A Domain-Independent Data Cleaning Algorithm for Detecting Similar-Duplicates

Kazi Shah Nawaz Ripon
Department of Informatics, University of Oslo, Norway
Computer Science and Engineering Discipline, Khulna University, Bangladesh
Email: ripon_cseku@yahoo.com

Ashiqur Rahman and G.M. Atiqur Rahaman
Computer Science and Engineering Discipline, Khulna University, Bangladesh
Email: {2ashiqur, gmatiqur}@gmail.com

Abstract—**Data mining algorithms generally assume that data will be clean and consistent. However, in practice, this is not always the case, and for this reason the detection and elimination of duplicate records is an important part of data cleaning. The presence of similar-duplicate records causes over-representation of data. If the database contains different representations of the same data, the results obtained from the data mining algorithm will be erroneous. The detection of similar-duplicate records is a difficult task, especially when the records are domain-independent. In this paper, we propose a novel domain-independent technique for better reconciling the similar-duplicate records. We also introduce new ideas for making similar-duplicate detection algorithms faster and more efficient. In addition, a significant modification of the transitivity rule is also proposed. Finally, we propose an algorithm that incorporates all these techniques for similar-duplicate detection into a domain-independent environment. The performance of the proposed method has been compared to other methods and the superiority of the proposed method has been confirmed by the experimental results.**

Index Terms—**Data cleaning, similar-duplicate, domain-independent, transitivity rule, approximate duplicate.**

I. Introduction

Data mining [1] is a knowledge discovery process used to extract hidden patterns from data. There are several steps involved in data mining as shown in Fig. 1 [2]. One of the major steps is data preparation; data cleaning (also known as data cleansing or scrubbing) is one of the sub-steps of data preparation. The intent of the process is to detect and remove errors and inconsistencies from data and improves their quality. Duplicate record detection and elimination is a challenge of data cleaning. Data cleaning is applicable, for example, for textual databases, image databases, and web mining. The main focus of this paper is to improve the methodologies used for detecting duplicate records in textual databases.

The importance of databases in today's information technology (IT) sector is overwhelming. The success rate of knowledge discovery operations performed on databases is highly dependent on the quality of data. For several reasons, the data quality may fall below the expected level. Data duplication is one such reason. A decrease in quality of stored data puts the knowledge discovered from such data into question. It is not too difficult to detect exact duplicates, but the situation becomes complex when an entity in the database contains similar-duplicates. Linguistically, *similar* means like but not exactly the same, and *duplicate* means the same in every detail. In similar-duplicates, entities are not identical in a bit-by-bit comparison because of errors, but in the absence of errors, they would actually be identical/duplicate. Here the term *similar-duplicate* is used to indicate strings that are not textual duplicates but are similar, and from measurement of their similarity, we can decide whether they are actually identical or not. Some researchers refer to similar-duplicates as *approximate duplicates* [3]. Records can be similar-duplicates for several reasons, including typing errors, abbreviations, extra words, missing values, word transpositions, illegal values, and similar mistakes. For example, "Mother" and "Mother" are identical, but "Mother" and "Mather" are not. However, the terms are similar if one considers the mismatch in characters between the two strings. Either "o" or "a" might be a typing error. Now, if one uses a string-matching algorithm which measures the similarity between "Mother" and "Mather," then from the resultant similarity, one can decide whether "Mother" and "Mather" are identical or not.

Over the years, a great deal of attention has been paid to resolving the problem of duplicate detection. Yet, most of the research conducted in this field has addressed domain-specific problems. Unfortunately, domain knowledge is not always available. Moreover, domain-specific methodologies apply only for some particular domains, and the rules developed for one domain often do not hold for different domains. Nevertheless, currently so many different domains exist that the need for domain-independent research is undeniable. Although dealing with domain-independent techniques for similar-duplicate record detection in database has received attention over the last few years, to our knowledge these approaches are

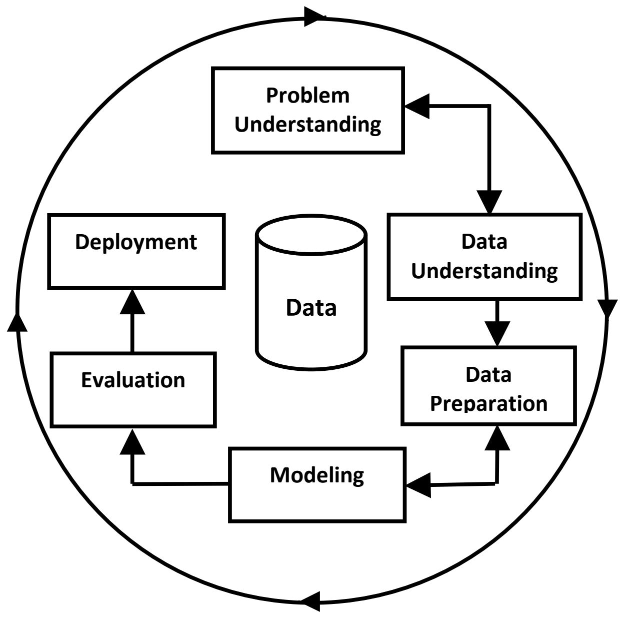

Figure 1. Data mining process [2].

still limited, and dominated by pure or advanced transitivity rule for domain-specific techniques [4]. As far as we know, there is no solution for domain-independent cases.

Usually, most of the database table contains unique or primary key fields, and these fields play a major role in restricting data from being duplicated during different update/retrieval processes. However, errors during entry in these fields, which contain mostly unique data, are a common source of inaccuracy in databases. On the one hand, if the unique data is entered manually, then the chance of error is very high. On the other hand, to reduce this type of error, these data are typically entered using drop-down or combo boxes. Sometimes, they are uniquely auto generated during the data entry process. This reduces the chance of errors in fields with unique values. However, it only guarantees that no two instances will have the same primary/unique key value; it does not guarantee that the same instance will not hold for two different primary key values. For example, a university staffer might want to enter student details into a database. During the data entry process, the staffer can select different student IDs from the drop-down/combo box or from a uniquely auto generated ID. Although this guarantees that no two students will have the same IDs, it never guarantees that in the database the same student will not be assigned two different IDs. The same problem holds true for the fields with common values. Still, there is no suitable solution to these types of problems. Similar-duplicate detection is the only method to handle them to date. As mentioned earlier, very few research works discuss the issue of better reconciling similar-duplicate records (bringing similar-duplicate records nearer), and the existing de-duplication techniques for doing so are not efficient enough for such types of problems [3, 5-7].

This paper presents a domain-independent technique for sorting similar records and a modification to the transitivity rule to apply into domain-independent cases. It also considers time for performing the de-duplication on real-world data and proposes a new idea for making the duplicate detection algorithms faster for real-world data. In addition, modification has been proposed to an existing similar-string matching algorithm [8] for improving efficiency. Finally, an algorithm implements all these techniques together. Experimental results show that the proposed schemes perform better than the existing domain-independent similar-string matching schemes. The rest of the paper is organized as follows. Section II contains literature review. Section III presents the background of the proposed methods. Section IV describes proposed methods and algorithms in details. To demonstrate the performance of the proposed approach, experimental results are presented and analyzed in Section V. Section VI contains some limitations and points to future works, followed by the conclusions in the final section.

II. Literature Review

The standard method for detecting exact duplicates is to sort the dataset and then to perform an exact matching with the neighbor records to determine whether two records are duplicates or not. Different authors have extended this approach for detecting the similar/approximate duplicate records in databases [9-13]. These approaches vary by the amount of domain knowledge supplied by the user. Some of these approaches are domain-specific and/or assumes that the user for each application domain will supply the domain knowledge. Lee *et al.* proposed a framework for data cleaning [4]. They addressed the problem of transitivity rule in the field of similar-duplicate detection, and proposed a method for avoiding this problem. Still, their proposed method is domain-dependent and requires some rules and certainty factor (*c.f.*). To specify these rules, domain knowledge is necessary. It also requires user involvement for defining the *c.f.* Wang *et al.* proposed another rule based duplicate detection methodology [10]. They used experts' knowledge to create rules and key from a set of attributes when the global key is absent. Some authors also proposed to create production rules based on domain-specific knowledge. However, creating rules is often time consuming. Certainly, duplicate detection systems should use as less domain knowledge as possible [14]. In addition, the result must be acceptable enough for keeping the time duration in a tolerable range. The reason is that usually data mining processes are applied on databases, which contain huge amount of data.

Hernandez and Stolfo proposed some of the earlier works for similar-duplicate detection [5, 7]. All these works are almost similar and based on Merge/Purge technique. However, their proposed Merge/Purge technique gives solution only for domain-specific problems. The authors used a window-based system to limit the number of comparisons required for a record with its neighbor records. It utilized equational theory that needs to be changed for each domain and also requires domain knowledge. In addition, it used a sorting technique based on keys selected by user. This technique fails, if the user makes a poor choice to sort the data. To increase the efficiency of the data cleaning method, Lee *et al.* proposed a token-based technique for data preprocessing [6]. This technique requires external source files, which contain useful information to remove typical errors from the dataset. Unfortunately, it can be used only for some particular domain-specific problems, and for most cases, it is not possible to get such an external source file. For sorting the dataset, they proposed a method similar to the method proposed in [5]. Their sorting method involved three stages. In the first stage, the data in each field of the selected attribute are tokenized, next the tokens are internally sorted and then they are used for sorting the dataset. This method fails, if the record contains errors in the selected attribute. Monge and Elkan proposed a completely domain-independent work [3]. Their proposed sorting method treats each record as one long string and sorts them lexicographically reading from left to right in one pass and right to left in another pass. The authors also proposed a priority queue based technique for limiting the number of comparisons. To group similar records they used pure transitivity rule, but they also mentioned some problems of using transitivity rule for detecting similar-duplicates. In [9], the authors presented a concept in which prediction is made by assigning weight proportional to the importance of information and using co-occurrence and textual similarity functions. However, this work is partially domain-independent and the memory and time requirement for performing the required operations is very high. Hylton proposed an algorithm for identifying and merging related bibliographic records, which are in different formats [11]. It was designed to work using authors' names and titles. It works only for bibliographic records; hence, it is domain-specific. Winkler proposed a method, which can be applied for a limited range of very good quality lists that have known matching characteristics [12]. Weis *et al.* proposed an industry-scale duplicate detection system [13], which is an extension of the domain-independent approach proposed for detecting duplicates in XML data [15]. However, the proposed industry-scale duplicate detection system is totally domain-dependent and requires rules and the knowledge of domain experts'.

Similar-string matching is one of the key requirements for similar-duplicate detection systems. Several works are available on similar/approximate string matching in the literature [16-19]. Unfortunately, most of them are designed to solve particular problems. The well-known Boyer and Moore's algorithm [17] is very effective to find a pattern in a string, but it is a kind of exact matching algorithm and does not hold for similar-string matching. Du *et al.* proposed an approximate string matching algorithm to solve only dictionary problems [18]. Smith-Waterman algorithm [19] is another well-known algorithm, which was originally developed to compare DNA or protein sequences. The most popular method for similar-string matching is based on edit-distance [20], which is the minimum number of unit operations (for example, add, delete, insert) needed to perform on individual characters to transform one string to the other [16, 21]. These types of functions are also called classical edit-distance functions, or textual similarity functions. Monge showed that recursive algorithm gives more accurate result for approximate string matching than previous classical algorithms [22]. Still, the complexity of the proposed algorithm is very high due to its recursive comparisons of sub-strings. On the other hand, Udechukwu *et al.* claimed that their proposed Positional algorithm improves the result of the recursive algorithm [8]. Conversely, experimental result shows that, for some cases, the original Positional algorithm fails to return an acceptable score. This situation arises when the shorter string is a sub-string of the longer string. However, we believe that with some modification, it can be made more efficient. In this paper, we presented a modified version of the Positional algorithm and the experimental results are satisfactory enough.

III. Background of The Proposed Method

The sorting method proposed in [6] is not completely domain-independent. In fact, it is partially domain-independent, as it requires the user to select the most important fields for sorting the similar records. A completely domain-independent technique for sorting the records was proposed in [3]. However, this technique fails, if the database table contains most unique field in one side and less unique field in the other side. It also fails, if error either in the most unique field or another record is similarly duplicated for error in the first and last field. Whereas our proposed method works efficiently for solving the problem of error in unique field or when the error is in some other fields, which has a high uniqueness value compared to that of others.

TABLE I.
Dataset with all unique records

	Field-1	Field-2	Field-3
R1	A	B	C
R2	D	E	F
R3	G	H	I
R4	J	K	L

TABLE II.
Dataset with similar-duplicate records

	Field-1	Field-2	Field-3
R1	A	B	C
R2	D	E	F
R3	G	H	I
R4	J	B	L

TABLE III.
Dataset with similar-duplicate records

	Field-1	Field-2	Field-3
R1	A	B	C
R4	J	B	L
R2	D	E	F
R3	G	H	I

Table I represents the dataset, where data in all records are unique or they become unique due to some errors. In this scenario, it is really hard to determine the sorting strategy. However in a real-world dataset, it is natural that some records will be similarly duplicates and we can assume that some of its values will be exactly duplicates as in Table II. In these cases, the uniqueness of the fields will be affected.

For example, in Table I, uniqueness of Field-1, Field-2 and Field-3 are 100%, but in Table II, uniqueness of Field-2 is 75%. In Table II records R1 and R4 may be similar or not. If we use the uniqueness information and sort the dataset using the less unique field, R1 and R4 will be closer as in Table III.

Similar-string matching algorithms are expensive [3]. They have to perform more computations than exact string matching algorithms. In a real-world dataset, the number of unique records will be many times higher than the number of duplicate records. Hence, performing an efficient similar-string matching algorithm is not required when the number of characters in a string is much lower than the number of characters in the second string or the characters in one string does not appear in a certain amount in the second string. Therefore, we decided to use a fast string matching algorithm that calls the efficient similar-string matching algorithm when the number of characters and the appearance of characters between two strings pass a threshold value. This will save a lot of computational efforts.

IV. Proposed Modifications and Algorithms

A. Modification of the Positional Algorithm

Most of the existing string matching algorithms do not handle abbreviations efficiently [23], and for many cases, they do not give acceptable measures of similarity. One of the efficient similar-string matching algorithms is Positional string matching algorithm [8]. It can handle abbreviations efficiently and is able to measure the similarity between two strings more efficiently than the classical string matching algorithms. However for some cases, it fails to return a good measure of similarity.

For example, consider two strings: A="John" and B="Johnathon". Using the original Positional algorithm, we get a match score of 1; whereas there are 5 characters mismatch between A and B. A match score of 1 usually means that the strings are exact duplicates. Therefore, the match score of 1 is not acceptable. In fact, the original Positional algorithm will return a match score of 1, irrespective of the number of characters exist in the left or right side of "John" of the second string. This is because during the match score calculation, the algorithm considers only the length of the shorter string ignoring the length of the other string. As was doing in the original Positional algorithm:

match score = (total match score returned by the shorter string / total number of characters in the shorter string).

Here only the shorter string returns the match score. However, it is reasonable to consider the longer string also. In fact, the size of both strings has its effect on the calculation of the final match score. In this work, we calculate the mach score as,

match score = (total match score returned by the shorter string / average number of characters in both strings)

Using the latest score for the above problem, we get a match score of 0.61, which is more acceptable.

B. Domain-Independent Transitivity Rule for Similar-Duplicate Detection

Transitivity rule states that if A is equivalent to B and B is equivalent to C then A is equivalent to C. Transitivity rule is used for similar-duplicate detection. Ref. [4] describes the problem of using transitivity rule for similar-duplicate record detection using one character threshold so that two strings will be similar. If there is only one character mismatch then "Father" is similar-duplicate of "Mather" and "Mather" is similar-duplicate of "Mother". According to the transitivity rule, "Father" is similar-duplicate of "Mother". Since there are two character mismatches between "Father" and "Mother"; they are not similar-duplicates according to our threshold. For these types of problems, a very high threshold is required to separate the values. However, a very high threshold will lower the value of Recall. A solution to the transitivity rule for detecting similar-duplicate records was proposed in [4] to obtain a high Recall without declaring these types of values as similar. However, all of their proposed solutions are rule based and it requires *c.f.*; which in turn requires the user involvement. Moreover, it was domain-specific. So far we know there is no solution for domain-independent cases to reduce the effect of the transitivity rule. Our proposed Domain-Independent Transitivity (DIT) algorithm (described in Algorithm-1) for similar-duplicate detection is totally domain-independent, and it does not require any rule, *c.f.*, or user involvement. In DIT algorithm, SIM stands for similarity and PR stands for power of a record. We call the latest record (the highest numbered record) in a window as the window creator. Input to the algorithm is two records: window creator and record to which the window creator declares similarity (i.e. passes a threshold value). It should be noted that if the window creator does not declare similarity with some records, then there is no need to invoke DIT.

Algorithm 1: DIT

```
1. Assign PR=1 to the record, which creates the window, call it
   R2.
2. Calculate R1's PR = (similarity between R2 and R1)*(PR of
   R2) and perform the following operations,
        (2a) If R1 is a representative of a set then recalculates
the PR of each record of the set with respect to the new PR of R1
in the set of R2.
            Move all records from the set of R1 to the set of R2
whose new PR passes the threshold value.
```

From the remaining records of the set of R1, make a new representative, if any exists, and update the PR of other records with respect to this new representative.

(2b) Otherwise, if R1 is not a representative then if it's PR is higher in the set of R2 then move it to the set of R2 and update it's PR, otherwise not.

(2c) Otherwise, it is the first time, R1 is declaring similarity with some record, simply add it to the set of R2 and record it's PR in the set of R2

Consider a pre-defined threshold of 0.7 for declaring two records as duplicates. If the similarity between records R2 and R1 is ≥ 0.7, then R1 is a duplicate of R2 and a set {R2, R1 (this set is open because it still can grow), will be created. Then, we associate a value PR with each record in the set. This value is the measurement of the ability of a record to bring another similar record in the set to which it belongs. If R2 declares a match with R1 in the dataset, then $PR_{R2} = 1$ and PR_{R1} = (similarity between R2 and R1)*(PR_{R2}). If PR_{R1} is above the threshold value then a set {(R2, 1), (R1, PR_{R1}), will be created. Suppose, there is another record R which was similar to R1, then PR_R = (similarity between R1 and R)*(PR_{R1}), will be recalculated with respect to R1's new PR in the new set and if the PR of R passes the threshold, then also it will be moved to the set where R1 has been moved. Otherwise R becomes the representative of the set.

If R1 is exact duplicate of R2, then $SIM_{R2,R1} = 1$ and $PR_{R2} = 1$. As a result, $PR_{R1} = SIM_{R2,R1}*PR_{R2} = (1)*(1) = 1$. $PR_{R1} > 0.7$, so, R1 will be accepted as a duplicate of R2.

If R1 is similar to R2 by half, then $SIM_{R2,R1} = 0.5$ and $PR_{R2} = 1$. Hence, $PR_{R1} = SIM_{R2,R1}*PR_{R2} = (0.5)*(1) = 0.5$. $PR_{R1} < 0.7$, so, R1 will not be accepted as a duplicate.

If $SIM_{R2,R1} = 0.9$ and $PR_{R2} = 1$, then $PR_{R1} = SIM_{R2,R1}*PR_{R2} = (0.9)*(1) = 0.9$. Since $PR_{R1} > 0.7$, R1 will be accepted as duplicate. Now if R is exact duplicate to R1, then $PR_R = SIM_{R1,R}*PR_{R1} = (1)*(0.9) = 0.9 = PR_{R1}$. So, in case of exact duplicate, PR is preserved.

However, if $SIM_{R2,R1} = 0.9$ and $PR_{R2} = 1$, then $PR_{R1} = SIM_{R2,R1}*PR_{R2} = (0.9)*(1) = 0.9$. $PR_{R1} > 0.7$, so, R1 will be accepted as duplicate of R2. If, now $SIM_{R1,R} = 0.8$, then $PR_R = SIM_{R1,R}*PR_{R1} = (0.8)*(0.9) = 0.72$. Since $PR_R > 0.7$, R will be accepted as duplicate in the set where R2 and R1 both exist.

Whereas, If $SIM_{R2,R1} = 0.8$ and $PR_{R2} = 1$, then $PR_{R1} = SIM_{R2,R1}*PR_{R2} = 0.8$. $PR_{R1} > 0.7$, so, R1 will be accepted as duplicate of R2. If now, $SIM_{R1,R} = 0.8$, then $PR_R = SIM_{R1,R}*PR_{R1} = 0.64$. $PR_R < 0.7$, so, R will not be accepted as a duplicate in the set where R2 exists. In order to be duplicate, $SIM_{R1,R}$ must be greater than 0.87.

Here the achievement is that, as R1 is diverging from R2, R is needed to be more similar to R1 in order to become the member of the set where R2 exists. From our experimental result, we found that although this method does not compare R2 and R directly, we get the approximate value of matching R2 and R straight away. If there is no similarity at all between R2 and R, then we get a value which is approximately zero or negligible.

C. Domain-Independent Duplicate Detection Algorithm (DIDD)

Our proposed Algorithm-2, Domain-Independent Duplicate Detection Algorithm (DIDD), is totally domain-independent. All the proposals made in this paper, can be used together using the DIDD algorithm. Moreover, with some simple modifications in the existing system, any part of the proposal made in this paper is also applicable with existing similar-duplicate detection system.

Algorithm 2: DIDD

```
1. Determine the uniqueness of each attribute of the dataset.
   Uniqueness is a numerical value, which returns the number
   of different data of an attribute.
2. Start sorting with the less unique attribute and sort the dataset
   using each attribute. If no such attribute can be determined
   then sort the dataset using any attribute which is more
   promising to bring the similar-duplicate records nearer than
   others.
3. Set up a window of length $n$.
4. Perform a less expensive and fast string matching operation
   between a new (the $n$-th record in the window) record and the
   remaining ($n-1$) records in the window one by one. If the
   matching score is below some pre-specified threshold value
   then continue with step 4.
5. Perform an expensive but efficient string matching operation
   only for those records whose similarity is above some pre-
   specified threshold in step 4.
6. If the similarity between two records in step 5 is above some
   pre-specified threshold and any of them already exist in a set
   of duplicate records then use the modified form of the
   transitivity rule. If none of them exist in any set then create a
   new set of duplicate records with these two records.
7. If the last record from any set of duplicate records goes outside
   the window then close that set of duplicate records. Save this
   set of duplicate records.
8. Repeat steps 4 to 7 until the last record of the dataset enter the
   window and is matched using step 4.
9. Take the set of duplicate records created in step 6 and keep a
   record from each set and delete the remaining records.
```

As in Fig. 2, after getting the initial input dataset, the DIDD algorithm attempts to sort it to bring the similar-duplicate records nearer. After that, it utilizes window-based system to limit the number of comparisons. The purpose of this step is to determine the neighbor records with which each record will be directly compared. At this stage, the DIDD algorithm uses two different string-matching algorithms to reduce the use of expensive but efficient similar-string matching algorithm for every pair of tuples found. The first string-matching algorithm measures the similarity between the two selected records

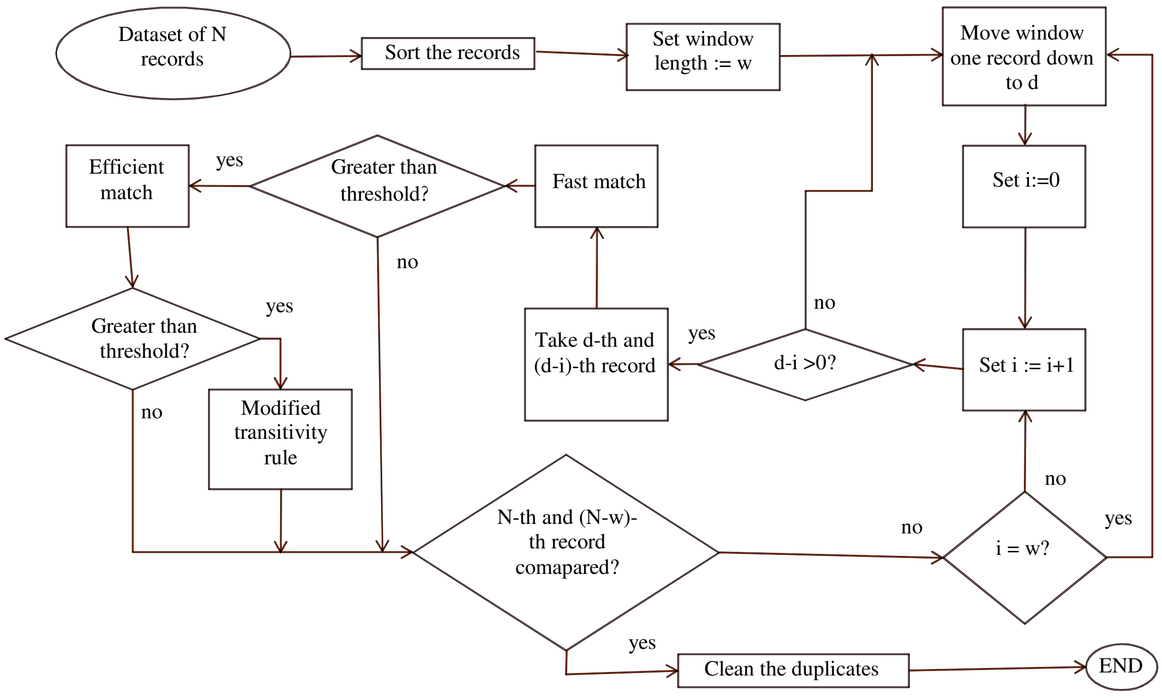

Figure 2. Flowchart of the proposed DIDD algorithm.

very quickly to estimate the chance of these two records to be identical. Comparing the similarity with a pre-defined threshold value serves this purpose. If it finds any reasonable chance, the second string-matching algorithm is invoked with the selected records; otherwise the DIDD algorithm checks the dataset for other records. In that case, the second string-matching algorithm calculates the similarity between the received records very efficiently. It is very important to get an efficient matching score because these similarity scores will be used again to apply the modified transitivity rule among the members of similar group. If the similarity measured by the second string-matching algorithm passes a threshold value, then the modified transitivity rule is applied for grouping the similar records. If there remain more records to be checked, then the process of measuring the similarity between two records are repeated. Otherwise the DIDD algorithm calls up any existing cleaning system to clean the selected duplicates and then finishes.

V. Experimental Results

A. Observation of the working process of DIT

Example-1

Input: FATHER, MATHER, MOTHER
Window length: 2
Threshold for fast comparison: 0.7
Threshold for word matching in positional comparison: 0.7
Threshold for row matching in positional comparison: 0.7
Data_table =
'FATHER'
'MATHER'
'MOTHER'

After duplicate detection and cleaning output: FATHER, MOTHER.

In the first step, comparing "MATHER" and "FATHER", we get a similarity value of 0.8. Our threshold is 0.7, so "MATHER" and "FATHER" are duplicates according to our threshold. They create a set {(MATHER, 1), (FATHER, 0.8)}. Since there is only one character mismatch in "MATHER" and "FATHER", any matching algorithm will declare them as duplicates.

In the second step, comparing "MOTHER" and "MATHER", we get a similarity value of 0.8. Since the threshold is 0.7, "MOTHER" and "MATHER" are duplicates according to our threshold value. They create a set {(MOTHER, 1), (MATHER, 0.8)}. Since there is only one character mismatch in "MOTHER" and "MATHER", any matching algorithm will declare them as duplicates.

In this stage, according to our modified rule "MATHER" will check the set it created earlier for any value which has a PR >= threshold with respect to MATHER's new PR. If such one exists, "MATHER" will bring it to the set where it has joined now. "MATHER" will find "FATHER" but FATHER's PR with respect to MATHER's new PR is 0.8*0.8 = 0.64 < threshold (0.7). As a result, "FATHER" will not come in the set where "MOTHER" exists. "MATHER" will leave all those values whose PR is less than the threshold with respect to MATHER's new PR in the previous set. In this stage (MOTHER, MATHER) and (FATHER) are creating two different sets.

In the third step “MOTHER” will do a direct match with “FATHER” and will get a value of 0.66 < threshold (0.7). So, it will not be declared as duplicate with “FATHER”.

From the direct match result and our threshold value, we can conclude that “MOTHER” is not duplicate of “FATHER” and the same result was obtained by our algorithm even when we did not compare “MOTHER” and “FATHER” directly. On the other hand, in the second step when “MATHER” was trying to take “FATHER” in the set where “MOTHER” exists, it will found that FATHER’s PR = 0.64 in the set where “MOTHER” exists which is ≈ 0.66. This is the direct match result of “MOTHER” and “FATHER”.

If we use the traditional transitivity rule, then we will get only one element instead of two. Because, “MATHER” is duplicate of “FATHER”. “MOTHER” is a duplicate of “MATHER”, so “FATHER” is a duplicate of “MOTHER”.

Example -2

Input: CSE, COMPUTER SCIENCE AND ENGINEERING, COMPUTER SCIENCE AND TECHNOLOGY, CHITTAGONG STOCK EXCHANGE

Window length: 3

Threshold for fast comparison: 0.75

Threshold for word matching in positional comparison: 0.8

Threshold for row matching in positional comparison: 0.75

Data_table =

'CSE'
'COMPUTER SCIENCE AND ENGINEERING'
'COMPUTER SCIENCE AND TECHNOLOGY'
'CHITTAGONG STOCK EXCHANGE'

After sorting the Data_table =

'CHITTAGONG STOCK EXCHANGE'
'COMPUTER SCIENCE AND ENGINEERING'
'COMPUTER SCIENCE AND TECHNOLOGY'
'CSE'

After duplicate detection and cleaning output: COMPUTER SCIENCE AND TECHNOLOGY, CSE.

After comparing “CHITTAGONG STOCK EXCHANGE” and “COMPUTER SCIENCE AND ENGINEERING”, fast match result is 0.5, so no need to compare it further. After comparing “COMPUTER SCIENCE AND ENGINEERING” and “COMPUTER SCIENCE AND TECHNOLOGY”, we find a similarity value of 0.75.

After comparing “CHITTAGONG STOCK EXCHANG”' and “COMPUTER SCIENCE AND TECHNOLOGY”, fast match result is 0.7167, so no need to compare it further. After comparing “COMPUTER SCIENCE AND TECHNOLOGY” and “CSE”, fast match result is 0.6667; again no need to compare it further.

After comparing “COMPUTER SCIENCE AND ENGINEERING” and “CSE”, we find a similarity value of 1, using the matching rule for abbreviation. Previously “COMPUTER SCIENCE AND ENGINEERING” was a member of the set started by “COMPUTER SCIENCE AND TECHNOLOGY”, but in that set, it had a PR = 0.75. If it moves to the set started by “CSE”, then PR = 1. As a result, it moves to the set started by CSE. After comparing “CHITTAGONG STOCK EXCHANGE” and “CSE”, we also find a similarity value of 1.

At the end, we find that the set started by “CSE” has two values: “COMPUTER SCIENCE AND ENGINEERING” and “CHITTAGONG STOCK EXCHANGE”, as a result “CSE” is kept as unique and the remaining two are deleted as duplicates. And the set started by “COMPUTER SCIENCE AND TECHNOLOGY” has no duplicate value, so it is also kept as unique.

B. Experiment with large datasets

The experiment was performed using 7 tests to compare the time requirement for the proposed algorithm with two existing domain-independent algorithms: Scheme-1 [8] and Scheme-2 [3]. We used desktop computer with Intel Pentium-IV processor, 512 MB RAM, windows XP operating system and Matlab-R2007b. The dataset was created by collecting data from the Internet [24-26] and merging them to a database table. Since, the proposed method is domain-independent; we are not concerned about the attributes’ name. There are total 8 attributes and some attributes contain more than 40 characters data. Table IV contains the description of the datasets.

TABLE IV
DESCRIPTION OF THE DATASET

Test No.	Description of the dataset
test-1	No duplicates
test-2	800 unique records and 5 exact duplicates per record
test-3	400 unique records and 10 exact duplicates per record
test-4	200 unique records and 20 exact duplicates per record
test-5	100 unique records and 40 exact duplicates per record
test-6	80 unique records and 50 exact duplicates per record
test-7	40 unique records and 100 exact duplicates per record

TABLE V
REQUIRED TIME WITH VARYING THE NUMBER OF DUPLICATES

Test No.	Proposed algorithm (sec)	Scheme-1 (sec)	Scheme-2 (sec)
test-1	**50**	490	522
test-2	**300**	460	472
test-3	**450**	**450**	470
test-4	480	**435**	464
test-5	530	**440**	460
test-6	550	**450**	**450**
test-7	670	470	**440**

TABLE VI
RECALL AND PRECISION FOR THE ALGORITHMS

Thresh old	Proposed algorithm			Scheme-1			Scheme-2		
	R	P	F1	R	P	F1	R	P	F1
0.1	1	**0.15**	.26	1	0.11	.2	1	0.11	.2
0.2	1	**0.15**	.26	1	0.11	.2	1	0.11	.2
0.3	1	**0.15**	.26	1	0.12	.21	1	0.11	.2
0.4	0.95	**0.59**	.73	1	0.36	.53	1	0.25	.4
0.5	**0.9**	1	.95	0.71	1	.83	0.77	1	.87
0.6	**0.9**	1	.95	0.52	1	.68	0.52	1	.68
0.7	**0.89**	1	.94	0.51	1	.67	0.51	1	.67
0.8	**0.84**	1	.91	0.51	1	.67	0.49	1	.66
0.9	**0.67**	1	.8	0.49	1	.66	0.4	1	.57
1	**0.67**	1	.8	0.46	1	.63	0.33	1	.5

Table V contains the required time. From Table V, it can be observed that the performance of our proposed algorithm is better than that of the other two algorithms when the number of duplicates per record is realistic, whereas the other two algorithms require more time when the number of duplicates per records is fewer. The reason is that they use a priority queue. In a priority queue, when the number of unique records is more, additional comparisons per records are required to find a match. If we also make the similar assumption like [3], there will be a maximum of 20 duplicate records per unique records. In that case, the test category falls from test-1 to test-4. In test-4, we require slightly more time. This is because, to improve the efficiency, we are using a modified form of the transitivity rule, which requires additional work than the simple transitivity rule.

It is usual that in a real-world dataset the scenario after test-4 will rarely occur. On the other-hand, in a real-world dataset, it is unlikely that every record will have a fixed number of duplicate records. When two records are the exact duplicates, our proposed algorithm needs to perform maximum number of operations. Hence, we used exact duplicates for time comparison.

Table VI shows the values of Recall (R), Precision (P) and F1 metric for the proposed method and the other two methods. It should be mentioned that, R = (number of true duplicates detected / the number of total duplicates in the dataset), P = (number of true duplicates detected / the total number of duplicates detected). F1 Metric [27] gives equal weight to both of them and is computed as

$$F1 = (2*R*P)/(R+P)$$

For comparing the Recall and Precision, we use a dataset, which contains 3900 unique records with a known number of duplicate records. From table VI, we can find that if the threshold is between 0.1 and 0.4, our Precision is better, whereas if the threshold is between 0.5 and 1, our Recall is better than the other two algorithms. For better understanding of the readers, these comparisons are also visually presented separately in Fig. 3, Fig. 4 and Fig. 5. This claim can also be justified by the values of F1 metric presented in Fig. 6.

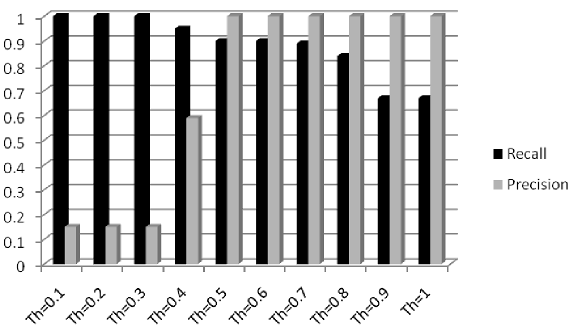

Figure 3. Recall and precision for the proposed algorithm.

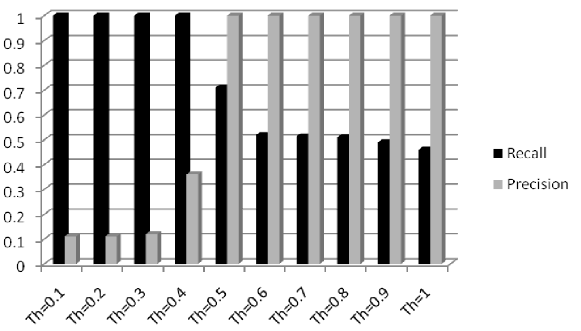

Figure 4. Recall and Precision for Scheme-1(Independent de-duplication in data cleaning).

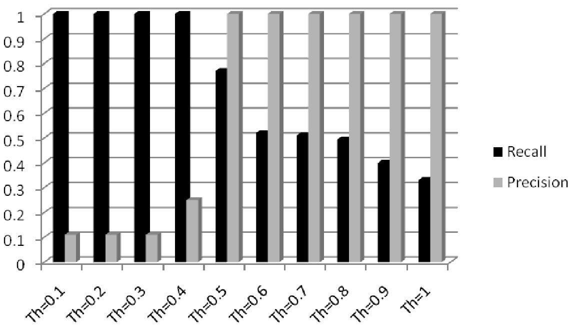

Figure 5. Recall and Precision for Scheme-2 (An efficient domain independent algorithm for detecting approximately duplicate database records).

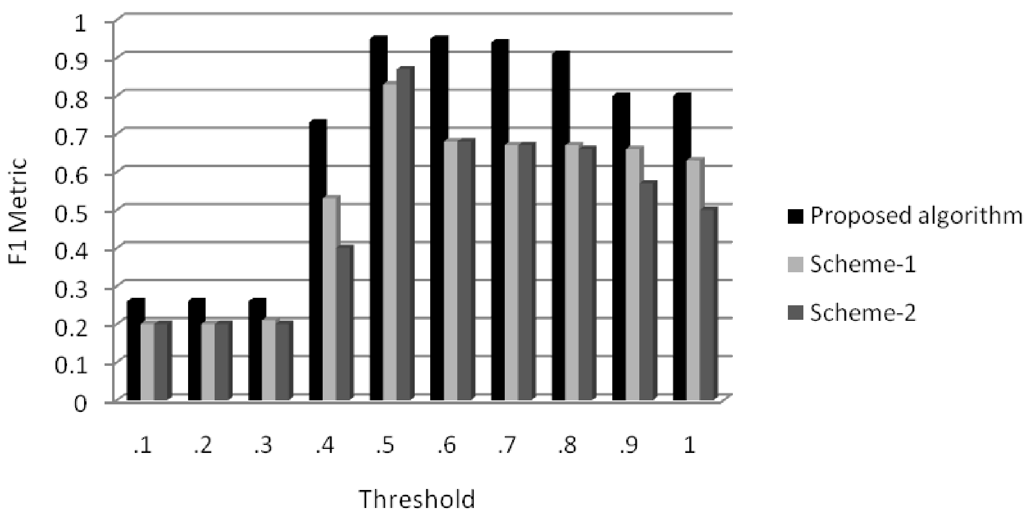

Figure 6. Performance using F1 metric

Efficiency of duplicate detection algorithms depends on the efficiency of the sorting method, as well as the efficiency of the similar-string matching algorithm. From the experimental results, it is clear that the proposed sorting method and the similar-string matching algorithm are more efficient than the other two algorithms for detecting similar-duplicates.

The values in the Table VI also justify this. From the table, it can be found that when Precision is the same, our sorting method and the modified string-matching algorithm is able to detect more duplicates than the other algorithms. At the threshold value of 0.4, our Recall is slightly lower than the other two algorithms. However, when Recall is the same, our Precision is always better than the other two algorithms. Considering both Recall and Precision, the overall efficiency of our algorithm is better. In short, our proposed DIT algorithm plays an important role for making the precision better than the other algorithms. Also, it reduces the number of false positives.

VI. Limitations and Future Works

Near-duplicates are those entities, which are identical from their sense/meanings, but not from their appearances. For example, “Master” and “Teacher” are neither identical nor similar in a character-by-character sense, but they are near-duplicates considering the fact that their meanings are almost the same in some cases. The string-matching algorithm we proposed in this paper, is not capable of detecting such near-duplicates. To detect such type of near-duplicates, the meanings of the related strings should be considered. Detection of near-duplicates can be simply achieved by replacing the string-matching algorithm with the one which is capable of detecting such type of duplicates. It will not bring any change in the proposed DIT or DIDD algorithm. Some authors use near-duplicate to mean similar/approximate duplicate, but because of the difference between the words *near* and *similar*, we do not agree to use *near* and *similar* synonymously.

In the proposed algorithms, threshold is the amount of similarity that determines the difference between two strings in order to be declared as duplicates. It can be any value in the range between 0 and 1.0. In a particular domain, the users can use threshold value of 0.5 (for example) to declare two strings as duplicates, when their resultant similarity is equal to or above 50%. For the same domain, if the users consider that the error is relatively low, they can use threshold value of 0.95 (for example) to declare two strings as duplicates, when their resultant similarity is equal to or above 95%. On the other hand, for two different domains, the users are free to use the same threshold 0.9 (for example), if they want to declare two strings as duplicates; where their resultant similarity is equal to or above 90%. It is possible to use different thresholds for the same domain or the same threshold for different domains depending on the range of errors in data. So the determination of threshold does not depend on domain, it depends on the users.

We used a window-based system for limiting the number of comparisons made with the nearby records, but a priority queue based system or other methods can also be used to test whether any further improvement can be made using all/any of the proposed techniques.

In this paper, we did not consider the problem of missing values or limiting the number of attributes, because these are usually handled before the duplicate detection phase. We also gave the same importance for each attribute during the calculation of similarity between records. In future, we hope to improve the sorting method to bring the similar-records closer and to set the field weights in some automatic way.

VII. Conclusions

This paper presents a domain-independent similar-duplicate detection algorithm for data cleaning in data mining. The efficiency of any duplicate detection algorithm depends heavily on the efficiency of sorting methods. The sorting method proposed in this paper is a novel technique for bringing the similar-records closer and it can sort the records efficiently. The proposed algorithm also explores the idea to make the duplicate detection algorithms faster and more efficient for real-life data. A domain-independent transitivity rule is also proposed to reduce the negative aspect of the transitivity rule for domain-independent cases. This paper also outlines an algorithm to implement all the proposed methods together. All the techniques, we have proposed in this paper can be applied individually or with other existing methods.

ACKNOWLEDGMENT

The authors wish to thank Dr. N. H. Siddique of University of Ulster for his feedback. His comments helped us to make this paper more readable.

REFERENCES

[1] Data Mining. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Data_mining. Accessed on: May 2009.

[2] DMS Tutorial – CRISP Process. http://dms.irb.hr/tutorial/dm_proces.php, Accessed on: May 2009.

[3] A. E. Monge and C. Elkan. "An efficient domain independent algorithm for detecting approximately duplicate database records," In *Proc. SIGMOD Workshop on Data Mining and Knowledge Discovery,* Arizona, pp. 23-29, May 1997.

[4] M. L. Lee, T. W. Ling and W. L. Low "IntelliClean: A knowledge-based intelligent data cleaner," In *Proc. Sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, Boston, USA, pp. 290-294, August 2000.

[5] M. A. Hernandez and S. J. Stolfo, "Real-world data is dirty: Data cleansing and the Merge/Purge problem," *Data Mining and Knowledge Discovery,* vol. 2, no. 1, pp. 9-37, 1998.

[6] M. L. Lee, H. Lu, T. W. Ling and Y. T. Ko, "Cleansing data for mining and warehousing," In *Proc. 10th International Conference on Database and Expert Systems Applications (*DEXA), Florence, Italy, pp. 751 – 760, September 1999.

[7] M. Hernandez and S. Stolfo, "The Merge/Purge problem for large databases," In *Proc. ACM SIGMOD International Conference on Management of Data*, San Jose, California, pp. 127 – 138, May 1995.

[8] A. Udechukwu and C. Ezeife, K. Barker, "Independent de-duplication in data cleaning," *Journal of Information and Organizational Sciences,* vol. 29, no. 2, pp. 53-68, 2005.

[9] R. Ananthakrishna, S. Chaudhuri and V. Ganti, "Eliminating fuzzy duplicates in data warehouses," In *Proc. 28th VLDB Conference, Hong Kong, China,* pp. 586-597, 2002.

[10] Y. R. Wang, S. E. Madnick, and D. C. Horton, "Inter-database instance identification in composite information systems," In *Proc. Twenty-Second Annual Hawaii International Conference on System Sciences,* Kailua-Kona, pp. 677–684, January 1989.

[11] J. A. Hylton. *Identifying and Merging Related Bibliographic Records.* Master of Engineering Thesis, Department of EECS, Massachusetts Institute of Technology, Cambridge, MA June 1996.

[12] W. E. Winkler, *Advanced Methods for Record Linkage* Technical Report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC, 1994.

[13] M. Weis, F. Naumann, U. Jehle, J. Lufter and H. Schuster, "Industry-scale duplicate detection," In *Proc. International Conference on Very Large Databases,* Auckland, NZ*,* vol. 1, no. 2, pp. 1253-1264, 2008.

[14] A. Monge, "Matching algorithms within a duplicate detection system," *IEEE Data Engineering Bulletin,* vol. 23, no. 4, pp.14-20, 2000.

[15] M. Weis and F. Naumann, "DogmatiX tracks down duplicates in XML," In *Proc. ACM SIGMOD International Conference on Management of Data,* Baltimore, MD, pp. 431-442, 2005.

[16] P. A. V. Hall and G. R. Dowling, "Approximate string matching," *ACM Computing Surveys,* vol. 12, no. 4, pp. 381– 402, 1980.

[17] R. S. Boyer and J. S. Moore, "A fast string-searching algorithm," *Communications of the ACM,* vol. 20, no. 10, pp. 762–772, 1977.

[18] M. W. Du and S. C. Chang, "Approach to designing very fast approximate string matching algorithms," *IEEE Transactions on Knowledge and Data Engineering,* vol. 6, pp. 620–633, August 1994.

[19] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology,* vol. 147, pp. 195–197, 1981.

[20] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics – Doklady10,* vol. 10, pp. 707–710, 1966.

[21] J. Peterson, "Computer programs for detecting and correcting spelling errors," *Communications of the ACM,* vol. 23, no. 12, pp. 676–687, 1980.

[22] A. Monge, *Adaptive Detection of Approximately Duplicate Database Records and The Database Integration Approach to Information Discovery.* Ph.D. Thesis, Dept. of Comp. Sci. and Eng., Univ. of California, San Diego, 1997.

[23] A. K. Elmagarmid, P. G. Ipeirotis and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Transaction on Knowledge and Data Engineering,* vol. 19, no. 1, pp. 1-16, 2007.

[24] Flag Forum – The Forum for discussions on All Things Flag related. http://flags.skalman.nu/flags/bib_main.html, Accessed on: September 2009.

[25] Downloads Home Page – Microsoft Office Online. http://office.microsoft.com/en-us/downloads/, Accessed on: September 2009.

[26] Microsoft Download Center. http://www.microsoft.com/downloads/, Accessed on: September 2009.

[27] H. K. Farsani and M. Nematbakhsh, "A semantic recommendation procedure for electronic product catalog," *International Journal of Applied Mathematics and Computer Sciences,* vol. 3, no. 2, pp. 86-91, 2006.