A genetic algorithm to discover relaxed functional dependencies from data

Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese

University of Salerno, Department of Computer Science,
via Giovanni Paolo II n.132, 84084 Fisciano (SA), Italy
{lcaruccio,deufemia,gpolese}@unisa.it

Abstract. Approximate functional dependencies are used in many emerging application domains, such as the identification of data inconsistencies or patterns of semantically related data, query rewriting, and so forth. They can approximate the canonical definition of functional dependency (FD) by relaxing on the data comparison (i.e., by considering data similarity rather than equality), on the extent (i.e., by admitting the possibility that the dependency holds on a subset of data), or both. Approximate FDs are difficult to be identified at design time like it happens with FDs. In this paper, we propose a genetic algorithm to discover approximate FDs from data. An empirical evaluation demonstrates the effectiveness of the algorithm.

Keywords: Functional Dependency, Genetic Algorithm, Discovery from data

1 Introduction

In the last decades, functional dependencies have been extended to deal with new problems, such as data cleansing [2], record matching [9], query rewriting upon schema evolutions [6], or to express constraints for complex data types, such as multimedia [7]. To this end, the canonical definition of FD has been extended either by using approximate paradigms rather than equality to compare attribute values, or by admitting subsets of data on which the FD does not hold. In the literature such extensions are referred to as *relaxations*, and these new dependencies are indicated with the term *relaxed functional dependencies* (RFDs) [3]. The relaxation degree is expressed by means of thresholds, indicating either the required similarity degree for the RFD to hold, or the minimum percentage of data on which the RFD must hold.

While FDs were originally specified at database design time, as properties of a schema that should hold on every instance of it, in order to reduce the designer effort and to dynamically adapt the RFDs to the evolution of the application domain, it is necessary to automatically discover them from data. This is made possible also thanks to the availability of big data collections, and to the contributions of several research areas, such as machine learning and knowledge discovery [18].

Most of the discovery algorithms described in the literature are intended for FDs [23], whereas few RFD definitions are equipped with algorithms for discovering them from data [18]. In this paper, we propose a genetic algorithm (GA) to identify a broad class of RFDs, including both relaxation criteria. In general, GAs are efficient for global searches in large search spaces, for which deterministic methods are not suitable. They perform operations inspired to natural species evolutions, such as natural selection, crossover, and mutation. In particular, by using these operations, the proposed algorithm iteratively generates new candidate RFDs, but only a subset of them, determined by means of a fitness function, survives to the evolution process. The fitness function exploits the support and confidence quality measures, mainly used for evaluating association rules. An empirical evaluation demonstrates the effectiveness of the algorithm.

The paper is organized as follows. Section 2 reviews the RFD discovery algorithms existing in the literature. Section 3 provides some background definitions about RFDs and formulates the problem of RFD discovery. Section 4 presents the proposed genetic algorithm named GA-RFD for discovering RFDs from data, whose performance are reported in Section 5. Finally, summary and concluding remarks are included in Section 6.

2 Related Work

In the literature there are two main categories of methods to automatically discover FDs from data, namely top-down and bottom-up methods. The formers exploit an attribute lattice to generate candidate FDs, which are successively tested to verify their validity. The whole process is made efficient by exploiting valid FDs to prune the search space for new candidate FDs. Examples of top-down approaches include the algorithms TANE [14], FD_Mine [27], FUN [21], and DFD [1]. On the other hand, bottom-up methods derive candidate FDs from two attribute subsets, namely *agree-sets* and *difference-sets*, which are built by comparing the values of attributes for all possible combinations of pairs of tuples. Examples of bottom-up approaches include the algorithms DepMiner [19], FastFD [26], and FDep [11]. Recently, an hybrid algorithm has been proposed in order to obtain better performance in all cases [24].

Although a recent survey listed thirdy-four different types of RFDs [3], very few of them are equipped with algorithms for discovering them from data [18]. These are reviewed in the following.

Approximate functional dependencies (AFDs) are FDs holding for 'most' rather than 'all' the tuples of a relation r [16]. The amount of tuples satisfying the AFDs can be calculated by means of several measures [12], among which the g_3 error measure is the most frequently used [16]. The latter represents the minimum fraction of tuples that must be removed from a relation instance r in order to make a candidate AFD valid. Most approaches for AFD discovery use a small portion of tuples (sampling) $s \subset r$ to decide whether an AFD holds on r [16]. The method proposed in [15] exploits the error measure of super keys to determine the approximate satisfaction of AFDs.

Like AFDs, *conditional functional dependencies* (CFDs) are FDs holding for a subset of tuples, but in this case the subset is identified by specifying conditions [2]. Among the approaches for discoverying CFDs from data, the algorithm proposed in [8] exploits an attribute lattice derived from partitions of attribute values, in order to generate candidate CFDs. Alternatively, the greedy algorithm proposed in [13] tries to derive CFDs by finding close-to-optimal tableau, where the closeness is measured by means of support and confidence. In [10] the authors adapt three algorithms used for FD discovery, namely FD_Miner, TANE, and FastFD, to the discovery of CFD.

Matching dependencies (MDs) are RFD_c defined in terms of similarity predicates to accommodate errors, and different representations in unreliable data sources [9]. They have been mainly proposed for object identification. The most recent algorithm for MD discovery evaluates the utility of candidate MDs for a given database instance, determining the corresponding similarity threshold pattern. The utility is measured by means of support and confidence MDs, whereas thresholds are determined by analyzing the statistical distribution of data.

Differential dependencies (DDs) are RFD_c based on differential constraints on attribute values [25]. Among the algorithms proposed for their discovery, the one proposed in [25] exploits reduction algorithms, which detect DDs by first fixing the RHS differential function for each attribute in r, and then finding the set of differential functions that left-reduce the LHS. The performances of the algorithm are improved by means of pruning strategies based on the subsumption order of differential functions, implication of DDs, and instance exclusion. Another approach for DD discovery tries to reduce the problem search space by assuming a user-specified distance threshold as upper limit for the distance intervals of LHS [17]. The algorithm is based on a distance-based subspace clustering model, and exploits pruning strategies to efficiently discover DDs when high threshold values are specified.

As opposed to these discovery algorithms, which focus each on a specific RFD, the discovery algorithm proposed in [5] aims to identify the entire class of RFDs [3] relaxing on the tuple comparison method. In particular, the approach relies on lattice-based algorithms conceived for FD discovery, which are feeded with similar subsets of tuples derived from previously computed differential matrices.

The algorithm proposed in this paper further extends the class of discovered RFDs, by also including those relaxing on the extent.

3 Discovery Relaxed Functional Dependencies from Data

3.1 Relaxed Functional Dependencies

Before defining RFDs we need to introduce the concept of *similarity constraint*, which expresses the similarity of two values on a specific domain, and it is represented through a function ϕ. In particular, given two attributes A and B on a domain D, $\phi(A, B)$ evaluates the similarity of A and B. As an example, ϕ can be defined in terms of a similarity metric $\approx$, like for instance the edit distance, such that $a \approx b$ is true if a and b are "close" enough w.r.t. a predefined threshold.

Given a relational schema R defined over a fixed set of attributes, and $R_1 = (A_1, \ldots, A_k)$ and $R_2 = (B_1, \ldots, B_m)$ two relation schemas of $\mathcal{R}$, an RFD φ on $\mathcal{R}$ is denoted by

$$\mathbb{D}_{c_1} \times \mathbb{D}_{c_2} : (X_1, X_2)_{\Phi_1} \xrightarrow{\Psi \geq \epsilon} (Y_1, Y_2)_{\Phi_2} \tag{1}$$

where

- $\mathbb{D}_{c_1} \times \mathbb{D}_{c_2} = \{(t_1, t_2) \in dom(R_1) \times dom(R_2) \mid \big(\bigwedge_{i=1}^{k} c_{1_i}(t_1[A_i]) \big) \wedge \big(\bigwedge_{j=1}^{m} c_{2_j}(t_2[B_j]) \big) \}$ where $c_1 = (c_{1_1}, \ldots, c_{1_k})$ and $c_2 = (c_{2_1}, \ldots, c_{2_m})$, with c_{1_i} and c_{2_j} predicates on $dom(A_i)$ and $dom(B_j)$, respectively, that filter the tuples on which φ applies;
- $X_1, Y_1 \subseteq attr(R_1)$, and $X_2, Y_2 \subseteq attr(R_2)$, with $X_1 \cap Y_1 = \varnothing$ and $X_2 \cap Y_2 = \varnothing$;
- Φ_1 (Φ_2, resp.) is a set of constraints $\phi[X_1, X_2]$ ($\phi[Y_1, Y_2]$, resp.) on attributes X_1 and X_2 (Y_1 and Y_2, resp.). For any pair of tuples $(t_1, t_2) \in \mathbb{D}_{c_1} \times \mathbb{D}_{c_2}$, the constraint $\phi[X_1, X_2]$ ($\phi[Y_1, Y_2]$, resp.) indicates true, if the similarity of t_1 and t_2 on attributes X_1 and X_2 (Y_1 and Y_2, resp.) agrees with the constraint specified by $\phi[X_1, X_2]$ ($\phi[Y_1, Y_2]$, resp.).
- Ψ is a coverage measure defined on $\mathbb{D}_{c_1} \times \mathbb{D}_{c_2}$, which quantifies the satisfiability degree of φ on r, and can be defined as a function $\Psi : dom(X) \times dom(Y) \rightarrow \mathbb{R}$ measuring the amount of tuples in r violating or satisfying φ.
- ϵ is a threshold indicating the lower bound (or upper bound in case the comparison operator is $\leq$) for the result of the coverage measure.

Given $r_1 \subseteq \mathbb{D}_{c_1}$ and $r_2 \subseteq \mathbb{D}_{c_2}$ two relation instances on (R_1, R_2), (r_1, r_2) *satisfies* the RFD φ, denoted by $(r_1, r_2) \vDash \varphi$, if and only if: $\forall\ (t_1, t_2) \in (r_1, r_2)$, if $\phi[X_1, X_2]$ indicates true for each constraint $\phi \in \Phi_1$, then *almost always* $\phi[Y_1, Y_2]$ indicates true for each constraint $\phi \in \Phi_2$. Here, *almost always* means that $\Psi(\pi_{X_1}(r_1)\pi_{X_2}(r_2), \pi_{Y_1}(r_1)\pi_{Y_2}(r_2)) \geq \epsilon$.

For RFDs defined on single relation schemas (i.e., $R_1 = R_2$), if $X_1 = X_2$ and $Y_1 = Y_2$, then we will simplify the RFD notation given in (1) by using $\mathbb{D}_c$, X, and Y to refer to the Cartesian product $\mathbb{D}_{c_1} \times \mathbb{D}_{c_2}$, and to the pairs (X, X) and (Y, Y), respectively. Moreover, if the RFD has to be satisfied by all tuples in r, then the symbol $\Psi_{err(0)}$ is shown on the arrow. Such coverage measure corresponds to the expression $\psi(X, Y) = 0$, where $\psi(X, Y)$ measures the number of tuples violating the RFD. As an example, the canonical FD can also be written as: $\mathbb{D}_{\text{TRUE}} : X_{\text{EQ}} \xrightarrow{\Psi_{err(0)}} Y_{\text{EQ}}$, where TRUE is a sequence of tautologies, hence $\mathbb{D}_{\text{TRUE}} = dom(R_1)$, whereas EQ is the equality constraint.

As an example, in a database of body measures it is likely to have a similar shoe size for people having a similar height. Thus, the following RFD might hold $\mathbb{D}_{\text{TRUE}} : \mathsf{Height}_{\approx} \xrightarrow{\Psi_{err(0)}} \mathsf{Shoe\ Size}_{\approx}$, where $\approx$ is a differential function. On the other hand, there might be exceptions to this dependencies, since few people might have close heights but more different shoe sizes. This can be modeled by introducing a different coverage measure into the RFD, making it approximate $\mathbb{D}_{\text{TRUE}} : \mathsf{Height}_{\approx} \xrightarrow{\psi(Height, ShoeSize) \leq 0.02} \mathsf{Shoe\ Size}_{\text{EQ}}$, where $\psi(Height, ShoeSize) \leq 0.02$ indicates that at most for 2% of tuples the RFD does not hold.

3.2 The RFD discovery problem

Given a relation r, the discovery of RFDs is the problem of finding a minimal cover set of valid RFDs that hold in r. This problem introduces further complexity to the already complex task of FD discovery from data, due to the significantly larger search space, which derives from the several similarity/distance functions that may be defined over each attribute [17]. Consequently, it is necessary to devise tractable algorithms capable of extracting RFDs with meaningful similarity constraints.

In this paper, we analyze the problem of RFD discovery from data given a set of thresholds for both attribute comparison and coverage measure. Although this reduces the overall complexity of the problem, it still remains more complex than the FDs discovery one. In fact, as said in Section 3.1, an RFD $X \rightarrow Y$ holds on a relation r iff whenever the distance between two tuples t_1 and t_2 in r is below a threshold value α_A on each attribute A belonging to X, then their distance is below a threshold value α_B on each attribute B belonging to Y, with a degree of certainty greater than a threshold value ϵ. For this reason, we have to analyze the similarity between portions of tuple pairs rather than their equality. This does not permit to exploit the equivalence classes of attribute values, usually used for FD discovery. Indeed, an attribute value might be similar to other attribute values that are not similar to each other.

4 Identifying Relaxed Functional Dependencies with a Genetic Algorithm

The proposed algorithm discovers RFDs by comparing tuples pairwise, based on the similarity of subsets of their attributes. In particular, pairs of tuples with a similarity higher than a threshold are represented with the value 1, and 0 otherwise. In this way, we reduce the RFD discovery problem to a search one, which is solved through a GA. The latter implements a simulation in which a population of candidate solutions to an optimization problem evolves towards better solutions. The process usually starts from a population of randomly generated individuals (Initialization), which evolves by stochastically selecting multiple individuals from the current population (based on a fitness function), and modified (recombined and possibly randomly mutated) to form a new population. The algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. In the first case, a satisfactory solution may or may not have been reached.

4.1 GA-RFD

In this section, we explain the GA to discover RFDs, which is named GA-RFD. In particular, in the following we first explain how the discovery problem has been encoded for GA processing, then we define the fitness function to evaluate the satisfactory degree of individuals, and finally, we present the GA-RFD algorithm.

Encoding. Given the thresholds for attribute comparison and coverage measure, the problem of RFD discovery reduces to find all possible dependencies $X \rightarrow Y$ satisfying the following rule: tuples that are similar on the LHS must correspond to those that are similar on the RHS. Such a rule must hold for almost all set of tuples that are similar on the LHS attributes, according to the threshold specified for the coverage measure.

In order to accomplish this with a GA, the dataset has to be modeled in a way that highlights similarity between tuple pairs. For this reason, in the proposed methodology we transform an input dataset in terms of a *difference dataset*. The latter contains the input dataset attributes as columns, the input dataset tuple pairs as rows, and the similarity between tuple pairs as values. In particular, given an attribute X_i and a tuple pair (t_1, t_2), the difference dataset will have the value 1 if $t_1[X_i]$ is similar to $t_2[X_i]$ according to the threshold associated to X_i, and the value 0 otherwise.

Example 1. Let us consider the sample dataset shown in Table 1a, and the similarity thresholds: 1 for the attributes Height and Shoe Size, and 10 for the attribute Weight. By using the absolute difference as a distance function, we can derive the difference dataset shown in Table 1b.

Table 1 A dataset of body measures.

Tuple ID	Height	Weight	Shoe size
1	175	70	40
2	175	75	39
3	175	69	40
4	176	71	40
5	178	81	41
6	169	73	37
7	170	62	39

(a) A sample dataset.

Pair ID	Height	Weight	Shoe Size
(1,2)	1	1	1
(1,3)	1	1	1
(1,4)	1	1	1
(1,5)	0	0	1
(1,6)	0	1	0
(1,7)	0	1	1
(2,3)	1	1	1
(2,4)	1	1	1
(2,5)	0	1	0
(2,6)	0	1	0
(2,7)	0	0	1
(3,4)	1	1	1
(3,5)	0	0	1
(3,6)	0	1	0
(3,7)	0	1	1
(4,5)	0	1	1
(4,6)	0	1	0
(4,7)	0	1	1
(5,6)	0	1	0
(5,7)	0	0	0
(6,7)	1	0	0

(b) The difference dataset.

In order to construct individuals in the population we use an array of integers representing the attribute indices in the difference dataset. In particular, each item i of the array corresponds to a specific attribute X_i in the dataset. In particular, since all the RFDs can be reduced w.l.o.g. to a format with a single attribute on the RHS, each individual of size k will represent a candidate RFD, where the $k-1$ attributes on the LHS correspond to the first $k-1$ items in the array, and the last one corresponds to the attribute on the RHS. It is worth to

notice that the algorithm can be iterated with several values for k in order to consider RFDs with different sizes on the LHS.

Example 2. The individual $[3, 1]$ for the dataset in Table 1a will represent the candidate RFD: Shoe Size $\rightarrow$ Height.

Fitness Function. In order to discover RFDs we exploit the concepts of *support* and *confidence* commonly used in the context of association rule mining. In particular, several studies have demonstrated the relationship existing between FDs and association rules [14]. Formally, let X and Y be two sets of items, the *support* of a set X, denoted with $sup(X)$, represents the ratio of the transactions in the dataset containing X, whereas the confidence represents the ratio $sup(X \cup Y)/sup(X)$.

In our context, $sup(X)$ represents the ratio of tuple pairs that are similar on all attributes in X. In this way, we can use the *confidence* as the fitness function to determine if a candidate RFD $X \rightarrow Y$ is valid: $conf(X \rightarrow Y) = \frac{supp(X \cup Y)}{supp(X)}$. This fitness function returns 1 if and only if whenever tuple pairs are similar on attributes in X, then they are similar also on Y. In order to validate a candidate RFD the proposed algorithm associates the threshold specified for the coverage measure to the fitness function.

Example 3. If we consider the individual $[3, 1]$ of Example 2, representing the candidate RFD: Shoe Size $\rightarrow$ Height, the associated support and confidence are: $sup(\text{Shoe Size}) = \frac{13}{21}$, $sup(\{\text{Shoe Size}, \text{Height}\}) = \frac{6}{21}$, $conf(\text{Shoe Size} \rightarrow \text{Height}) = \frac{6}{21} \times \frac{21}{13} = \frac{6}{13} = 0.46$. This candidate RFD will not be considered as valid, unless the coverage threshold specified in input is less than 0.46.

Algorithm 1 The general GA-RFD algorithm

```
1: procedure GA-RFD(difference dataset D, extent threshold β, int maxgen, int
   pop_size, int k, double p1, double p2)
2:     Set pop, final_pop, new_pop
3:     int gen ← 0
4:     final_pop ← INITIALIZE(pop_size, k, numCols(D))
5:     while gen ≤ maxgen do
6:         if (allOf(final_pop, β)) then
7:             break
8:         end if
9:         new_pop ← SELECT(final_pop, β)
10:        pop ← ∅
11:        pop ← CROSSOVER(new_pop, k, p1)
12:        pop ← MUTATE(pop, k, numCols(D), p2)
13:        final_pop ← new_pop ∪ pop
14:        g ← g + 1
15:    end while
16:    return final_pop
17: end procedure
```

Algorithm. The pseudo-code of the main procedure of GA-RFD is shown in Algorithm 1. It first produces the initial population through the INITIALIZE procedure, and then, within a cycle, it selects only the individuals that reach the fitness function objective (SELECT procedure), makes crossover on selected individuals according to a fixed probability p_1 (CROSSOVER procedure), mutates some of them according to a fixed probability p_2 (MUTATE procedure). The new

population is evaluated in the next iteration of the cycle. GA-RFD terminates if and only if either the population is composed of individuals, each one representing a valid RFD according to the β-threshold for the coverage measure, which determines the fitness objective; or the number of iterations exceeds the specified maximum number.

The INITIALIZE procedure shown in Algorithm 2 creates a set of individuals representing the initial population, according to the size *pop_size* specified as input. The used random values are ranged in the number of columns of the dataset, since each individual element will represent a column index.

The SELECT procedure shown in Algorithm 3 analyzes the individuals of the population *pop* and constructs a new population (*selected_pop*) by including individuals with a probability related to the fitness function objective β.

Algorithm 2 The INITIALIZE procedure

```
1: procedure INITIALIZE(int pop_size, int k, int numCols)
2:     Set pop
3:     for i ≤ pop_size do
4:         List x
5:         for j ≤ k do
6:             add randInt(1,numCols) to x
7:         end for
8:         add x to pop
9:     end for
10:    return pop
11: end procedure
```

Algorithm 3 The SELECT procedure

```
1: procedure SELECT(Set pop, double β)
2:     Set selected_pop → {}
3:     double fitness, confidence
4:     for each c ∈ pop do
5:         confidence = supp(c.{X∪Y}) / supp(c.X)
6:         if confidence ≥ β then
7:             fitness = 1
8:         else
9:             fitness = confidence/β
10:        end if
11:        if rand() < fitness then
12:            selected_pop = selected_pop ∪ {c}
13:        end if
14:    end for
15:    return selected_pop
16: end procedure
```

The CROSSOVER procedure shown in Algorithm 4 creates a new population of individuals (*new_pop*) by crossing individual pairs. We use a one-point crossover to recombine individuals; the *crosspoint* is selected randomly. In this procedure we use the function *swap*, which changes a sublist of two individuals by swapping one to each other.

The MUTATE procedure shown in Algorithm 5 creates a new population of individuals (*new_pop*) by mutating one value in each individual. The individual element that has to be changed (*gene*) and the new value (*new_gene*) are selected randomly. In this procedure we use the function *swapValue*, which changes a

value in a individual by swapping it with the new value. The MUTATE procedure permits to detect candidate RFDs that do not directly depend on the initial population. It has associated a low probability, which limits the randomness of the whole methodology.

Algorithm 4 The CROSSOVER procedure

```
1: procedure CROSSOVER(Set pop, int k, double p)
2:     Set new_pop → {}
3:     int crosspoint
4:     for each c1,c2 ∈ pop with c1 ≠ c2 do
5:         if rand() < p then
6:             crosspoint = randInt(1,k)
7:             for i ≤ crosspoint do
8:                 swap(c1,c2, i)
9:             end for
10:            add c1 to new_pop
11:            add c2 to new_pop
12:        end if
13:    end for
14:    return new_pop
15: end procedure
```

Algorithm 5 The MUTATE procedure

```
1: procedure CROSSOVER(Set pop, int k, int numCols, double p)
2:     Set new_pop
3:     int gene, new_gene
4:     for each c ∈ pop do
5:         if rand() < p then
6:             gene = randInt(1,k)
7:             new_gene = randInt(1,numCols)
8:             swapValue(c, gene, new_gene)
9:         end if
10:        add c to new_pop
11:    end for
12:    return new_pop
13: end procedure
```

5 Empirical Evaluation

In this section we describe the performed empirical evaluation to verify the effectiveness of the proposed GA-RFD algorithm. In particular, we first evaluate the performance of the algorithm when used to discover FDs, then we evaluated the RFD discovery performance.

The evaluation has been performed on a version of GA-RFD implemented in the Python language, and on a machine with an AMD Opteron (TM) CPU, 8 GB RAM, and Linux Ubuntu 16.10 OS. Moreover, we used two real-world datasets, also employed for evaluating several FD discovery algorithms [22], and the sample dataset introduced in Example 1. The probability parameters for the crossover and mutate procedures have been set to 0.85 and 0.3, respectively.

Evaluation by discovering FDs. This experiment compares the performances of GA-RFD with those of the FD discovery algorithms with the aim of evaluating the effectiveness of GA-RFD with respect to well-known results, as in the case of

iris and *bridges* datasets [23]. Here, we do not evaluate the time performances, since FD and RFD discovery problems are not comparable. In order to reduce the problem to discover canonical FDs we assigned the value 0 to all tuple comparison thresholds and the value 1 to the coverage threshold. Table 2 reports the FDs discovered on the analyzed datasets. For the *iris* and *bridge* datasets GA-RFD achieves the same results obtained by other FD discovery algorithms [23].

Table 2 Results obtained by GA-RFD algorithm to discovery FDs.

Dataset	#Columns	#Rows	Size(KB)	#FDs
sample-dataset	3	7	1	2
iris	5	150	5	4
bridges	13	108	6	142

Evaluation by discovering RFDs. In order to verify the effectiveness of GA-RFD, we dirtied some values in the datasets used in the previous experiment with the aim of verifying whether the number of canonical FDs also changes. In other words, we simulated the presence of dirtiness in data, and analyzed GA-RFD overtakes errors in dependency discovery. For this reason, we evaluated how the number of RFDs changes by varying either the tuple comparison thresholds or the coverage measure threshold. It is worth to notice that, we have used this methodology in order to highlight the usefulness of the proposed algorithm w.r.t. the results shown in Table 2. In fact, in real contexts we use GA-RFD with different thresholds, those indicating similar tuples, and the one specifying the satisfiability degree.

Table 3 reports for each dataset: (*i*) the number of errors introduced in the dataset, (*ii*) the number FDs of discovered on the dataset with dirty data, (*iii*) the number of discovered RFDs by associating a threshold ≤1 to one attribute in order to manage similarities in the value comparison[1], (*iv*) the number of RFDs discovered with a coverage threshold of 0.9.

Table 3 Results obtained by using a threshold 1 on the comparison of attribute values, and the results obtained using a coverage threshold of 0.9.

Datasets	n. errors	FDs	RFDs by relaxing on tuple comparison	RFDs by relaxing on extent
sample-dataset2	1	1	2	1
iris2	2	3	4	3
bridges2	2	144	143	254

We can observe that the errors introduced in *sample-dataset2* and *iris2* have reduced the number of discovered FDs. On the contrary, for *bridge2* such a number increases, since a minimal FD ϕ_1 has been violated by errors and three new FDs, whose LHS has more attributes than LHS(ϕ_1), have been discovered. The relaxation on attribute comparison has allowed to capture the errors introduced in the datasets. However, for *bridge2* a new dependency has been discovered

[1] In particular, we have associated the threshold 1 to the first attribute in the *sample-dataset2*, and to the fifth attribute in both *iris2* and *bridges2*. The other attributes have associated threshold 0.

beyond those reported in 2. Regarding the relaxation of tuple coverage, for *sample-dataset2* the threshold is too narrow to capture the introduced error; for *iris2* we obtained less but more general dependencies than those reported in 2; for *bridge2* many more dependencies were discovered, due to the fact that the introduced errors is not significant compared to the size of the dataset, and to the number of dependencies holding in it, as reported in 2.

6 Conclusion and Future Work

In this paper we have proposed GA-RFD, a GA for discovering RFDs from data. It analyzes tuple pairs similarity through a *difference dataset*, and validates RFDs by calculating the confidence on each candidate RFD. A preliminary evaluation of the algorithm has been carried out to assess the effectiveness of the approach.

In the future, we would like to further improve this approach in order to automatically discover the RFDs and the threshold ranges of validity, without requesting their specification to the user. Furthermore, we would like to investigate the discovery of GA-RFD in the context of user interaction logs, especially in web applications, aiming to mine user intent [4]. To this end, mashup repositories are a further interesting application domain, since they are precious sources of data concerning mashup component usage, which can be useful to opportunely advise the development of new mashups [20].

References

1. Abedjan, Z., Schulze, P., Naumann, F.: DFD: Efficient functional dependency discovery. In: Proceedings of the 23rd ACM International Conference on Information and Knowledge Management. pp. 949–958. CIKM '14 (2014)
2. Bohannon, P., Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for data cleaning. In: Proceedings of the 25th International Conference on Data Engineering. pp. 746–755. ICDE '07 (2007)
3. Caruccio, L., Deufemia, V., Polese, G.: Relaxed functional dependencies – A survey of approaches. IEEE TKDE 28(1), 147–165 (2016)
4. Caruccio, L., Deufemia, V., Polese, G.: Understanding user intent on the web through interaction mining. Journal of Visual Languages & Computing 31, Part B, 230 – 236 (2015)
5. Caruccio, L., Deufemia, V., Polese, G.: On the discovery of relaxed functional dependencies. In: Proceedings of the 20th International Database Engineering & Applications Symposium. pp. 53–61. IDEAS '16 (2016)
6. Caruccio, L., Polese, G., Tortora, G.: Synchronization of queries and views upon schema evolutions: A survey. ACM Transactions on Database Systems (TODS) 41(2), 9 (2016)
7. Chang, S.K., Deufemia, V., Polese, G., Vacca, M.: A normalization framework for multimedia databases. IEEE TKDE 19(12), 1666–1679 (2007)
8. Chiang, F., Miller, R.J.: Discovering data quality rules. Proceedings of the VLDB Endowment 1(1), 1166–1177 (2008)
9. Fan, W., Gao, H., Jia, X., Li, J., Ma, S.: Dynamic constraints for record matching. The VLDB Journal 20, 495–520 (2011)

10. Fan, W., Geerts, F., Lakshmanan, L.V.S., Xiong, M.: Discovering conditional functional dependencies. In: Proceedings of the 25th International Conference on Data Engineering, ICDE'09. pp. 1231–1234 (2009)
11. Flach, P.A., Savnik, I.: Database dependency discovery: A machine learning approach. AI Commun. 12(3), 139–160 (1999)
12. Giannella, C., Robertson, E.: On approximation measures for functional dependencies. Inform. Syst. 29(6), 483–507 (2004)
13. Golab, L., Karloff, H., Korn, F., Srivastava, D., Yu, B.: On generating near-optimal tableaux for conditional functional dependencies. Proceedings of the VLDB Endowment 1(1), 376–390 (2008)
14. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: TANE: An efficient algorithm for discovering functional and approximate dependencies. The Computer Journal 42(2), 100–111 (1999)
15. King, R., Oil, J.: Discovery of functional and approximate functional dependencies in relational databases. J. Applied Math. and Decision Sciences 7(1), 49–59 (2003)
16. Kivinen, J., Mannila, H.: Approximate inference of functional dependencies from relations. Theor. Comput. Sci. 149(1), 129–149 (1995)
17. Kwashie, S., Liu, J., Li, J., Ye, F.: Mining differential dependencies: A subspace clustering approach. In: Wang, H., Sharaf, M.A. (eds.) Proceedings of Australasian Database Conference. pp. 50–61. ADC '14 (2014)
18. Liu, J., Li, J., Liu, C., Chen, Y.: Discover dependencies from data - A review. IEEE Transactions on Knowledge and Data Engineering 24(2), 251–264 (2012)
19. Lopes, S., Petit, J.M., Lakhal, L.: Efficient discovery of functional dependencies and armstrong relations. In: Proceedings of the 7th International Conference on Extending Database Technology. pp. 350–364. EDBT '00 (2000)
20. Mario Andrés Paredes-Valverde, Giner Alor-Hernández, A.R.G.R.V.G.E.J.D.: A systematic review of tools, languages, and methodologies for mashup development. Software Practice & Experience 45(13), 365397 (2015)
21. Novelli, N., Cicchetti, R.: Fun: An efficient algorithm for mining functional and embedded dependencies. In: Proceedings of 8th International Conference Database Theory, pp. 189–203. ICDT '01 (2001)
22. Papenbrock, T., Bergmann, T., Finke, M., Zwiener, J., Naumann, F.: Data profiling with Metanome. Proceedings of the VLDB Endowment 8(12), 1860–1863 (2015)
23. Papenbrock, T., Ehrlich, J., Marten, J., Neubert, T., Rudolph, J.P., Schönberg, M., Zwiener, J., Naumann, F.: Functional dependency discovery: An experimental evaluation of seven algorithms. Proceedings of the VLDB Endowment 8(10), 1082–1093 (2015)
24. Papenbrock, T., Naumann, F.: A hybrid approach to functional dependency discovery. In: Proceedings of the 2016 International Conference on Management of Data. pp. 821–833. ACM (2016)
25. Song, S., Chen, L.: Differential dependencies: Reasoning and discovery. ACM Transactions on Database Systems 36, 16 (2011)
26. Wyss, C., Giannella, C., Robertson, E.: FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In: Procs of Intl Conf. on Data Warehousing and Knowl. Disc. pp. 101–110. DaWaK '01 (2001)
27. Yao, H., Hamilton, H.J., Butz, C.J.: FD_Mine: Discovering functional dependencies in a database using equivalences. In: Proceedings of IEEE International Conference on Data Mining. pp. 729–732. ICDM '02 (2002)