VIII. Implementation Details of EnuMiner

A. Algorithm Overview

The previous work [18] introduced editing rules and studied how to determine certain fixes with editing rules Σ and master data D_m. However, editing rules discovery problem is not thoroughly studied. One way to obtain editing rules is to ask experts to design for each dataset. To avoid suffering the high cost of experts, we first propose an enumeration-based algorithm named EnuMiner that automatically mines editing rules as presented in Algorithm 5.

In general, EnuMiner discovers different editing rules φ by enumerating possible combinations of attributes (X, X_m) and pattern t_p, and returns a set of non-redundant editing rules, which maximizes the sum of utility.

Specifically, it first initializes a candidate rule set Σ_c (line 2-9). Each possible attribute pair $(A, A_m) \in \{(A, A_m)|A_m \in M(A), A \in R \setminus Y\}$ is considered as $LHS(\varphi)$ and the pattern t_p in φ is empty by default. In this way, we initialize the candidate rule set Σ_c whose size is $\sum_{A \in R \setminus Y} |M(A)|$. Then, for each candidate rule $\varphi_c = ((X, X_m) \rightarrow (Y, Y_m), t_p) \in \Sigma_c$, we try to specialize it to discover more specific rules (lines 11-21). There are two ways for editing rule specialization: (a) Adding a new attribute pair (A, A_m) into (X, X_m) (line 14-17). (b) Adding an attribute and a specific value from its domain (A, v) into t_p (line 18-21), where A should not be already included in $LHS_p(\varphi_c)$. The new discovered editing rules will be included in the candidate set Σ_c and the discovered rule set Σ (line 22-28). When there is no candidate editing rule in Σ_c, the enumeration process terminates and we obtain the discovered rule set Σ (line 10). Finally, we rank all editing rules Σ according to their utility measures and return the a set of non-redundant editing rules Σ_K that maximizes the sum of utility (line 30).

The Rank algorithm is presented in Algorithm 6. Given a set Σ of editing rules, we first sort editing rules φ in Σ from the largest to the smallest according to the utility measure $\mathcal{U}(\varphi)$ (line 1). An empty set Σ_K is initialized to store the top-K editing rules. Then, we obtain the rule φ by iterating the sorted rule set Σ, and add it into Σ_K if there exist no rule $\varphi' \in \Sigma_K$ dominates φ, i.e., $\nexists \varphi' \in \Sigma_K, \varphi' \lessdot \varphi$. Finally, we can obtain a set of editing rules Σ_K consisting of top-K non-redundant rules under utility measure. The general idea of the algorithm is to select the high-utility rule first, and then consider the dominant relationship between the rules. In this way, the discovered rules are expected to be high-utility and representative, which can lead to high-quality data cleaning results and is easy for users to understand the cleaning process.

B. Acceleration

To enumerate all editing rules $\varphi = ((X \cup \{A\}, X_m \cup \{A_m\}) \rightarrow (Y, Y_m), t_p)$, we do not only consider whether an attribute should be included in X or a specific value should be specified as a condition in pattern t_p. A too huge space is unacceptable to enumerate due to the expensive time cost. So we propose several optimizations to accelerate EnuMiner (Algorithm 5).

Algorithm 5 EnuMiner Algorithm

Input: Input Data, D; Master Data: D_m; To-Repair Attribute Pair: (Y, Y_m); Support Threshold: η_s
Output: Discovered Rule Set, Σ;

1: $I_D = HashMap(), I_{D_m} = HashMap()$ // Init Index Mapping
2: $\Sigma = \{\}, \Sigma_c = \{\}, R_x = R \setminus Y$
3: **for** $A \in X$ **do**
4: **for** $A_m \in M(A)$ **do**
5: $\varphi = ((A, A_m) \rightarrow (Y, Y_m), ())$
6: $\Sigma.add(\varphi)$
7: **if** $C(\varphi)$¡1 **then**
8: $\Sigma_c.add(\varphi)$
9: $I_D.put(\varphi, D.index)$ // Update I_{D_m} accordingly.
10: **while** $\Sigma_c \neq \emptyset$ **do**
11: **for** $\varphi_c = ((X, X_m) \rightarrow (Y, Y_m), t_p) \in \Sigma_c$ **do**
12: $\Sigma_n = \{\}$
13: $D' = D[I_D.get(\varphi_c)], D'_m = D_m[I_{D_m}.get(\varphi_c)]$
14: **for** $A \in R_x \setminus X_c$ **do**
15: **for** $A_m \in M(A)$ **do**
16: $\varphi = ((X \cup \{A\}, X_m \cup \{A_m\}) \rightarrow (Y, Y_m), t_p)$
17: $\Sigma_n.add(\varphi)$
18: **if** $A \notin X_p$ **then**
19: **for** $v \in dom(D'[A])$ **do** // subspace searching
20: $\varphi = ((X, X_m) \rightarrow (Y, Y_m), t_p \cup \{(A, v)\})$
21: $\Sigma_n.add(\varphi)$
22: **for** $\varphi \in \Sigma_n$ **do**
23: $I_D.put(\varphi, \text{cover}(\varphi).index)$ // Update I_{D_m} accordingly.
24: // Measure rules in the sub-space
25: **if** $S(\varphi) < \eta_s$ or $C(\varphi)$=1 **then**
26: Σ_n.remove(φ) // pruning
27: **if** $S(\varphi) \geq \eta_s$ **then**
28: $\Sigma.add(\varphi)$
29: $\Sigma_c = \Sigma_n$
30: $\Sigma_K = \text{Rank}(\Sigma, K)$
31: **return** Σ_K

Algorithm 6 Rank

Input: Editing Rule Set: Σ; Rule Number: K;
Output: Discovered Rule Set, Σ_K;

1: $\Sigma = Sort(\Sigma)$ // Sorting according to rule utility $\mathcal{U}(\varphi)$
2: $\Sigma_K = \{\}$
3: **for** $\varphi \in \Sigma$ **do**
4: **if** $\nexists \varphi' \in \Sigma_K, \varphi' \lessdot \varphi$ **then**
5: $\Sigma_K.add(\varphi)$
6: **if** $|\Sigma_K| = K$ **then**
7: $break$
8: **return** Σ_K

1) Pruning: Following CFD [17] and DD discovery [37], we also set a support threshold η_s and only rules such that $S(\varphi) \geq \eta_s$ will be considered as valid rules. According to Lemma 1, we can safely stop adding more attribute pairs to X or value conditions to pattern t_p to the rule whose support is smaller than η_s, and remove it from the candidate set (lines 25-26). In addition, We also stopped refining any rules that return only one candidate fix, i.e., $C(\varphi) = 1$.

2) Subspace Searching: It is not difficult to find that for any editing rule φ_1 and φ_2 refined on φ_1, the space covered by φ_2 must be the subspace of covered by φ_1, i.e., cover$(\varphi_2) \subset$ cover(φ_1) (see proof of Lemma1). Therefore, we can record the covered tuples $D' \subset D$ of each discovered rule φ. When specializing φ, we can only consider values in D' and thus avoid enumerating many useless rules (line 19). Furthermore, we do not have to process a lot of repeated queries to find D' for rule discovery and rule measure (line 13).

IX. Proof

Proof of Lemma 1. Let cover(φ_1) and cover(φ_2) denote the set of input tuples covered by φ_1 and φ_2 respectively, e.g., cover$(\varphi_1) = \{t_i | t_i[X_{p1}] = t_{p1}[X_{p1}], t_i \in D\}$. According to Definition 2 and 3, given $\varphi_1 \preccurlyeq \varphi_2$, we have $t_{p1} \preccurlyeq t_{p2}$, i.e., $X_{p1} \subsetneq X_{p2} \wedge t_{p1}[X_{p1}] = t_{p2}[X_{p1}]$. Thus, we have cover$(\varphi_2) \subset$ cover(φ_1), and $|\text{cover}(\varphi_2)| \leq |\text{cover}(\varphi_1)|$. Recall that $S(\varphi)$ denotes how many input tuples are covered by a given rule φ, $S(\varphi)$ can be calculated via $S(\varphi) = \frac{|\text{cover}(\varphi)|}{|D|}$. Therefore, we have $S(\varphi_1) \geq S(\varphi_2)$ if two eRs satisfy $\varphi_1 \preccurlyeq \varphi_2$. □