-consuming if done manually [19, 29]. Accordingly, several efforts have been exerted to automate the repair process [12, 14–16, 29]. For example, Baran [19] uses user supervision plus multiple context types, such as cell values and neighbor tuples, to first classify and then repair errors. HoloClean [28] relies on functional dependencies (FDs) to learn statistics over the data and generates a graph of data repair candidates. Similarly, Horizon [29] builds graphs of FDs and then chooses repairs with maximum support. Again, Lopster does not rely on predefined rules, nor on user-supervision.

Other methods propose variations of the Transformer's attention mechanism, such as the successor of HoloClean, AimNet [32], or DeepMVI [3]. Both methods are specific to missing values, and time series respectively. In contrast, Lopster aims at being agnostic to both error types and data domains. Aside from self-attention, another architecture explored for missing values was generative adversarial networks(GANs) in the works DAGAN [17] and Peng et al. [25], with a focus on explainability. In this regard, Lopster can detail the error repair function used, and which error type - among the ones used for training - was the closest to the identified error.

Recently, the use of large language models (LLMs) has emerged as a prevailing trend in different domains. Some table representation LLM models tackle error repair indirectly, mainly for the missing value imputation and string repair tasks [6, 10, 21, 30, 33, 34]. However, so far, these approaches are at the very beginning and they have thus only been used for specific errors and data types, such as text, but not numeric data. Moreover, other works showed that LLMs, originally designed to deal with natural language, still struggle with tabular data engineering tasks [21, 34].

Finally, in practice several benchmarks [1, 24] found tools such as Baran and HoloClean to be time-consuming, due to feature generation or enforcing FDs. Most of the baselines did not terminate within 24 hours on a 32GB memory machine with an 8-core CPU, as reported in [29]. Alternatively, Lopster offers a generalizable data cleaning, with a lower overhead.

Equivariance in Computer Vision. Equivariance is commonplace in the computer vision literature, where developing models that are equivariant to transformations of the input images is an important topic of research [5, 7, 11, 13, 31].

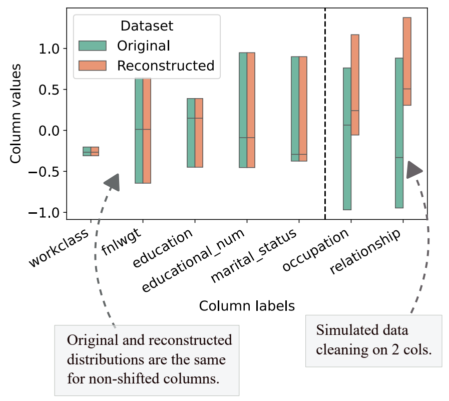


Figure 12: Qualitative evaluation on the Adult dataset with 2 shifted columns. We compare the original tuples distribution by column (green boxes), to partially shifted and then reconstructed tuples (orange boxes). It shows disentanglement between the columns and our error dimension, since only the distributions of the shifted columns change.

Initially, Worrall et al. [31] proposed an interpretable feature space, i.e., a latent space with a notion of order and distance, as a metric space. Building from these initial ideas, Keller and Welling [13] proposed topographically organized latent spaces: models in which activations are spatially organized based on salient features of the input, such as color or scale. Fundamental to our work, the authors argue that *temporal coherence* is the key factor to induce the ML models to group input transformations in a semantically meaningful set. To achieve *temporal coherence* they train using a discrete and ordered set of transformations ("rolls" of the latent space), which must be composed of enough intermediate steps ("slow enough"). We got to the same conclusion in our ablation study on the ideal number of error transformations (Figure 10a).

7 CONCLUSION

In this paper, we presented the Lopster framework that simplifies the data cleaning task by moving it from the input space to the model's latent space. Through our extensive evaluation procedure, we have shown the effectiveness of our model, even under great distribution-shift between error types seen during training and the REIN benchmark. As a potential avenue of future work, we aim to extend our work also towards other data engineering tasks. For example, latent space representations have also been used in data augmentation on images, and we aim to use the idea for tabular data, where the latent space is shaped to represent meaningful augmentations instead of errors. Ultimately we want to investigate how complete data engineering pipelines (e.g., data augmentation, feature selection, data wrangling) can be mapped to latent space transformations, and thus provide generalizability for full pipelines.

ACKNOWLEDGMENTS

This research is funded by the BMBF project AICoM (grant number FKZ: 02P20A064), and partially funded by the LOEWE Spitzenprofessur of the state of Hesse. Moreover, we want to thank hessian.AI, TU Darmstadt as well as DFKI Darmstadt for their support.

REFERENCES

[1] Mohamed Abdelaal, Christian Hammacher, and Harald Schoening. 2023. REIN: A Comprehensive Benchmark Framework for Data Cleaning Methods in ML Pipelines. In *Proceedings of the 26th International Conference on Extending Database Technology (EDBT)*.

[2] Mohamed Abdelaal, Tim Ktitarev, Daniel Staedtler, and Harald Schoening. 2024. SAGED: Meta learning-powered Error Detection Technique for Tabular Data. In *27th International Conference on Extending Database Technology (EDBT)*.

[3] Parikshit Bansal, Prathamesh Deshpande, and Sunita Sarawagi. 2021. Missing Value Imputation on Multidimensional Time Series. *Proc. VLDB Endow.* 14 (2021), 2533–2545.

[4] Felix Biessmann, Tammo Rukat, Phillipp Schmidt, Prathik Naidu, Sebastian Schelter, Andrey Taptunov, Dustin Lange, and David Salinas. 2019. DataWig: Missing value imputation for tables. *Journal of Machine Learning Research* 20, 175 (2019), 1–6.

[5] Diane Bouchacourt, Mark Ibrahim, and Stéphane Deny. 2021. Addressing the Topological Defects of Disentanglement via Distributed Operators. *CoRR* abs/2102.05623 (2021). arXiv:2102.05623 https://arxiv.org/abs/2102.05623

[6] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2020. TURL: Table Understanding through Representation Learning. *Proc. VLDB Endow.* 14, 3 (nov 2020), 307–319. https://doi.org/10.14778/3430915.3430921

[7] Emilien Dupont, Miguel Bautista Martin, Alex Colburn, Aditya Sankar, Josh Susskind, and Qi Shan. 2020. Equivariant neural rendering. In *International Conference on Machine Learning*. PMLR, 2761–2770.

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

[9] Mark Ibrahim, Diane Bouchacourt, and Ari Morcos. 2022. Robust self-supervised learning with lie groups. *arXiv preprint arXiv:2210.13356* (2022).

[10] Hiroshi Iida, Dung Ngoc Thai, Varun Manjunatha, and Mohit Iyyer. 2021. TABBIE: Pretrained Representations of Tabular Data. In *NAACL*.

[11] Jianbo Jiao and João F Henriques. 2021. Quantised Transforming Auto-Encoders: Achieving Equivariance to Arbitrary Transformations in Deep Networks. *arXiv preprint arXiv:2111.12873* (2021).

[12] Bojan Karlaš, Peng Li, Renzhi Wu, Nezihe Merve Gürel, Xu Chu, Wentao Wu, and Ce Zhang. 2020. Nearest Neighbor Classifiers over Incomplete Information: From Certain Answers to Certain Predictions. *Proc. VLDB Endow.* 14, 3 (nov 2020), 255–267. https://doi.org/10.14778/3430915.3430917

[13] T Anderson Keller and Max Welling. 2021. Topographic vaes learn equivariant capsules. *Advances in Neural Information Processing Systems* 34 (2021), 28585–28597.

[14] 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). arXiv:1711.01299 http://arxiv.org/abs/1711.01299

[15] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. 2016. ActiveClean: Interactive Data Cleaning for Statistical Modeling. *Proc. VLDB Endow.* 9, 12 (aug 2016), 948–959. https://doi.org/10.14778/2994509.2994514

[16] Yuening Li, Zhengzhang Chen, Daochen Zha, Kaixiong Zhou, Haifeng Jin, Haifeng Chen, and Xia Hu. 2021. Autood: Neural architecture search for outlier detection. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2117–2122.

[17] Tongyu Liu, Ju Fan, Yinqing Luo, Nan Tang, Guoliang Li, and Xiaoyong Du. 2021. Adaptive Data Augmentation for Supervised Learning over Missing Data. *Proc. VLDB Endow.* 14 (2021), 1202–1214.

[18] Zifan Liu, Zhechun Zhou, and Theodoros Rekatsinas. 2022. Picket: guarding against corrupted data in tabular data during learning and inference. *The VLDB Journal* (2022), 1–29.

[19] Mohammad Mahdavi and Ziawasch Abedjan. 2021. Semi-Supervised Data Cleaning with Raha and Baran. In *CIDR*.

[20] Kien Mai Ngoc and Myunggwon Hwang. 2020. Finding the Best k for the Dimension of the Latent Space in Autoencoders. In *Computational Collective Intelligence*, Ngoc Thanh Nguyen, Bao Hung Hoang, Cong Phap Huynh, Dosam Hwang, Bogdan Trawiński, and Gottfried Vossen (Eds.). Springer International Publishing, Cham, 453–464.

[21] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911* (2022).

[22] Mona Nashaat, Aindrila Ghosh, James Miller, and Shaikh Quader. 2021. TabReformer: Unsupervised Representation Learning for Erroneous Data Detection. *ACM/IMS Transactions on Data Science* 2, 3 (2021), 1–29.

[23] Felix Neutatz, Mohammad Mahdavi, and Ziawasch Abedjan. 2019. Ed2: A case for active learning in error detection. In *Proceedings of the 28th ACM international conference on information and knowledge management*. 2249–2252.

[24] Wei Ni, Xiaoye Miao, Xiangyu Zhao, Yangyang Wu, and Jianwei Yin. 2023. Automatic Data Repair: Are We Ready to Deploy? *arXiv preprint arXiv:2310.00711* (2023).

[25] Jinfeng Peng, Derong Shen, Nan Tang, Tieying Liu, Yue Kou, Tiezheng Nie, Hang Cui, and Ge Yu. 2022. Self-supervised and Interpretable Data Cleaning with Sequence Generative Adversarial Networks. *Proceedings of the VLDB Endowment* 16, 3 (2022), 433–446.

[26] Clément Pit-Claudel, Zelda Mariet, Rachael Harding, and Sam Madden. 2016. Outlier Detection in Heterogeneous Datasets using Automatic Tuple Expansion. (02 2016).

[27] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid training data creation with weak supervision. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 11. NIH Public Access, 269.

[28] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (aug 2017), 1190–1201. https://doi.org/10.14778/3137628.3137631

[29] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Ahmed R. Mahmood, and Michael Stonebraker. 2021. Horizon: Scalable Dependency-Driven Data Cleaning. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2546–2554. https://doi.org/10.14778/3476249.3476301

[30] Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. 2023. DataVinci: Learning Syntactic and Semantic String Repairs. *arXiv preprint arXiv:2308.10922* (2023).

[31] Daniel E Worrall, Stephan J Garbin, Daniyar Turmukhambetov, and Gabriel J Brostow. 2017. Interpretable transformations with encoder-decoder networks. In *Proceedings of the IEEE International Conference on Computer Vision*. 5726–5735.

[32] Richard Wu, Aoqian Zhang, Ihab Ilyas, and Theodoros Rekatsinas. 2020. Attention-based Learning for Missing Data Imputation in HoloClean. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 307–325. https://proceedings.mlsys.org/paper/2020/file/202cb962ac59075b964b07152d234b70-Paper.pdf

[33] Pengcheng Yin, Graham Neubig, Wen tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. *ArXiv* abs/2005.08314 (2020).

[34] Haochen Zhang, Yuyang Dong, Chuan Xiao, and Masafumi Oyamada. 2023. Large Language Models as Data Preprocessors. arXiv:2308.16361 [cs.AI]