A Hybrid Data Cleaning Framework using Markov Logic Networks

Yunjun Gao, Congcong Ge, Xiaoye Miao, Haobo Wang, Bin Yao, Qing Li

Abstract—With the increase of dirty data, data cleaning turns into a crux of data analysis. Most of the existing algorithms rely on either qualitative techniques (e.g., data rules) or quantitative ones (e.g., statistical methods). In this paper, we present a novel hybrid data cleaning framework on top of Markov logic networks (MLNs), termed as MLNClean, which is capable of cleaning both schema-level and instance-level errors. MLNClean mainly consists of two cleaning stages, namely, first cleaning multiple data versions separately (each of which corresponds to one data rule), and then deriving the final clean data based on multiple data versions. Moreover, we propose a series of techniques/concepts, e.g., the MLN index, the concepts of reliability score and fusion score, to facilitate the cleaning process. Extensive experimental results on both real and synthetic datasets demonstrate the superiority of MLNClean to the state-of-the-art approach in terms of both accuracy and efficiency.

Index Terms—data cleaning, MLN, qualitative, quantitative

1 INTRODUCTION

Data analysis benefits from a wide variety of reliable information. The existence of dirty data not only leads to erroneous decisions or unreliable analysis but probably causes a blow to the corporate economy [1]. As a consequence, there has been a surge of interest from both industry and academia on data cleaning [2]. The purpose of data cleaning is generally to correct errors, to remove duplicate information, and to provide data consistency. It usually includes two steps, i.e., error detecting and error repairing. The first step is to find where dirty data hide, and the second one is to correct dirty data detected in the previous step.

Example 1. Table 1 depicts a group of sampled tuples from a dirty hospital information dataset $T$. It contains four attributes, including hospital name (HN), city (CT), state (ST), and phone number (PN). The dataset needs to comply with three integrity constraints, i.e., one functional dependency (FD), one denial constraint (DC), and one conditional functional dependency (CFD).

$$(r_1) \text{ FD: } CT \Rightarrow ST$$

$$(r_2) \text{ DC: } \forall t, t' \in T, -(PN(t, v) = PN(t', v) \land ST(t, v) \neq ST(t', v))$$

$$(r_3) \text{ CFD: } HN(“ELIZA”), CT(“BOAZ”) \Rightarrow PN(“2567688400”)$$

Specifically, the rule $r_1$ means that a city uniquely determines a state, the rule $r_2$ indicates that two hospitals located in different states have different phone numbers, and the rule $r_3$ means that a hospital named “ELIZA” and located in city “BOAZ”, has a specific phone number “2567688400”. Errors appeared in the tuples are highlighted in colored cells, and they can be treated from two different levels, i.e., the schema-level and instance-level [3].

| TID | HN    | CT    | ST   | PN            |
|-----|-------|-------|------|---------------|
| t_1 | ALABAMA | DOTHAN | AL   | 3347938701    |
| t_2 | ALABAMA | DOTHAN | AL   | 3347938701    |
| t_3 | ELIZA  | DOTHAN | AL   | 2567688410    |
| t_4 | ELIZA  | BOAZ  | AK   | 2567688400    |
| t_5 | ELIZA  | BOAZ  | AL   | 2567688400    |
| t_6 | ELIZA  | BOAZ  | AL   | 2567688400    |

The schema-level errors refer to the values that violate integrity constraints. For example, tuples $t_4$, $t_5$, and $t_6$ are violated on the attribute ST w.r.t. $r_1$. The instance level errors contain replacement errors, typos, and duplicates in this example. In particular, the replacement error signifies that a value is incorrectly recorded as another value. That is, the value is completely wrong. For instance, $t_3$'s (CT) being “DOTHAN” is a replacement error, the correct value should be “BOAZ” in this cell. Typos, also called misprints, are caused by the typing process. For example, for $t_2$'s (CT), being “DO” is a typo, and the correct value should be “DOTHAN” in this cell. In addition, duplicates indicate that there are multiple tuples corresponding to the same real entity, e.g., tuples $t_4$, $t_5$, and $t_6$.

Data cleaning methods can be divided into two major categories including qualitative techniques and quantitative ones. The qualitative techniques [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17] mainly rely on integrity constraints to express data quality rules. They detect errors which violate integrity constraints, and repair errors with the principle of minimality (i.e., minimizing the impact on the dataset by trying to preserve as many tuples as possible). Take the dataset in Table 1 as an example. Tuples $t_4$ and $t_5$ are violated on the attribute ST w.r.t. $r_1$. Thus, according to the principle of minimality, it replaces the value “AK” with “AL” on the attribute ST of $t_4$, whereas it fails to repair the attributes CT and PN of $t_3$. In addition, the attribute CT of $t_2$ cannot be repaired since it does not violate any rule. In contrast, the second category containing quantitative techniques...
[13], [19], [20] employs statistical approaches to detect possible errors, and finds probable repair candidates of errors based on the probability theory. Qualitative techniques guarantee that the cleaning results are in accordance with data quality rules, and quantitative techniques ensure that the cleaning results conform to statistical characteristics. Recently, new attempts are made by [21], [22], which combine qualitative and quantitative techniques. Nevertheless, they either consider only one kind of integrity constraints (e.g., FDs), or keep the error detecting and repairing steps in isolation and focus on data repairing process, incurring the redundant computation.

In this paper, we propose a novel hybrid data cleaning framework, termed as MLNClean. It aims to address two key challenges: (i) how to combine the advantages of both quantitative and qualitative techniques to deal with multiple error types; and (ii) how to boost cleaning efficiency as much as possible. In the first place, in terms of the first challenge, we seamlessly integrate data quality rules and Markov logic networks (MLNs) into MLNClean, such that it combines the advantages of both qualitative and quantitative techniques, and it is able to cope with schema-level errors (that violate integrity constraints) and instance-level errors (including replacement errors, typos, and duplicates). Moreover, we present a critical cleaning criterion based on a new concept of reliability score, which is defined by considering both the principle of minimality (using the distance metric) and the statistical characteristics (adopting the weight learning of MLNs).

Regarding the second challenge, we enable MLNClean to seamlessly handle both error detecting and error repairing stages. It helps to avoid redundant computation, and therefore minimizes the computation cost of the entire cleaning process. Furthermore, we develop an effective MLN index to shrink the search space. Specifically, the MLN index is built as a two-layer hash table with each block in the first layer including a set of groups in the second layer. Each block is with respect to a data rule that involves a set of data attributes. One block contains a set of groups. In particular, dirty values within each block are cleaned independently, which does not need the access to the information outside the block. In addition, it is noteworthy that, instead of deciding whether each value is clean or not per time in traditional methods, MLNClean chooses to decide whether one piece of data (w.r.t. several attribute values involving one rule) is clean or not per time. Hence, the efficiency of MLNClean is further gained. In a nutshell, MLNClean has the following contributions:

- Our proposed data cleaning framework MLNClean combines the advantages of both qualitative and quantitative techniques via integrating data quality rules and Markov logic networks (MLNs). MLNClean consists of two major cleaning stages, i.e., cleaning rule-based multiple data versions and deriving the unified clean data, which seamlessly performs error detecting and repairing.
- In the first stage, for a data version w.r.t. each block (built in the MLN index), MLNClean first processes abnormal groups, and then, it cleans errors within one group using a novel concept of reliability score.
- In the second stage, MLNClean unifies the final clean data set based on multiple data versions in the previous stage, where a newly defined fusion score is employed to eliminate conflicts among data versions.
- Extensive experiments on both real and synthetic datasets confirm that MLNClean outperforms the state-of-the-art approach in terms of both accuracy and efficiency.

The rest of this paper is organized as follows. We review related work in Section 2. Then, Section 3 introduces data cleaning semantics as well as some concepts related to Markov logic network. In Section 4, we overview the cleaning framework MLNClean. Section 5 elaborates the two-stage data cleaning process. Section 6 details the distributed version of MLNClean on Spark. In Section 7, we report the experimental results and our findings, and then, we conclude our work with future work in Section 8.

2 Related Work

Existing data cleaning methods can be partitioned into two categories: (i) qualitative techniques and (ii) quantitative ones. The qualitative techniques mainly utilize integrity constraints to clean errors, including ones using FDs [6], [14], [2], [15], or CFDs [3], [11], [13], or DCs [5], [10], [17]. In addition to the above methods that repair data violating only one specific constraint, Temporal [4], LLUANATIC [14], NADEEF [12], BigDansing [15], and CleanM [23] support data cleaning against violations of at least two kinds of those constraints. In particular, Temporal is extended with temporal dimension, to capture the duration information for data cleaning. The generic data cleaning platform NADEEF supports the customization for application-specific data quality problems. It provides a programming interface that allows users to specify multiple types of integrity constraints. BigDansing translates the insights of NADEEF into the Map-Reduce framework. In contrast, the recent work CleanM integrates the physical and logical optimizations used in BigDansing to demonstrate its superiority. It is worthwhile to mention that, CleanM focuses on error detecting regardless of error repairing, but MLNClean considers both error detecting and repairing.

The quantitative techniques use data itself to construct appropriate models and predict repair solutions on the basis of data distributions. ERACER [19], SCARE [20], and ActiveClean [13] are among this group. ERACER is an iterative statistical framework based on belief propagation and relationship-dependent networks. SCARE cleans data by combining the machine learning and probability models. It repairs errors based on the maximum likelihood estimation. ActiveClean is a stepwise cleaning method in which models are updated incrementally rather than retrained, and thus, the cleaning accuracy could gradually increase. The group of methods is more suitable to the cases where the fraction of dirty data is far less than that of clean data. The less the dirty data, the more reliable the learned parameters. Theoretically, large-scale datasets can benefit the sophisticated statistical models.

One hybrid method [21] that combines both qualitative and quantitative techniques is then proposed. However, it only combines FDs and statistical methods without considering other types of integrity constraints. Thereafter, the state-of-the-art method HoloClean [22] unifies several data repair signals including integrity constraints and external data to construct a knowledge-base probabilistic graphical model by using DeepDive [24]. HoloClean aims at error repairing, which employs existing approaches to detect errors. By contrast, our proposed framework MLNClean tackle both error detecting and repairing. As empirically confirmed, MLNClean is superior to HoloClean in terms of both efficiency and accuracy. Also, MLNClean can clean various
instance-level errors (e.g., replacement errors and typos), while HoloClean fails to solve them in some cases.

3 Preliminaries

In this section, we describe data cleaning semantics and some concepts related to Markov logic networks. Table 2 summarizes the symbols used frequently throughout this paper.

A dataset $T$ with $d$ dimensions $A_1, A_2, ..., A_d$ consists of a set of tuples $(t_1, t_2, ..., t_n)$, and each tuple $t_k$ has the value $t_{ki} | A_k \in C(A_k)$ on the attribute $A_k$. $C(A_k)$ denotes the domain of attribute $A_k$. There are usually some integrity constraints that should hold on the dataset $T$, such as functional dependencies (FDs), denial constraints (DCs), as well as conditional functional dependencies (CFDs). In addition, each integrity constraint could be considered as two parts, i.e., the reason part and result part, and the reason part determines the result part. In other words, there is no the same reason to determine multiple different results. As an example, for the rule $r_1$, CT is the reason part, while ST is the result part. CT uniquely determines ST.

According to Markov logic theory [25], every integrity constraint can be converted into a unified form. For ease of presentation, we call the rule in the unified form an MLN rule. The MLN rule is expressed as $l_1 \lor l_2 \lor ... \lor l_n$, where $l_i$ is a literal for $i = 1, ..., n$. A literal is any expression that contains a predicate symbol applied to a variable or a constant, e.g., $CT(v),HN("ELIZA")$. For the rules shown in Example 1, they can be transformed into the following MLN rules, where $l_1, l_2 \in T$.

| IC rule | MLN rule | Ground MLN rules |
|---------|----------|------------------|
| $CT \Rightarrow ST$ | $\neg CT \lor ST$ | $\neg CT("DOTH") \lor ST("AL")$ | $\neg CT("BOAZ") \lor ST("AK")$ |

Table 2 Example of Ground MLN Rules w.r.t. $r_1$

4 MLNClean Framework

In this section, we briefly introduce the procedure of MLNClean. Figure 1 depicts the general framework of MLNClean.

The framework MLNClean receives a dirty dataset together with a set of integrity constraints (ICs). It outputs clean data through three steps, including pre-processing, MLN index construction, and two-stage data cleaning. In the pre-processing phase, MLNClean first transforms integrity constraints into MLN rules, and derives ground MLN rules of each MLN rule based on the dirty dataset. Then, MLNClean builds a two-layer MLN index with a set of blocks in the first layer and a set of groups in the second layer. The MLN index is a vital structure, which helps to narrow the search space of repair candidates for the subsequent data cleaning phase. Next, MLNClean enters the two-stage data cleaning phase, which first cleans multiple data versions independently (with each data version coming from different blocks), and then derives the final unified clean data on top of the previous multi-version data. The procedure of MLNClean is shown in Algorithm 1.

MLN index construction. A MLN index is a two-layer hash table. There are a set of blocks in the first layer, each of which has a set of groups in the second layer. One block corresponds to one MLN rule. In other words, the data attribute values related to a set of ground MLN rules that belong to one same MLN rule are put together to form one block. Thus, the number of blocks is equivalent to the number of MLN rules. For convenience, we call the data attribute values of each
Algorithm 1: The general procedure of MLNClean

Input: a dirty dataset $T$ and a set of data rules $R$
Output: a clean dataset $T^c$

```plaintext
1 $B$ ← $\emptyset$
2 foreach $r_i \in R$ do
3   $B_i$ ← $\emptyset$ // $B_i$: a block w.r.t. $r_i$
4   foreach tuple $t \in T$ do
5     $\gamma_i$ ← $\emptyset$ // $\gamma_i$: a piece of data for $t$ w.r.t. $r_i$
6     $v_i$ ← attribute values of $t$ w.r.t. the reason part of $r_i$
7     $v_r$ ← attribute values of $t$ w.r.t. the result part of $r_i$
8     insert $v_i$ and $v_r$ into $\gamma_i$
9     if there is no group $G_{ij}$ sharing the same $v_i$ from $B_i$
10    then
11       $G_{ij}$ ← $\emptyset$ // $G_{ij}$: a group
12       add $G_{ij}$ to $B_i$
13       add $\gamma_i$ to $G_{ij}$ that shares the same $v_i$
14       insert $B_i$ into $B$
15   endforeach
16   // Two-stage cleaning process */
17   $B_i$ ← AGP($B_i$) // abnormal group process
18   foreach group $G \in B$, do
19     $G$ ← R-score based cleaning
20     $G$ ← RSC($B_i$, $G$)
21   // F-score based conflict resolution
22   $T^c$ ← FSCR($T$, $B$) return $T^c$
```

5 TWO-STAGE DATA CLEANING

In this section, we describe the two-stage data cleaning process in MLNClean, including cleaning multiple data versions and deriving the unified clean data.

5.1 Cleaning Multiple Data Versions

As mentioned in Section 4.1 when cleaning multiple data versions, there are two major missions, i.e., processing abnormal groups and cleaning each group. Through them, schema-level errors (violating integrity constraints) are addressed as well as some instance-level errors including replacement errors and typos.

5.1.1 Processing Abnormal Groups

Based on a MLN index, for a tuple with error(s) in the reason part of a data rule, the corresponding piece of data (i.e., $\gamma$) might erroneously form or belong to a group, and thereby, we call the corresponding group as abnormal group. For example, there is a tuple $t_2$ [CT] being “DOTH” in the sample dataset shown in Table 1. It forms a group $G_{12}$ in the MLN index, as depicted in Figure 2. Indeed, the piece of data being {CT: DOTH, ST: AL} w.r.t. $t_2$ should be in the same group $G_{11}$ containing {CT: DOTHAN, ST: AL}. Hence, $G_{12}$ is actually an abnormal group. To this end, we propose an abnormal group process strategy, termed as AGP, to first identify those abnormal groups and then merge them to the corresponding normal groups.

First, we observe that, the group size and the distance to other groups within the same block are key factors for abnormal group. In general, the relatively small groups (that have less pieces of data) are prone to be abnormal. The closer to other groups, the most likely to be abnormal. Thus, AGP adopts a simple but effective method to identify abnormal groups. Specifically, if the number of tuples related to $\gamma$ contained in a group is not larger than a threshold $\tau$, AGP regards this group as an abnormal group; otherwise as a normal group.

For instance, in terms of the MLN index shown in Figure 2 when setting $\tau$ as 1, groups $G_{12}$, $G_{22}$, and $G_{31}$ are identified as abnormal groups. Then, for $G_{12}$, its nearest group is $G_{11}$ based on the Levenshtein distance. Hence, $G_{12}$ is merged with $G_{11}$. Similarly, $G_{22}$ is merged with $G_{23}$, and $G_{31}$ is merged with $G_{32}$. 

Fig. 2. Illustration of the MLN index over the sample dataset

The general procedure of MLNClean.
The time complexity of AGP is $O(|B| \times |G_a| \times (|G| - |G_a|))$, where $|B|$ is the total number of blocks, and $|G|$ (and $|G_a|$) is the average number of groups (and abnormal groups) in a block. In addition, it is worth mentioning that, how to identify abnormal groups as accurately as possible is essential to the overall performance of the cleaning framework, since this step has the biggest propagated impact to the final cleaning accuracy. Therefore, we are going to conduct in-depth exploration about this issue in our future work.

5.1.2 Cleaning within Each Group

In the constructed MLN index, one group shares the same value(s) on the reason part of the corresponding rule. Ideally, if data are clean, one group contains only one and only one piece of data, meaning that the same values on the reason part cannot derive different values on the result part. However, when one group contains several pieces of data (that are the same on the reason part), there definitely exist dirty values. In light of this, we present a cleaning strategy using a concept of reliability score, called r-score based cleaning (RSC) for short, to clean dirty values within each group.

RSC judges which piece of data included in the group is clean using the reliability score. Then, RSC corrects the other dirty ones with the detected clean one, so that each group has one and only one piece of data eventually. It is noteworthy that, RSC cleans the pieces of data within every block independently, which does not need the information outside the block. Actually, we can even know that, RSC is executed to clean data within each group separately, if regardless of Markov weight learning. In other words, what we would like to highlight here is that, the MLN index structure indeed helps to minimize the search space for RSC.

The concept of the reliability score is stated in Definition 2 which is defined to evaluate the possibility degree of the piece of data (i.e., $\gamma$) being clean. The $\gamma$ with the highest reliability score is most likely clean. As a result, all the other pieces of data within the same group are replaced with the piece of data $\gamma$ having the highest reliability score.

**Definition 2.** (Reliability score). For a piece of data $\gamma_i$ in a group $G$, its reliability score, denoted as r-score($\gamma_i$), is defined in Eq. (1):

$$r\text{-score}(\gamma_i) = \min_{\gamma \in (G - \{\gamma_i\})} dist(\gamma_i, \gamma) \times Pr(\gamma)$$

where $dist(\gamma_i, \gamma) = \frac{1}{2} \cdot d(\gamma_i, \gamma)$, $Z$ is a normalization function to make $dist(\gamma_i, \gamma)$ within the interval $[0, 1]$. $\gamma$ denotes the number of tuples related to $\gamma_i$ in group $G$, and $Pr(\gamma)$ is the probability of $\gamma_i$ being clean.

The definition of reliability score considers two factors: (i) **Distance**, which represents the minimum cost of replacing a $\gamma_i$ with others. The greater the distance is, the more likely the $\gamma_i$ is clean. (ii) **Probability**, which indicates the possibility of the $\gamma_i$ being clean. The higher the probability is, the more likely the $\gamma_i$ is clean.

We attempt to derive the probability $Pr(\gamma_i)$ in Definition 2 as follows. First, the probability distribution of values $x$ specified by the Markov network is given by Eq. (2) from [25].

$$Pr(x) = \frac{1}{Z} \exp \left( \sum_{i=1}^{N} w_i n_i(x) \right)$$

where $Z$ is the normalized function, which can be treated as a constant, $n_i(x)$ is the number of true groundings of rule $r_i$ in $x$, $w_i$ is the weight of $r_i$, and $N$ represents the number of rules. When it computes $Pr(\gamma_i)$ involving ground MLN rules in our case, $n_i(\gamma)$ equals one for its corresponding ground MLN rule, and $n_j(\gamma)$ is zero for other ground MLN rules. Hence, we have

$$\ln Pr(\gamma_i) = w_i - \ln Z$$

In particular, $\ln Z$ is a constant, and $\ln Pr(\gamma_i)$ is a monotonically increasing function. Thus, the higher the probability $Pr(\gamma_i)$, the greater the weight $w_i$. As a result, instead of deriving the probability $Pr(\gamma_i)$ directly, we leverage the weight $w_i$ of $\gamma_i$ to compute a

### Table 1

| TID | CT | ST | Min dist | Markov weight | R-score |
|-----|----|----|----------|---------------|---------|
| $t_1$ | BOAZ | AK | 0.1 | 0.414 | 0.0414 |
| $t_{3,6}$ | BOAZ | AL | 0.2 | 0.416 | 0.0832 |

![Figure 3](image)

Fig. 3. Illustration of reliability score computation

reliability score, i.e., $r$-score($\gamma_i$) = $\min_{\gamma \in (G - \{\gamma_i\})} dist(\gamma_i, \gamma) \times w_i$.

In implementation, the weight $w_i$ is computed by MLN weight learning method from Tuffy [26], which adopts diagonal Newton method. Particularly, the prior weight $w_i^0$ of each $\gamma_i$ for weight learning is given by

$$w_i^0 = \frac{c(\gamma_i)}{\sum_{j=1}^{|G|} c(\gamma_j)}$$

where $c(\gamma_i)$ represents the number of tuples related to $\gamma_i$, and $M$ denotes the number of different $\gamma_i$s within a block. For example, for the piece of data being $\{CT: BOAZ, ST: AK\}$ in group $G_{13}$ from block $B_3$, the initial weight is set as $1/6$.

**Example 2.** Take the group $G_{13}$ belonging to block $B_3$ (as depicted in Figure 2) as an example. $G_{13}$ includes two pieces of data, denoted as $\gamma_1$ and $\gamma_2$, namely, $\gamma_1$ is $\{CT: BOAZ, ST: AK\}$ and $\gamma_2$ is $\{CT: BOAZ, ST: AL\}$. They have the same value on the reason part but different values on the result part. Obviously, there is at least one error at the value of attribute ST within the group, according to the data rule $r_1$. The reliability score of each $\gamma_i$ is derived, as shown in Figure 3 where the Levenshtein distance is used. The piece of data $\gamma_1$ being $\{CT: BOAZ; ST: AL\}$ has higher reliability score than $\gamma_2$ being $\{CT: BOAZ; ST: AK\}$. Thus, $\gamma_1$ is regarded as the clean one in this group, and $\gamma_2$ is finally replaced with $\gamma_1$ by RSC strategy.

In addition, for ease of understanding, Figure 4 illustrates three clean data versions after adopting AGP and RSC strategies consecutively. In particular, the group $G_{23}$ from $B_4$ is skipped to calculate the reliability score, because this group has reached the ideal state with only one $\gamma_i$ in it. Finally, the first clean data version contains $\{CT: DOThan, ST: AL\}$ in $G_{13}$ (w.r.t. $t_1, t_2$, and $t_3$) and $\{CT: BOAZ, ST: AL\}$ in $G_{13}$ (w.r.t. $t_4, t_5$, and $t_6$). The second clean data version incorporates $\{PN: 3347928701, ST: AL\}$ in $G_{23}$ (w.r.t. $t_1$ and $t_2$) and $\{PN: 2567688400, ST: AL\}$ in $G_{23}$ (w.r.t. $t_3, t_4, t_5$, and $t_6$). The third clean data version consists of $\{HN: ELIZA, CT: BOAZ, PN: 2567688400\}$ in $G_{32}$ (w.r.t. $t_3, t_4$, and $t_5$).

5.2 Deriving the Unified Clean Data

The first cleaning stage of MLNClean has obtained the ruled-based multiple clean data versions. Now, we are ready to enter the second cleaning stage. It aims to execute the data fusion on top of multi-version data, in order to get the final clean data, in which data conflicts among different data versions have to be solved. Note that, this stage provides the second opportunity to clean erroneous data that are not (or incorrectly) repaired in the first cleaning stage as many as possible.

Take the tuple $t_3$ depicted in Table 1 as an example. After finishing the first cleaning stage, the piece of data w.r.t. $t_3$ in $B_1$ (w.r.t. the first data version) is $\{CT: DOThan, ST: AL\}$, while in $B_3$ (w.r.t. the third data version), the piece of data related to $t_3$ is $\{HN: ELIZA, CT: BOAZ, PN: 2567688400\}$. It is obvious that, $t_3$’s [CT] has two different values (i.e., “DOThan” and “BOAZ”) from the two versions. That is to say, there exist conflicts on attributes CT of $t_3$, and that should be eliminated to get the final clean $t_3$. Besides, although $\{CT: DOThan, ST: AL\}$ conforms to the rule $r_1$, there might still exist errors w.r.t $t_3$ in the case that it is erroneously classified into a group and thereby cannot be correctly repaired in the first cleaning stage.

Accordingly, during executing the data fusion on top of multi-version data versions, we identify a set of candidates to solve conflicts, which refers to all the possible fusion versions for a tuple. Hence, we introduce a novel concept of fusion score (i.e., $f$-score) to get the most likely clean fusion version. Specifically, for a tuple $t$, the fusion
The function GetFusionT in Algorithm 2 works in terms of tuple $t'\in T$ to replace the corresponding attribute values of $t$ with $t'_{f_{\text{max}}}$. In the sequel, it invokes itself at line 24 to merge with the other version of $t$. 

**Example 3.** We illustrate how FSCR works in terms of tuple $t_3$ in the sample dataset, based on three versions (denoted by $\gamma_1$, $\gamma_2$, and $\gamma_3$) of tuple $t_3$ (obtained from the first cleaning stage). In particular, $\gamma_1$ denotes \{CT: DOTHAN, ST: AL\} from block $B_1$, $\gamma_2$ is with respect to \{PN: 2567688400, ST: AL\} from $B_2$, and $\gamma_3$ represents \{HN: ELIZA, CT: BOAZ\} from $B_3$. Hence, following Algorithm 2 for the tuple $t_3$, FSCR gets $V(t_3) = \{\gamma_1, \gamma_2, \gamma_3\}$. For simplicity, we show two fusion attempts: (i) merging $\gamma_1$, $\gamma_2$, and $\gamma_3$ in order, and (ii) merging $\gamma_2$, $\gamma_3$, and $\gamma_1$ in order. For the first attempt, $\gamma_1$ and $\gamma_2$ are merged directly, since there is no conflict between them. Then, it proceeds to merge with $\gamma_3$ from block $B_3$. While there is a conflict on the attribute CT, here GetFusionT tries to find another $\gamma'$ from block $B_3$, such that $\gamma'$ has the same value on the common attribute CT for the fusion of $\gamma_1$ and $\gamma_3$. Unfortunately, there does not exist such $\gamma'$ in block $B_3$. Thus, GetFusionT terminates the current fusion. For the second attempt, $\gamma_2$ and $\gamma_3$ are firstly merged, as there is no conflict between them. Then, GetFusionT merges the fusion of $\gamma_2$ and $\gamma_3$ with $\gamma_1$ from block $B_1$. However, there is a conflict on the attribute CT. At that time, GetFusionT successfully finds a $\gamma'$ being \{CT: BOAZ, ST: AL\} from block $B_1$. Finally, the final fusion version of $t_3$, i.e., \{HN: ELIZA, CT: BOAZ\} from $B_3$, is obtained.

Let $|T|$ be the number of tuples in the dataset and $m$ be the average number of versions for a tuple. $m$ is bounded by the number of blocks/rules. There are at most $m!$ fusion versions for a tuple.
Algorithm 3: Data partition method

Input: a dataset $T$, a data partition $\mathcal{P} = (P_1, \ldots, P_k)$
Output: the data partition $\mathcal{P}$

1. $s \leftarrow \lceil \frac{|T|}{n} \rceil$
2. initialize each part $P_i$ as an empty max-heap for $i = 1, \ldots, k$
3. randomly select a centroid $o_i$, and insert it into the part $P_i$ for $i = 1, \ldots, k$
4. collect those $k$ centroids into $O$
5. for each $t_i \in (T - O)$ do
6.   find $P_j$ with min $dist(t_i, o_j)$
7.   if $|P_j| < s$ then
8.      insert $t_i$ into the part $P_j$
9.   else
10.      $t_e \leftarrow t_i$; get the top node $t_{top}$ of $P_j$
11.     if $dist(t_i, o_j) < dist(t_{top}, o_j)$ then
12.        $t_e \leftarrow t_{top}$; replace $t_{top}$ with $t_i$
13.     find $P_i$ with $\min_{o_j \in O} dist(t_e, o_j)$
14.    add $t_e$ to $P_i$
15. return $\mathcal{P}$

Fig. 5. Illustration of data partition

Each fusion version needs $O(m)$ time to find the candidate version $\gamma'$. Therefore, FSCR takes $O(|T| \times m \times n)$ time. In addition, after eliminating conflicts via FSCR, MLNClean automatically detects and removes duplicate tuples. Take the sample dataset as an example, $t_1$ and $t_2$ are duplicates, and $t_3$, $t_4$, $t_5$, and $t_6$ are duplicates. MLNClean deletes extra duplicate tuples among them to finish the cleaning process.

6 DISTRIBUTED MLNCLEAN

In order to enable MLNClean to work well even for large-scale datasets with a large number of data rules, we aim to deploy MLNClean in the Spark system. In this section, we describe the distributed MLNClean program.

First, data skew is a critical issue in the distributed system, which may lead the overall process to delay. Hence, an effective data partition strategy is needed for the distributed MLNClean version. As a result, the distributed MLNClean version executes in the following procedure. It first partitions the whole dataset into several parts, and allocates each part to a worker node. Then, it cleans each part using the stand-alone MLNClean. When each part has been cleaned, those parts are gathered to derive the final clean dataset, during which conflicts and duplicates are eliminated in the same way to stand-alone MLNClean.

Our data partition method is depicted in Algorithm 4, which aims to divide the dataset $T$ into $k$ parts, denoted as $\mathcal{P} = (P_1, P_2, \ldots, P_k)$. The maximum capacity of each part is set as $\lceil \frac{|T|}{n} \rceil$ (denoted by $s$) for uniform distribution (line 1). Tuples are stored in a maximum heap for each part, where each node also records the distance between the corresponding tuple and the centroid of the part. Then, each tuple $t_i$ is allocated to the part $P_j$, if the distance between $t_i$ and $o_j$ is minimal and the number of tuples in $P_j$ is smaller than $s$ (lines 6-8). When the size of $P_j$ is larger than $s$, if the distance $dist(t_i, o_j)$ is smaller than the distance from the top node (denoted by $t_{top}$) to the centroid of this part $P_j$ (i.e., $dist(t_{top}, o_j)$), $t_{top}$ is replaced with $t_i$ in the part $P_j$. Meanwhile, $t_{top}$ is inserted into its closest partition $P_i$ if it is not full. On the other hand, if $dist(t_i, o_j) \geq dist(t_{top}, o_j)$, $t_i$ is directly added to its closest part $P_i$ that is not full (lines 10-14).

Take the sample dataset shown in Table 1 as an example. Figure 5 illustrates the partition result. The inner digital of each part denotes the distance between the tuple and the centroid. For instance, the digital 4 of partition $P_2$ represents the distance between tuple $t_3$ and the centroid $t_2$.

The time complexity of data partition algorithm is $O(|T| \times \lg s)$, where $|T|$ is the total number of tuples in the dataset, and $O(\lg s)$ is the complexity of one insertion operation in a maximum heap.

For the distributed MLNClean program, due to the small-scale of tuples allocated in each part, the result of every weight learning on each worker (w.r.t. each part) might not be very reliable. For example, in the part $P_2$, the learned weight of $\gamma$ being $\{\text{CT}: \text{DOTHAN, ST}: \text{AL}\}$ may be unreliable, since there is no relevant evidence for learning the weight of $\gamma$. Conversely, in the part $P_1$, there is a tuple $t_3$ providing the relevant evidence for learning the weight of $\gamma$. Thus, we adjust the weight of each $\gamma$ via Eq. 6 in virtue of the relevant evidence from other parts.

$$\overline{w}(\gamma) = \frac{\sum_{i=1}^{k} n_i \times w_i}{\sum_{i=1}^{k} n_i}$$

where $k$ is the number of parts in the data partition, $n_i$ is the number of tuples related to $\gamma$ in the part $P_i$, and $w_i$ represents the learned weight of $\gamma$ in the part $P_i$. As a result, each $\gamma$ corresponds to a unique weight $\overline{w}$ in global, and it is used for the subsequent cleaning process.

7 EXPERIMENTS

In this section, we present a comprehensive experimental evaluation. In what follows, we evaluate our proposed data cleaning framework MLNClean using both real-world and synthetic datasets in the following scenarios: (i) the experimental comparisons between MLNClean and the state-of-the-art method HoloClean [22], (ii) the effect of various parameters on the performance of MLNClean, and (iii) the performance of distributed MLNClean version on the Spark platform.

7.1 Experimental Setup

In the experiments, we use two real-world datasets, i.e., HAI and CAR, and one synthetic dataset, i.e., TPC-H.

HAI [11] is a real dataset that provides information about healthcare associated infections occurred in hospitals. It contains 231,265 tuples.

CAR [12] contains the used vehicle information, including model, make, type, year, condition, wheelDrive, doors, and engine attributes. It consists of 30,760 tuples.

TPC-H [13] is a benchmark for performance metrics for systems operating. The two largest tables including lineitem and customer tables are utilized to create a synthetic dataset. It contains 6,001,115 tuples.

1. https://data.medicare.gov/data/hospital-compare
2. https://www.cars.com/
3. http://www.tpc.org/tpcch/
Table 4 summarizes integrity constraints of each dataset used in our experiments, and they are given by domain experts. We add errors randomly, including typos and replacement errors, on attributes related to integrity constraints shown in Table 4 for each dataset. Specifically, we randomly delete any letter of an attribute value to construct a typo. For a replacement error, we replace a value with another value from the same domain. For each dataset, we generate 5% error rate by default, including a half fraction of typos and another half fraction of replacement errors. It is necessary to mention that, enterprises typically find data error rates are approximately 5% [27], and the reported error rates are no more than 30% in many case studies [28]. Note that, the error rate is defined as the number of erroneous values to the number of total attribute values. In addition, we use the Levenshtein distance as the distance metric, unless otherwise stated.

We utilize F1-score to evaluate the accuracy of data cleaning methods. Specifically, it is defined as

\[
F1-score = \frac{2 \times precision \times recall}{precision + recall}
\]

where precision is equal to the ratio of correctly repaired attribute values to the total number of updated attribute values, and recall equals the ratio of correctly repaired attribute values to the total number of erroneous values. In addition, unless otherwise stated, the experiments were conducted on a Dell PowerEdge T620 with one Intel(R) Xeon(R) E5-2620 v2 2.10GHz processors (6 physical cores and 12 CPU threads) and 188GB RAM.

7.2 Comparisons with HoloClean

In this section, we verify the performance of MLNClean and HoloClean. Since HoloClean adopts external modules for error detection and it can only fix errors caught by the error detection phase, we set the detection accuracy of HoloClean as 100% for an absolutely fair evaluation, which helps avoid the effect of the detection accuracy on the subsequent error repairing phase.

Effect of error percentage. We change the error percentage from 5% to 30%, and report the corresponding experimental results in Figure 6. As observed from Figure 6(a) and Figure 6(b) for both MLNClean and HoloClean, the accuracy decreases slightly as the error percentage increases. For MLNClean, there are two reasons for the decline. The first reason is that, with the increase of error percentage, AGP is prone to wrongly treat more normal groups as abnormal ones, and the following cleaning steps are subject to the chain reaction of AGP, resulting in degraded accuracy. The second reason originates in statistical characteristics of RSC, which employs the reliability score based on Markov weight learning. The larger error percentage leads to the less reliability of the learned weight. On the other hand, HoloClean separates the whole dataset into noisy and clean parts. It uses clean values which are picked by error detection methods to learn the statistical model parameters. Then, it employs the trained model to infer the probability of each noisy value. When error rate increases, the statistical difference between the noisy and clean parts enlarges, which incurs the unsuitable parameters for the inference of noisy values. The results also show that, MLNClean has much higher accuracy than HoloClean for all cases, which reflects the superiority of the two-stage cleaning of MLNClean. This is because, when erroneous values become more and more, HoloClean relying solely on probabilistic reasoning is becoming weaker. However, MLNClean considering both statistical characteristics and the principle of minimality is much stronger, thereby it has better performance.

As shown in Figure 6(c) and Figure 6(d) in terms of the execution time, we can observe that the time cost increases when the error percentage is growing for both MLNClean and HoloClean. Note that, the overall runtime of MLNClean includes both error detection time and repairing time, but the total runtime of HoloClean only involves the error repairing time due to its property. For MLNClean, the growth of the total time cost mainly results from the Markov weight learning, which occupies almost 95% of the total time. Specifically, the increase of error percentage makes it more difficult to determine whether a value is clean or erroneous, thus leading to the slower convergence of Markov weight learning. For HoloClean, the runtime is mainly determined by the compile and repair phase. In particular, for more errors, the candidate set of each value turns larger, incurring more overhead.

Furthermore, MLNClean is consistently faster than HoloClean, even though MLNClean deals with both error detection and repairing stages and HoloClean only focuses on the error repairing. The superiority of MLNClean comes from the cleaning scheme of MLNClean. In MLNClean, the smallest unit of data cleaning is a piece of data, i.e., γ, which contains multiple attribute values. Thus, it is able to clean multiple values per time. However, in HoloClean, the minimum cleaning unit is a single attribute value, namely, it only cleans one value per time. Hence, it needs longer time for HoloClean to clean all errors.

Effect of error type ratio. In order to investigate the effect of different error types on the performance of MLNClean and HoloClean, we vary the error type ratio, and set the total error percentage as 5% by default. We consider two error types, i.e., replacement errors and typos. Specifically, we change the proportion of replacement errors to the 5% total errors, denoted as ret, from 0 to 100%. In particular, ret being zero means that there is no replacement error in the 5% total errors, namely, all the 5% total errors are typos. ret being 100% indicates that all the 5% total errors are replacement errors.

The corresponding experimental results are plotted in Figure 7. We can observe that, HoloClean is rather sensitive to the error type ratio ret on the CAR dataset. In contrast, the performance of MLNClean is stable on both datasets. The reason behind mainly comes from two aspects. From the method aspect, as explained previously, HoloClean trains the model using the clean part, and infers values in the dirty part using the trained model to correct errors. Following the generation methods, the replacement errors are incorrect values from the corresponding same domain, and thus, they exist in both clean and dirty data parts. In contrast, typos are absent from the clean part, which leads the trained model to be weak for many typos. Thus, HoloClean is supposed to be sensitive to typos. From the dataset aspect, CAR is rather sparse while HAI is relatively much dense. Hence, HoloClean is much sensitive to ret over CAR than that over HAI. The F1-score of HoloClean in CAR shows a growing trend when varying ret from 0 to 100%. Especially when there are only typos errors in the data set, the cleaning result is the worst. On the other hand, MLNClean fully considers both types of errors via the two-stage cleaning strategy. Thus, it is much stable with the change of ret. It further confirms...
the superiority of MLNClean. In addition, it is necessary to mention that, the execution time is not very sensitive to the error type. Hence, we omit the related description due to the space constraint.

7.3 Results on MLNClean

In this section, we study the effect of different parameters (i.e., the value of threshold in AGP strategy, the total error percentage, and the distance metric) on the performance of MLNClean. Especially, for in-depth investigation, we also explore the effect of parameters on the three components of MLNClean, including AGP strategy, RSC method, and FSCR strategy, each of which has an impact on the data cleaning accuracy of MLNClean. In particular, in order to appropriately measure the accuracy of each component, we introduce a series of metrics for them.

For AGP strategy, we define Precision-A as the fraction of correctly merged abnormal groups over the total number of detected abnormal groups, and Recall-A as the fraction of correctly merged abnormal groups over the total number of real abnormal groups. For RSC method, Precision-R is defined as the ratio of correctly repaired \( \gamma_s \) to the total number of repaired \( \gamma_s \), and Recall-R is equal to the ratio of correctly repaired \( \gamma_s \) to the number of \( \gamma_s \) which contain errors. In addition, for FSCR strategy, Precision-F (resp. Recall-F) corresponds to the fraction of correctly repaired attribute values by FSCR over the number of erroneous attribute values that include detected conflicts (resp. the total number of erroneous attribute values).

7.3.1 Effect of Threshold

Effect on the performance of AGP. First, we evaluate the performance of abnormal group process (i.e., AGP) strategy when varying the value of threshold \( \tau \). The corresponding results are shown in Figure 6. Note that, we also report the total size of pieces of data (i.e., \#dag) within detected abnormal groups by AGP under different thresholds. For simplicity, we call it the number of detected abnormal \( \gamma_s \) (i.e., \#dag for short).

The first observation is that, the accuracy of AGP (both precision and recall) first ascends and then drops as the value of \( \tau \) grows over both datasets. AGP achieves the highest accuracy for \( \text{CAR} \) dataset at \( \tau \) being 1. For \( \text{HAI} \) dataset, it achieves the highest accuracy when \( \tau \) gets 10. The second observation is that, when \( \tau \) is zero, the accuracy is near to zero on both datasets. The reason is that no group is treated as abnormal in this case. It corresponds to the lowest size of abnormal \( \gamma_s \) (i.e., \#dag depicted in diagrams). In contrast, when the value of \( \tau \) exceeds the optimal value, the accuracy deteriorates, while the corresponding size of abnormal \( \gamma_s \) increases over both datasets. This is because, more normal groups are detected as abnormal groups. In particular, there is an extreme situation that the accuracy sharply drops to zero, while the corresponding size of abnormal \( \gamma_s \) grows significantly when \( \tau \) is larger than 30 on \( \text{HAI} \) dataset. The reason behind is that, the vast majority of normal groups are wrongly detected as abnormal ones.

Effect on the performance of RSC. Then, we investigate the impact of the threshold on the accuracy of the reliability score based cleaning strategy (i.e., RSC). As shown in Figure 9, an appropriate value of threshold (i.e., \( \tau \) being 1 on \( \text{CAR} \) and \( \tau \) being 10 on \( \text{HAI} \)) contributes to the higher accuracy of RSC. Nonetheless, when the value of \( \tau \) deviates from the optimal value, the accuracy gets worse. The reason is that, the further the \( \tau \) is to the optimal value, the more the groups are processed wrongly by AGP, and thus, the less the pieces of data in groups are correctly repaired by RSC. Besides, the precision of RSC (w.r.t. \( \text{Recall-R} \)) remains higher than the recall of RSC (w.r.t. \( \text{Recall-R} \)). This is because, when more groups are wrongly processed by AGP in previous step, RSC executed within each group is not able to repair more errors caused by AGP, resulting in the lower recall. There is such an extreme case that the recall sharply drops nearly to zero when \( \tau \) is larger than 30 on \( \text{HAI} \) dataset.

Effect on the performance of FSCR. Next, we study the performance of our presented conflict resolution method (i.e., FSCR) when varying the value of threshold. The corresponding results are depicted in Figure 10. As expected, the appropriate value of threshold (i.e., \( \tau \) being 1 on \( \text{CAR} \) and \( \tau \) being 10 on \( \text{HAI} \)) contributes to the optimal accuracy. Moreover, we can find that, the precision maintains high value when the value of \( \tau \) deviates from the optimal value. According to the definition of precision, it means that few detected conflicts are wrongly repaired. Besides, the lower recall than precision indicates that, some errors have not been detected by FSCR. The recall sharply drops, even near to zero for \( \text{HAI} \) when \( \tau \) is larger than 30. It signifies that, there are more and more erroneous values that are not detected by FSCR, since those errors have not been correctly processed by AGP or RSC in previous phases.
7.3.2 Effect of Error Percentage

Effect on the performance of AGP. First, we verify the effect of error percentage on the performance of AGP. The corresponding results are shown in Figure 12 with various error percentages. It is observed that, as the growth of error percentage, the accuracy of AGP decreases. This is because, the higher the error rate, the more the (abnormal) groups, and hence the less the γs within one group. Thus, when there are less and less γs within a group with the increase of error rate, AGP easily tends to treat more and more normal groups wrongly as abnormal ones, for a fixed value of threshold on each dataset. As a result, both the precision and recall of AGP gets lower according to their definitions.

Effect on the performance of RSC. Then, we investigate the accuracy of RSC method when changing the error percentage, and report the corresponding results in Figure 13. We can observe that, both the precision and recall of RSC drop slightly with the increasing error rate. There are two major reasons for the trends. The first one is about the propagated influence of the decreasing accuracy of AGP in the previous step. The second reason comes from the statistical characteristic of RSC, which employs a reliability score based on Markov weight learning. The larger the error rate, the less reliable the learned weights, and the lower the accuracy of RSC. On the other hand, we have to mention that, RSC is quite robust to the change of error rate. In particular, the drop of precision is around 10%, and the drop of recall is around 1%. In addition, the recall is higher than the precision in most cases. The reason is that, with the growth of error rate, the number of repaired γs by RSC increases faster than the number of γs containing errors, which partly results from more wrongly processed groups of AGP (as explained earlier).

Effect on the performance of FSCR. We also study the impact of error percentage on the performance of FSCR. The corresponding results are depicted in Figure 14. One can observe that, the accuracy has no significant fluctuation with the changing error percentage.

The values of both precision and recall are always above 90%, and the fluctuation of them is within 6%. The high accuracy of FSCR reflects that, FSCR is indeed capable of cleaning out those errors which have not been correctly cleaned by AGP or RSC in previous stages. Furthermore, it attributes the higher recall than precision to more wrongly detected conflicts by FSCR, due to the relatively lower accuracy of AGP and RSC in previous stages. In addition, it is worth pointing out that, we have analyzed the overall performance of MLNClean in terms of the experimental results in Figure 6 when changing the error percentage. Thus, we omit the related description here due to the space constraint.

7.3.3 Effect of Distance Metrics

The distance metric plays an important role in MLNClean from two aspects. First, it is the basis of measuring the similarity of two groups, involving AGP strategy. Second, it is an significant factor of computing the reliability score, which is employed by RSC method. Thus, we also evaluate the effect of different distance metrics, including the Levenshtein distance and cosine distance, on the accuracy of MLNClean. As shown in Table 5, the accuracy of MLNClean using the Levenshtein distance is higher than that using cosine distance on both datasets. The reason is that, for the cosine distance, if the foremost few characters of a string are incorrectly spelled, the cosine distance from it to its similar string might be large. Nevertheless, the Levenshtein distance just decides how many different characters between two strings, regardless of the positions of those characters. Thus, the Levenshtein distance is more suitable to deal with various error types.

7.4 Results on Distributed MLNClean

In this section, we evaluate the performance of our proposed distributed MLNClean version using larger HAI and TPC-H datasets. This set of experiments was implemented on Spark 1.0.2, and was executed on a 11-node Dell cluster (1 master with 10 workers), each node has two Intel(R) Xeon(R) E5-2620 v3 2.4GHz processors (12 physical cores, 24 CPU threads) and 64GB RAM.

Figure 15 plots the corresponding results when varying the error percentage. As expected, with the growth of error percentage, the execution time of MLNClean gets longer, and its accuracy gets lower on both HAI and TPC-H datasets. The reason behind is similar to that analyzed in Section 7.2. We would like to point out that, when the error percentage increases from 5% to 30%, the accuracy...
of MLNClean is always above 95% for all cases, and the drop of accuracy is about less than 3% over both datasets. Consequently, MLNClean maintains good robustness on Spark platform.

In addition, we change the number of workers from 2 to 10 on TPC-H dataset, and present the corresponding results in Table 6. One can observe that, the time cost drops as the number of workers grows, while the accuracy has very slight fluctuation. When the number of workers changes from 2 to 10, the efficiency has about 6.7 times speedup.

8 CONCLUSIONS

In this paper, we propose a novel hybrid data cleaning framework MLNClean on top of Markov logic networks (MLNs). It combines the advantages of quantitative methods and qualitative ones, and is capable of cleaning both schema-level and instance-level errors. With the help of an effective two-layer MLN index, MLNClean consists of two major cleaning stages, i.e., first cleaning multiple data versions independently and then deriving the final unified clean data from multi-version data. In the first cleaning stage, an AGP strategy is presented to process abnormal groups (built on the MLN index). Based on a new concept of reliability score, an RSC method is developed to clean data within each group. Moreover, in the second cleaning stage, with a newly defined concept of fusion score, an FSCR algorithm is proposed to eliminate conflicts when unifying multiple data versions. Extensive experimental results on both real and synthetic datasets demonstrate the superiority of MLNClean to the state-of-the-art approach in terms of both accuracy and efficiency.

In the future, we intend to establish more sophisticated strategies to process abnormal groups, since the performance of this step significantly affects the overall performance of MLNClean.

ACKNOWLEDGMENTS

This work was supported in part by the National Key R&D Program of China under Grant No. 2018YFB1004003, the 973 Program under Grant No. 2015CB352502, the NSFC under Grant No. 61522208, the NSFC-Zhejiang Joint Fund under Grant No. U1609217, and the ZJU-Hikvision Joint Project. Both Yunjun Gao and Xiaoye Miao are the corresponding authors of the work.

REFERENCES

[1] W. W. Eckerson, “Data quality and the bottom line: Achieving business success through a commitment to high quality data,” The Data Warehousing Institute, pp. 1–36, 2002.
[2] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang, “Data cleaning: Overview and emerging challenges,” in SIGMOD, pp. 2201–2206, 2016.
[3] E. Rahm and H. H. Do, “Data cleaning: Problems and current approaches,” IEEE Data Eng. Bull., vol. 23, no. 4, pp. 3–13, 2000.
[4] Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker, “Temporal rules discovery for web data cleaning,” PVLDB, vol. 9, no. 4, pp. 336–347, 2015.
[5] L. Bertossi, L. Bravo, E. Franzoni, and A. Lopatenko, “Complexity and approximation of fixing numerical attributes in databases under integrity constraints,” in International Workshop on Database Programming Languages, pp. 262–278, Springer, 2005.
[6] G. Beskales, I. F. Ilyas, and L. Golab, “Sampling the repairs of functional dependency violations under hard constraints,” PVLDB, vol. 3, no. 1, pp. 197–207, 2010.
[7] G. Beskales, I. F. Ilyas, L. Golab, and A. Galilullin, “On the relative trust between inconsistent data and inaccurate constraints,” in ICDE, pp. 541–552, 2013.
[8] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kremenetsidis, “Conditional functional dependencies for data cleaning,” in ICDE, pp. 746–755, 2007.
[9] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi, “A cost-based model and effective heuristic for repairing constraints by value modification,” in SIGMOD, pp. 143–154, 2005.
[10] X. Chu, I. F. Ilyas, and P. Papotti, “Holistic data cleaning: Putting violations into context,” in ICDE, pp. 458–469, 2013.
[11] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, “Improving data quality: Consistency and accuracy,” in PVLDB, pp. 315–326, 2007.
[12] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang, “NADEEP: A commodity data cleaning system,” in SIGMOD, pp. 541–552, 2013.
[13] W. Fan, F. Geerts, X. Jia, and A. Kremenetsidis, “Conditional functional dependencies for capturing data inconsistencies,” ACM Trans. Database Syst., vol. 33, no. 2, pp. 61–64, 2008.
[14] F. Geerts, G. Mecca, P. Papotti, and D. Santoro, “The LLUNATIC data-cleaning framework,” PVLDB, vol. 6, no. 9, pp. 625–636, 2013.
[15] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J. Quiñé-Ruiz, N. Tang, and S. Yin, “BigDancing: A system for big data cleaning,” in SIGMOD, pp. 1215–1230, 2015.
[16] S. Kolahi and L. V. S. Lakshmanan, “On approximating optimum repairs for functional dependency violations,” in ICDT, pp. 53–62, 2009.
[17] A. Lopatenko and L. Bravo, “Efficient approximation algorithms for repairing inconsistent databases,” in ICDE, pp. 216–225, 2007.
[18] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg, “Active-clean: Interactive data cleaning for statistical modeling,” PVLDB, vol. 9, no. 12, pp. 948–959, 2016.
[19] C. Mayfield, J. Neville, and S. Prabhakar, “ERACER: A database approach for statistical inference and data cleaning;” in SIGMOD, pp. 75–86, 2010.
[20] M. Yakout, L. Berti-Equille, and A. K. Elmagarmid, “Don’t be scared: Use scalable automatic repairing with maximal likelihood and bounded changes,” in SIGMOD, pp. 553–564, 2013.
[21] N. Prokoshyina, J. Szlichta, F. Chiang, R. J. Miller, and D. Srivastava, “Combining quantitative and logical data cleaning,” PVLDB, vol. 9, no. 4, pp. 300–311, 2015.
[22] T. Rekatsinas, Y. Chen, I. F. Ilyas, and C. Ré, “Holoclean: Holistic data repairs with probabilistic inference,” PVLDB, vol. 10, no. 11, pp. 1190–1201, 2017.
[23] S. Giannakopoulos, M. Karpathiotakis, B. Gaidioz, and A. Ailamaki, “CleanM: An optimized query language for unified scale-out data cleaning,” PVLDB, vol. 10, no. 11, pp. 1466–1477, 2017.
[24] F. Niu, C. Zhang, C. Ré, and J. W. Shavit, “DeepDive: Web-scale knowledge-base construction using statistical learning and inference,” VLDS, vol. 12, pp. 25–28, 2012.
[25] P. M. Domingos and D. Lowd, Markov Logic: An Interface Layer for Artificial Intelligence. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2009.
[26] F. Niu, C. Ré, A. Doan, and J. W. Shavit, “Tufty: Scaling up statistical inference in markov logic networks using an RDBMS,” PVLDB, vol. 4, no. 6, pp. 373–384, 2011.
[27] W. Fan and F. Geerts, Foundations of Data Quality Management. Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2012.
[28] T. C. Redman, “The impact of poor data quality on the typical enterprise,” Commun. ACM, vol. 41, no. 2, pp. 79–82, 1998.