Discovering *non-constant* Conditional Functional Dependencies with Built-in Predicates

Antonella Zanzi and Alberto Trombetta

Dipartimento di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria, via Mazzini 5, 21100 Varese, Italy
{antonella.zanzi,alberto.trombetta}@uninsubria.it

Abstract. In the context of the data quality research area, Conditional Functional Dependencies with built-in predicates ($\mathrm{CFD}^{\mathrm{p}}$s) have been recently defined as extensions of Conditional Functional Dependencies with the addition, in the patterns of their data values, of the comparison operators. $\mathrm{CFD}^{\mathrm{p}}$s can be used to impose constraints on data; they can also represent relationships among data, and therefore they can be mined from datasets. In the present work, after having introduced the distinction between *constant* and *non-constant* $\mathrm{CFD}^{\mathrm{p}}$s, we describe an algorithm to discover *non-constant* $\mathrm{CFD}^{\mathrm{p}}$s from datasets.

Keywords: Functional Dependencies, Data Constraints, Data Quality, Data Mining.

1 Introduction

Conditional Functional Dependencies with built-in predicates ($\mathrm{CFD}^{\mathrm{p}}$s) have been defined in [3] as extensions of Conditional Functional Dependencies (CFDs) [8] (which have been proposed in the data quality field as extensions of Functional Dependencies – FDs).

FDs and their extensions, capturing data inconsistency, can be used to evaluate the quality of a dataset and – to a certain extent – also for data cleaning purposes. For example, the use of FDs for data cleaning purposes in relational databases is described in [16], where data dirtiness is equaled to the violation of FDs, and in [5] CFDs have been proposed as a method for inconsistency detection and repairing.

This approach is used, for example, in *Semandaq* [7], a tool using CFDs for data cleaning purposes. Another tool, called *Data Auditor*, is presented in [10] and supports more types of constraints (i.e., CFDs, *conditional inclusion dependencies*, and *conditional sequential dependencies*) used to test data inconsistency and completeness.

In a previous work [19] – along with other types of constraints and dependencies, such as FDs, CFDs, order dependencies and existence constraints – we used $\mathrm{CFD}^{\mathrm{p}}$s in the context of data quality evaluation. In particular, we developed a tool to check a dataset against a set of data quality rules expressed with the XML markup language.

H. Decker et al. (Eds.): DEXA 2014, Part I, LNCS 8644, pp. 35–49, 2014.

CFD^{p}s can potentially express additional constraints and quality rules that cannot be expressed by FDs and CFDs and thus be useful in the data quality field. However, their identification is not often straightforward just looking at the data. For this reason a tool supporting the discovery of CFD^{p}s can be useful to identify rules to be used in the evaluation of the quality of a dataset.

In the present work, after having distinguished between *constant* and *non-constant* CFD^{p}s, we describe an algorithm for discovering *non-constant* CFD^{p}s.

2 CFD^p Definition

CFDs specify constant patterns in terms of equality, while CFD^{p}s are CFDs with built-in predicates ($\neq$, $<$, $>$, $\leq$, $\geq$) in the patterns of their data values. It is assumed that the domain of an attribute is totally ordered if $<$, $>$, $\leq$ or $\geq$ is defined on it.

Syntax. Given a relation schema R and a relation instance r over R, a CFD^p φ on R is a pair $R(X \rightarrow Y, T_p)$, where: (1) $X, Y \subseteq R$; (2) $X \rightarrow Y$ is a standard FD, referred to as the FD embedded in φ; (3) T_p is a tableau with attributes in X and Y, referred to as the pattern tableau of φ, where, for each A in $X \cup Y$ and each tuple $t_{p_i} \in T_p$, $t_{p_i}[A]$ is either an unnamed variable '_' that draws values from $dom(A)$ or 'op a', where op is one of $=, \neq, <, >, \leq, \geq$, and '$a$' is a constant in $dom(A)$. □

Semantics. Considering the CFD^p φ:$R(X \rightarrow Y, T_p)$, where $T_p = t_{p_1}, \ldots, t_{p_k}$, a data tuple t of R is said to match $LHS(\varphi)$, denoted by $t[X] \asymp T_p[X]$, if for each tuple t_{p_i} in T_p and each attribute A in X, either (a) $t_{p_i}[A]$ is the wildcard '_' (which matches any value in dom(A)), or (b) $t[A]$ op a if $t_{p_i}[A]$ is 'op a', where the operator op ($=, \neq, <, >, \leq$ or $\geq$) is interpreted by its standard semantics.

Each pattern tuple t_{p_i} specifies a condition via $t_{p_i}[X]$, and $t[X] \asymp T_p[X]$ if $t[X]$ satisfies the conjunction of all these conditions. Similarly, the notion that t matches $RHS(\varphi)$ is defined, denoted by $t[Y] \asymp T_p[Y]$. An instance I of R satisfies the CFD^p φ, if for each pair of tuples t_1, t_2 in the instance I, if $t_1[X]$ and $t_2[X]$ are equal and in addition they both match the pattern tableau $T_p[X]$, then $t_1[Y]$ and $t_2[Y]$ must also be equal to each other and must match the pattern tableau $T_p[Y]$. □

2.1 *Constant* and *non-constant* CFD^{p}s

Extending the definition introduced for CFDs in [9], we distinguish between *constant* and *variable* – or *non-constant* – CFD^{p}s, calling:

- *constant*, the CFD^{p}s having in their pattern tableaux only operators and constant values (that is, without any unnamed variable '_');
- *non-constant*, the CFD^{p}s having, for the attributes in its right-hand side, an unnamed variable '_' in each pattern tuple of its pattern tableau.

Examples of *constant* (φ_1 and φ_2) and *non-constant* (φ_3, φ_4 and φ_5) CFD^{p}s for the *Iris* dataset[1] are shown in table 1: φ_1 indicates that when the length

[1] From the UCI Machine Learning Repository (http://archive.ics.uci.edu/ml).

Table 1. Examples of *constant* - and *non-constant* CFD^{p}s for the *Iris* dataset

φ_1: iris(petalLength $\rightarrow$ class, T_1)

T_1:

petalLength	class
< 2	*Iris setosa*

φ_2: iris(petalWidth, petalLength $\rightarrow$ class, T_2)

T_2:

petalWidth	petalLength	class
> 1.7	> 4.8	*Iris virginica*

φ_3: iris(sepalLength, petalWidth $\rightarrow$ class, T_3)

T_3:

sepalLength	petalWidth	class
< 5.9	–	–

φ_4: iris(sepalLength, petalLength $\rightarrow$ class, T_4)

T_4:

sepalLength	petalLength	class
$\neq 6.3$	$\neq 4.9$	–

φ_5: iris(petalLength, sepalWidth $\rightarrow$ class, T_5)

T_5:

petalLength	sepalWidth	class
$\neq 4.8$	–	–
$\neq 5.1$	–	–

of the petal is less than 2 cm then the class of the flower corresponds to *Iris setosa*; φ_2 expresses that when the width of the petal is greater than 1.7 cm and – at the same time – the length of the petal is greater than 4.8 cm then the class of the flower corresponds to *Iris virginica*; φ_3 expresses that the FD $sepalLength, petalWidth \rightarrow class$ holds on the subset of the relation tuples having the length of the sepal less than 5.9 cm; φ_4 expresses that the FD $sepalLength, petalLength \rightarrow class$ holds if the length of sepal is different from 6.3 cm and the length of the petal is different from 4.9 cm; finally, φ_5 expresses that the FD $petalLength, sepalWidth \rightarrow class$ holds if the length of the petal is different from 4.8 cm and from 5.1 cm.

3 Discovering CFD^{p}s

CFD^{p}s can be used to add information on data as exemplified in [3], in which case, the dependencies cannot be detected from the analysis of the dataset. However, the CFD^{p}s characterizing a dataset can be discovered analyzing the tuples contained in it.

We propose an algorithm for discovering from a dataset a subset of the existing CFD^{p}s satisfiyng the requirements to be *non-constant*, to have in their right-hand side only one attribute[2] and to have, in their pattern tableaux, conditions with operators only for numerical attributes.

[2] Without loss of generality because of the Armstrong decomposition rule: if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$.

More formally, the algorithm looks for CFD^{p}s that can be written as R($LHS \rightarrow RHS$, T_p), where:

- $LHS \rightarrow RHS$ is the FD embedded in the CFD^p;
- RHS contains a single attribute $A \in R$;
- $LHS \cap A = \emptyset$;
- $X, T \subset R$, $T \neq \emptyset$, $LHS = X \cup T$ and $X \cap T = \emptyset$;
- $\forall\ B \in T\ dom(B)$ is numeric;
- T_p is a pattern tableau with attributes in LHS and RHS;
- $t_p[A]$='_';
- $\forall\ Z \in X$ and $\forall$ tuple $t_{p_i} \in T_p$, $t_{p_i}[Z]$ is an unnamed variable '_' that draws values from $dom(Z)$;
- $\forall\ B \in T$ and $\forall$ tuple $t_{p_i} \in T_p$, $t_{p_i}[B]$ is '*op b*', where '*b*' is a constant in $dom(B)$ and *op* is one of the following operators: $<$, $>$, $\leq$, $\geq$, $\neq$, $=$.

In the following, we will refer to the attributes in X as *variable attributes*, to the attributes in T (for which conditions are searched) as *target attributes*, and to the conditions in the pattern tableau T_p as *target conditions*.

The algorithm is based on the selection of the tuples that do not satisfy a target dependency and on the use of the values of these tuples to build the conditions to obtain valid dependencies.

The algorithm accepts the following input parameters:

- *maxSizeLHS* – setting the maximum number of attributes that the dependencies have to contain in their LHS;
- *sizeT* – setting the size of the set T containing the *target attributes*;
- *maxNumConditions* – an optional parameter setting the maximum number of conditions that can be present in a dependency (i.e., the number of rows in the dependency pattern tableau);
- *depSupport* – an optional parameter indicating, in percentage respect to the dataset tuples, the support required for the resulting dependency (i.e., the minimum number of tuples satisfying the dependency).

The first step performed by the algorithm is the generation of candidates for the target dependencies, in the form $LHS \rightarrow A$ with the attributes in LHS divided in the *variable attributes* set X and in the *target attributes* set T.

To generate the candidates, we have adopted the small-to-large search approach, which has been successfully used in algorithms to discover traditional FDs and in many data mining applications, starting to compute dependencies with a number of attributes equal to the size of the set T in their left-hand side and then proceeding adding *variable attributes* in the set X.

In order to reduce the time spent by the algorithm producing the candidates, some pruning approaches have been introduced. A relevant reduction in the number of the generated candidates applies when a FD $Y \rightarrow A$, with $Y \subset R$ and $A \in R$, holds on the dataset. In this case, it is not necessary to build any candidates of the form $Z \rightarrow A$, with $Z \subset R$ and $Y \subseteq Z$.

The number of generated candidates is reduced also: (1) in the presence of attributes having the same value for all the tuples in the dataset – such attributes

```
Data: An instance relation r over the schema R
Input parameters: maxSizeLHS, sizeT, maxNumConditions, depSupport
Result: CFDps
resultSet = ∅;
RHS = {{A}|∀A ∈ R};
numSupportTuples = computeSupportTuples(depSupport);
for Y ∈ RHS do
    LHSattr_init = {{B}|∀B ∈ (R − Y)};
    LHSattr_1 = pruneSet(LHSattr_init, Y);
    l = 1;
    while l ≤ (maxSizeLHS) do
        candidateSet = generateCandidates(LHSattr_l, Y);
        for candidate ∈ candidateSet do
            patternTableauSet = findTargetConditions(candidate,
            numSupportTuples);
            if patternTableauSet ≠ ∅ then
                for patternTableau ∈ patternTableauSet do
                    if acceptResults(candidate, patternTableau,
                    maxNumConditions, numSupportTuples) then
                        resultSet += buildCFDp(candidate, patternTableau);
                    end
                end
            end
        end
        LHSattr_{l+1_init} = computeSetNextLevel(LHSattr_l, l);
        LHSattr_{l+1} = pruneSet(X_{l+1_init}, Y);
        l = l + 1;
    end
end
```

Pseudocode 1. Algorithm main steps

are not included in the candidate generation process; (2) in the presence of attributes having distinct values for each tuple in the dataset – such attributes are excluded from the RHS of the candidate when the support required for the dependency in greater than 1. Furthermore, the input parameters *maxSizeLHS* and *sizeT* contribute in reducing the number of generated candidates and thus the execution time of the algorithm.

After having determined a candidate, it is necessary to verify if it can be a CFD$^{\mathrm{p}}$ and determine which are the values for the attributes in the set T that have to be excluded to obtain a valid CFD$^{\mathrm{p}}$. To perform this step, the algorithm proceeds in computing the tuple equivalence sets[3] for the set of attributes present in the candidate.

The algorithm selects the sets to be excluded and the sets to be accepted in order to obtain a valid dependency: the sets with the same values for the

[3] Two tuples t_1 and t_2 are equivalent respect to a set Y of attributes if $\forall\ B \in Y$ $t_1[B]{=}t_2[B]$.

```
procedure generateCandidates(LHSattr, Y)
    candidateSet = ∅;
    for S ∈ LHSattr do
        setT = buildSetT(S, sizeT);
        for T ∈ setT do
            setXattr = S − T;
            setX = buildSetX(setXattr);
            for X ∈ setX do
                candidateSet += buildCandidate(X, T, Y);
            end
        end
    end
    return candidateSet;

procedure pruneSet(S, Y)
    newSet = ∅;
    for Z ∈ S do
        if Z → Y holds on R then
            newSet = S − Z;
        end
    end
    return newSet;
```

Pseudocode 2. Algorithm procedures

attributes in LHS but different values of the attribute A are excluded. The values of the *target attributes* (the attributes in the set T) of the tuples contained in the excluded sets will be used to build the conditions for the dependency pattern tableau.

At this step, to reduce useless computation, the input parameter *depSupport* – when present – is used to filter out the candidates having in their selected sets a number of tuples less greater than the required support.

Then, as a preliminary step in the determination of the intervals, for every *target attribute*, the minimum distance among the values on the attribute domain is computed. It will be used to determine if the values are contiguous or not and thus to decide for each value if it has to be part of an interval condition or if it will generate an inequality condition.

Afterwards, the algorithm builds a set with the values of the *target attributes* for all the tuples contained in the excluded sets; this last set is used by the algorithm to compute the interval (or intervals) for which the candidate is a valid dependency. Instead of an interval, an equality condition is generated when an open interval contains only one value between the extreme values; e.g., when the interval is $(x - 1, x + 1)$ then the conditions "$> x - 1$" and "$< x + 1$" are replaced by the condition "$= x$".

Because of the semantics of the CFD^{p}s stating that a tuple has to satisfy the conjunction of all the conditions in a pattern tableau, if more than one interval

procedure *findTargetConditions(candidate, numSupportTuples)*
    $patternTableauSet = \emptyset$;
    $equivalentSetsList$ = computeEquivalentSets($candidate$);
    $acceptedSetsList$ = selectSetsToBeKept($equivalentSetsList$);
    if *countTuples(acceptedSetsList)* $\geq$ *numSupportTuples* **then**
        $excludedSetsList$ = selectSetsToBeExcluded($equivalentSetsList$, $acceptedSetsList$);
        $patternTableauSet$ = computeConditions($acceptedSetsList$, $excludedSetsList$);
    end
    return $patternTableauSet$;

procedure *acceptResults(candidate, patternTableau, maxNumConditions, numSupportTuples)*
    if $size(patternTableau) <= maxNumConditions$ **then**
        $numTuples$ = countTuples($candidate$, $patternTableau$);
        if $numTuples >= numSupportTuples$ **then**
            return $true$;
        end
    end
    return $false$;

Pseudocode 3. Algorithm procedures

is identified for a candidate, it is necessary to build different pattern tableaux for that candidate.

If the input parameters *maxNumConditions* and *depSupport* have been set, the last step consists in the acceptance or rejection of the dependency according to the values of these parameters: a dependency is accepted if the number of conditions in its pattern tableau is less than or equal to the *maxNumConditions* parameter and if it is satisfied by a number of tuples greater than or equal to the support required by the *depSupport* parameter.

4 Testing the Algorithm

The algorithm has been implemented using the Java programming language and the PostgreSQL DBMS. The first test of the algorithm has been performed using some of the datasets provided by the UCI Machine Learning Repository [2], such as the Iris, Seeds, Escherichia Coli, BUPA Liver disorder[4], Yeast[5] and Wisconsin breast cancer[6] datasets.

To show some examples of the results produced by the algorithm, we use the following datasets:

[4] In the BUPA Liver disorder dataset duplicate rows have been excluded.
[5] In the Yeast dataset duplicate rows have been excluded.
[6] In the Wisconsin breast cancer dataset the attribute called Sample Code Number and the rows containing empty attributes have been excluded.

Table 2. Results of the execution of the algorithm on the *BUPA Liver* dataset

φ_1: BUPA-liver(alkphos, sgpt, drinks $\rightarrow$ selector, T_1)

T_1:

alkphos	sgpt	drinks	selector
≥ 23	–	–	–
$\neq 85$	–	–	–
≤ 138	–	–	–

φ_2: BUPA-liver(gammagt, mcv, alkphos $\rightarrow$ sgot, T_2)

T_2:

gammagt	mcv	alkphos	sgot
> 5	–	–	–
< 297	–	–	–

φ_3: BUPA-liver(gammagt, mcv, alkphos $\rightarrow$ sgpt, T_3)

T_3:

gammagt	mcv	alkphos	sgpt
> 5	–	–	–
< 297	–	–	–

φ_4: BUPA-liver(sgpt, mcv, gammagt $\rightarrow$ drinks, T_4)

T_4:

sgpt	mcv	gammagt	drinks
≥ 4	–	–	–
$\neq 9$	–	–	–
≤ 155	–	–	–

φ_5: BUPA-liver(sgpt, mcv, gammagt $\rightarrow$ alkphos, T_5)

T_5:

sgpt	mcv	gammagt	alkphos
≥ 4	–	–	–
$\neq 9$	–	–	–
≤ 155	–	–	–

– The *Iris* dataset, which has 5 attributes respectively called Petal Length, Petal Width, Sepal Length, Sepal Width, and Class.
– The *BUPA Liver* dataset, which contains the following 7 attributes (all of them with values in the domain of the integer numbers): Mean Corpuscular Volume (*mcv*), Alkaline Phosphotase (*alkphos*), Alamine Aminotransferase (*sgpt*), Aspartate Aminotransferase (*sgot*), Gamma-Glutamyl Transpeptidase (*gammagt*), number of half-pint equivalents of alcoholic beverages drunk per day (*ndrinks*), and a field used to split data into two sets (*selector*).
– The *Wisconsin breast cancer* dataset, which contains the following 10 attributes (all of them with values in the domain of the integer numbers): Clump Thickness, Uniformity of Cell Size, Uniformity of Cell Shape, Marginal Adhesion, Single Epithelial Cell Size, Bare Nuclei, Bland Chromatin, Normal Nucleoli, Mitoses, and Class; the first 9 attributes have values in the range 1-10, while the Class attribute can have two values: 2 for "benign", 4 for "malignant".

Table 2 shows the CFD^{p}s resulting from the execution of the algorithm on the *BUPA Liver* dataset with the following values for the input parameters:

Table 3. Results of the execution of the algorithm on the *Wisconsin breast cancer* dataset

φ_1: wbc(uniformityCellShape, singleEpithelialCellSize, bareNuclei, normalNucleoli $\rightarrow$ class, T_1)

T_1:

uniformityCellShape	singleEpithelialCellSize	bareNuclei	normalNucleoli	class
≥ 1.0	–	–	–	–
< 7.0	–	–	–	–

φ_2: wbc(bareNuclei, clumpThickness, uniformityCellSize, uniformityCellShape $\rightarrow$ class, T_2)

T_2:

bareNuclei	clumpThickness	uniformityCellSize	uniformityCellShape	class
≥ 1.0	–	–	–	–
< 10.0	–	–	–	–

φ_3: wbc(bareNuclei, uniformityCellShape, marginalAdhesion, singleEpithelialCellSize $\rightarrow$ class, T_3)

T_3:

bareNuclei	uniformityCellShape	marginalAdhesion	singleEpithelialCellSize	class
≥ 1.0	–	–	–	–
< 10.0	–	–	–	–

maxSizeLHS equal to 3 , *sizeT* equal to 1, *maxNumConditions* equal to 3 and *depSupport* equal to 0.98. Table 3 shows the CFD^ps resulting from the execution of the algorithm on the *Wisconsin breast cancer* dataset with the following input parameters: *maxSizeLHS* equal to 4, *sizeT* equal to 1, *maxNumConditions* equal to 2 and *depSupport* equal to 0.8. Table 4 shows the CFD^ps resulting from the execution of the algorithm on the *Iris* dataset with the following input parameters: *maxSizeLHS* equal to 3, *sizeT* equal to 1, *maxNumConditions* equal to 3 and *depSupport* equal to 0.6. Finally, table 4 shows the CFD^ps resulting from the execution of the algorithm on the *Iris* dataset with the following input parameters: *maxSizeLHS* equal to 2, *sizeT* equal to 2, *maxNumConditions* equal to 5 and *depSupport* equal to 0.98.

Depending on the values assigned to the input parameters (in particular to the dependency support parameter), on the number of attributes and tuples in the relation, and, of course, on the type of data, the number of generated CFD^ps can vary greatly.

Table 6 reports the number of CFD^ps identified by the algorithm on different datasets provided by the UCI Machine Learning Repository with different values for the support input parameter *depSupport*. The results shown in the table have been computed with the following input parameters: *maxSizeLHS* equal to 4, *sizeT* equal to 1 and *maxNumConditions* equal to 4; while the values used for the dependency support parameter – called k – are specified in the table.

Furthermore, table 7 reports the number of CFD^ps identified by the algorithm on the same datasets using different values for the input parameter *maxSizeLHS* – the maximum number of attributes in the LHS of the dependency. In this case, the results have been computed with the dependency support equal to 0.5 and

Table 4. Results of the execution of the algorithm on the *Iris* dataset

φ_1: iris(petalLength, sepalLength → class, T_1)

T_1:

petalLength	sepalLength	class
≥ 1.0	–	–
$\neq 4.9$	–	–
≤ 6.9	–	–

φ_2: iris(sepalLength, petalLength → class, T_2)

T_2:

sepalLength	petalLength	class
≥ 4.3	–	–
$\neq 6.3$	–	–
≤ 7.9	–	–

φ_1: iris(sepalWidth, petalLength → class, T_1)

T_3:

sepalWidth	petalLength	class
> 2.8	–	–
≤ 4.4	–	–

φ_1: iris(petalWidth, sepalLength → class, T_1)

T_4:

petalWidth	petalLength	class
≥ 0.1	–	–
$\neq 1.8$	–	–
≤ 2.5	–	–

φ_1: iris(petalLength, petalWidth → class, T_1)

T_5:

petalLength	petalWidth	class
≥ 1.0	–	–
$\neq 4.8$	–	–
≤ 6.9	–	–

sizeT equal to 1 but without any limit on the maximum number of conditions allowed in the resulting pattern tableaux.

The results show that the number of the CFD^{p}s identified by the algorithm increases when the maximum size of LHS increases and – as expected – decreases at the increasing of the dependency support required through the input parameter. The high numbers of dependencies found when the input parameter for the dependency support is not specified is mainly determined by the presence of CFD^{p}s satisfied by a single tuple.

The approach to generate, during the same step, different tableaux for a candidate – producing disjoint intervals – determines that the dependencies generated for the same candidate are not redundant. However, redundant CFD^{p}s can be generated when there exist:

– two CFD^{p}s φ_a:$R(Z_1 \to A,\ T_{p_1})$ and φ_b:$R(Z_2 \to A,\ T_{p_2})$, with $Z_1 \subset Z_2$, $Z_1{=}X_1 \cup T_1$, $Z_2{=}X_2 \cup T_2$, $T_1{=}T_2$ and $X_1 \subset X_2$: if the conditions in T_2 are subsumed by the conditions in T_1 then φ_b is redundant.

Table 5. Results of the execution of the algorithm on the *Iris* dataset

φ_1: iris(sepalLength, petalLength → class, T_1)

T_1:

sepalLength	petalLength	class
≥ 4.3	≥ 1.0	–
$\neq 6.3$	$\neq 4.9$	–
≤ 7.9	≤ 6.9	–

φ_2: iris(petalLength, petalWidth → class, T_2)

T_2:

petalLength	petalWidth	class
≥ 1.0	≥ 0.1	–
$\neq 4.8$	$\neq 1.8$	–
≤ 6.9	≤ 2.5	–

φ_3: iris(sepalWidth, petalLength → class, T_3)

T_3:

sepalWidth	petalLength	class
≥ 2.0	≥ 1.0	–
$\neq 2.7$	$\neq 5.1$	–
$\neq 2.8$	$\neq 4.8$	–
≤ 4.4	≤ 6.9	–

Table 6. Results from the execution of the algorithm with different values of the input parameter *depSupport* (k)

Dataset name	$\|R\|$	$\|r\|$	number of **CFD^{p}s** k not defined	$k \geq 0.1$	$k \geq 0.5$	$k \geq 0.8$
Iris	5	150	274	72	19	8
BUPA Liver	7	341	1413	596	228	126
Seeds	8	210	78	48	38	27
E. Coli	9	336	3307	699	174	139
Wisconsin breast cancer	10	683	6578	1160	72	40
Yeast	10	1462	11236	1540	253	194

Table 7. Results from the execution of the algorithm with different values of the input parameter *maxSizeLHS* (max$\|LHS\|$)

Dataset name	$\|R\|$	$\|r\|$	number of **CFD^{p}s** max$\|LHS\|$=2	max$\|LHS\|$=3	max$\|LHS\|$=4	max$\|LHS\|$=5
Iris	5	150	12	22	32	–
BUPA Liver	7	341	7	135	286	328
Seeds	8	210	76	91	91	91
E. Coli	9	336	66	210	413	558
Wisconsin breast cancer	10	683	0	6	74	198
Yeast	10	1462	17	143	382	659

Table 8. Results of the execution of the algorithm on the *Iris* dataset

φ_1: iris(petalWidth → class, T_1)

T_1:

petalWidth	class
≥ 0.1	–
< 1.4	–

φ_2: iris(petalWidth, sepalLength → class, T_2)

T_2:

petalWidth	sepalLength	class
≥ 0.1	–	–
< 1.4	–	–

φ_3: iris(sepalLength, petalWidth → class, T_3)

T_3:

sepalLength	petalWidth	class
≥ 4.3	–	–
< 5.9	–	–

φ_4: iris(sepalWidth, petalLength → class, T_4)

T_4:

sepalWidth	petalLength	class
> 2.8	–	–
≤ 4.4	–	–

- two CFD^{p}s φ_a:$R(Z_1 \to A, T_{p_1})$ and φ_b:$R(Z_2 \to A, T_{p_2})$, with $Z_1 \subseteq Z_2$, Z_1=$X_1 \cup T_1$, Z_2=$X_2 \cup T_2$, $T_1 \subset T_2$: if the conditions in T_2 are subsumed by the conditions in T_1 then φ_b is redundant.

However, the support of the CFD^{p}s can be different, and it can be higher for the dependency φ_b.

An example of the first case is shown in table 8 with the results from the execution of the algorithm on the *Iris* dataset (the following input parameters have been used: *maxSizeLHS* equal to 2, *sizeT* equal to 1, *maxNumConditions* equal to 2 and *depSupport* equal to 0.5), in particular the CFD^{p}s φ_1 and φ_2; whereas an example of the second case can be observed comparing table 4 and table 5.

5 Related Work

For the discovery of *non-constant* CFD^{p}s, to date and to our knowledge, there are no published algorithms.

Similarities between CFD^{p}s and approximate functional dependencies[7] [12] can be highlighted: in both cases a dependency holds excluding a subset of the set of tuples. However, the process to find a CFD^p requires the identification of the *target conditions* contained in the pattern tableau, while in the case of

[7] An approximate FD is a FD that does not hold over a small fraction of the tuples; specifically, $X \to Y$ is an approximate FD if and only if the $error(X \to Y)$ is at most equal to an error threshold ϵ ($0 < \epsilon < 1$), where the error is measured as the fraction of tuples that violate the dependency.

approximate dependencies it is sufficient to determine the number of tuples non-satisfying the dependency.

Several algorithms for the discovery of FDs have been proposed since 1990s and more recently for CFDs.

Examples of algorithms developed to discover traditional FDs are: TANE [11], Dep-miner [14], Fast-FD [17], FD_Mine [18].

For the discovery of general CFDs the following algorithms have been proposed: an algorithm based on the attribute lattice search strategy is presented in [4]; Fast-CFD [9] is inspired by the Fast-FD algorithm; CTANE [9] extends the TANE algorithm; CFD-Mine [1] is also based on an extension of the TANE algorithm. Moreover, some algorithms for the discovery of only constant CFDs have been proposed: CFDMiner [9] is based on techniques for mining closed item sets and finds a canonical cover of k-frequent minimal constant CFDs; an algorithm that extends the notion of non-redundant sets, closure and quasi-closure is described in [6]; in [13] new criteria to further prune the search space used by CFDMiner to discover the minimal set of CFDs are proposed.

6 Conclusions and Future Work

In this work we have introduced an algorithm to discover *non-constant* $\mathrm{CFD^p}$s from datasets. Aim of the developed algorithm is the identification of a subset of the existing *non-constant* $\mathrm{CFD^p}$s characterized by the requirements mentioned in section 3, without looking specifically for CFDs, for which dedicated algorithms already exist. The algorithm implements the approach of selecting the tuples that do not satisfy a dependency and using the values of the attributes of the identified tuples to build the *target conditions* to obtain valid dependencies.

The results of the first algorithm test, which has been executed on datasets from the UCI Machine Learning Repository, show that the number of $\mathrm{CFD^p}$s generated by the algorithm can vary greatly depending on the values assigned to the input parameters, on the number of attributes and tuples in the relation, and – of course – on the type of data. When too many $\mathrm{CFD^p}$s are retrieved from a dataset, the input parameters – in particular the *depSupport* and *maxSizeLHS* parameters – help in decreasing the number of identified dependencies. A high value of the *depSupport* parameter determines also the identification of the most interesting dependencies to be practically used in the data quality context.

As in the case of the algorithms for discovering FDs [15], the worst case time complexity of the developed algorithm, with respect to the number of attributes and tuples in the relation, is exponential. The criteria used by the algorithm to prune the number of candidates and the input parameters help in improving the algorithm efficiency as in reducing the number of identified dependencies.

As future work we plan to test the algorithm on other datasets and to experiment with other candidate pruning approaches to improve the algorithm efficiency. We are also studying the feasibility of an extension to the algorithm in order to include non-numeric attributes in the *target attribute* set T, considering the alphanumeric ordering or a semantic ordering defined on the domains of the relation attributes.

References

1. Aqel, M., Shilbayeh, N., Hakawati, M.: CFD-Mine: An efficient algorithm for discovering functional and conditional functional dependencies. Trends in Applied Sciences Research 7(4), 285–302 (2012)
2. Bache, K., Lichman, M.: UCI Machine Learning Repository (2013), `http://archive.ics.uci.edu/ml`
3. Chen, W., Fan, W., Ma, S.: Analyses and validation of conditional dependencies with built-in predicates. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) DEXA 2009. LNCS, vol. 5690, pp. 576–591. Springer, Heidelberg (2009)
4. Chiang, F., Miller, R.: Discovering data quality rules. Proceedings of the VLDB Endowment 1(1), 1166–1177 (2008)
5. Cong, G., Fan, W., Geerts, F., Jia, X., Ma, S.: Improving data quality: Consistency and accuracy. In: Koch, C., et al. (eds.) International Conference on Very Large Data Bases (VLDB 2007), pp. 315–326. ACM (2007)
6. Diallo, T., Novelli, N., Petit, J.M.: Discovering (frequent) constant conditional functional dependencies. Int. Journal of Data Mining, Modelling and Management 4(5), 205–223 (2012)
7. Fan, W., Geerts, F., Jia, X.: Semandaq: A data quality system based on conditional functional dependencies. Proceedings of the VLDB Endowment 1(2), 1460–1463 (2008)
8. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for capturing data inconsistencies. ACM Transactions on Database Systems (TODS) 33(2), 94–115 (2008)
9. Fan, W., Geerts, F., Li, J., Xiong, M.: Discovering conditional functional dependencies. IEEE Transactions on Knowledge and Data Engineering (TKDE) 23(5), 683–697 (2011)
10. Golab, L., Karloff, H., Korn, F., Srivastava, D.: Data Auditor: Exploring data quality and semantics using pattern tableaux. Proceedings of the VLDB Endowment 3(2), 1641–1644 (2010)
11. Huhtala, Y., Karkkainen, J., Porkka, P., Toivonen, H.: TANE: An efficient algorithm for discovering functional and approximate dependencies. Computer Journal 42(2), 100–111 (1999)
12. Kivinen, J., Mannila, H.: Approximate inference of functional dependencies from relations. Theoretical Computer Science 149(1), 129–149 (1995)
13. Li, J., Liu, J., Toivonen, H., Yong, J.: Effective pruning for the discovery of conditional functional dependencies. The Computer Journal 56(3), 378–392 (2013)
14. Lopes, S., Petit, J.-M., Lakhal, L.: Efficient discovery of functional dependencies and Armstrong relations. In: Zaniolo, C., Lockemann, P.C., Scholl, M.H., Grust, T. (eds.) EDBT 2000. LNCS, vol. 1777, pp. 350–364. Springer, Heidelberg (2000)
15. Mannila, H., Raiha, K.J.: On the complexity of inferring functional dependencies. Discrete Applied Mathematics 40, 237–243 (1992)
16. Pivert, O., Prade, H.: Handling dirty databases: From user warning to data cleaning — Towards an interactive approach. In: Deshpande, A., Hunter, A. (eds.) SUM 2010. LNCS, vol. 6379, pp. 292–305. Springer, Heidelberg (2010)

17. Wyss, C., Giannella, C., Robertson, E.: FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. In: Kambayashi, Y., Winiwarter, W., Arikawa, M. (eds.) DaWaK 2001. LNCS, vol. 2114, pp. 101–110. Springer, Heidelberg (2001)
18. Yao, H., Hamilton, H.: Mining functional dependencies from data. Journal Data Mining and Knowledge Discovery 16(2), 197–219 (2008)
19. Zanzi, A., Trombetta, A.: Data quality evaluation of scientific datasets: A case study in a policy support context. In: International Conference on Data Management Technologies and Applications (DATA 2013), pp. 167–174. SciTePress (2013)