Coreference detection of low quality objects

Joachim Nielandt[1], Antoon Bronselaer[1] and Guy De Tré[1]

Ghent University, Belgium
Document, Database and Content Management
{joachim.nielandt,antoon.bronselaer,guy.detre}@telin.ugent.be

Abstract. The problem of record linkage is a widely studied problem that aims to identify coreferent (i.e. duplicate) data in a structured data source. As indicated by Winkler, a solution to the record linkage problem is only possible if the error rate is sufficiently low. In other words, in order to succesfully deduplicate a database, the objects in the database must be of sufficient quality. However, this assumption is not always feasible. In this paper, it is investigated how merging of low quality objects into one high quality object can improve the process of record linkage. This general idea is illustrated in the context of strings comparison, where strings of low quality (i.e. with a high typographical error rate) are merged into a string of high quality by using an n-dimensional Levenshtein distance matrix and compute the optimal alignment between the dirty strings. Results are presented and possible refinements are proposed.

Keywords: object merging, string merging, error reduction

1 Introduction

In today's age, handling large amounts of data is becoming more and more important. Whether we are talking about databases, raw text files, data transmissions or images, they are all stored and transmitted in massive quantities. With the rise of *Web 2.0* a lot of this data is becoming user-generated which is less subject to quality control. Due to this fact we are faced with an abundance of errors and coreferent (i.e. duplicate) information. The elimination of such coreferent information is called *record linkage* and it is pointed out by Winkler [9] that this is only possible if the data in question meet some minimal constraints. One of these constraints is that "*the amount of (typographical) errors should be low*". However, in practical situations, this constraint is not always feasible. As an example, consider the case of person identification based on biometrical data. The collection of such biometrical data (for example for pictures) can be a process that suffers from a high error rate. Indeed, if we are provided with an image of low quality, the extraction of features is guaranteed to be of low quality too. However, it is a low cost operation to collect several images of the same person. This means that we can easily be provided with a collection of pictures (i.e. objects) that we know to be coreferent. If we succeed to *merge* the extracted features of this collection into features of high quality, the process of

record linkage becomes possible. In this paper we use single strings as an example to illustrate this reasoning, in the sense that we take a string and contaminate it with random operations to generate a set of *dirty* duplicates. We investigate how we can combine the knowledge of this duplicate information into a cleaner (if not perfect) version.

In Section 2, a couple of concepts are explained that are used in the rest of the paper. In Section 3, the problem of record linkage and the Winkler constraints are given in depth. In Section 4, we propose a solution to solve the merging problem for strings while detailing some results and comparing different settings. Future work is discussed in Section 5, regarding improvements on the current solution and different approaches that could be used, while in Section 6 we offer the conclusion of the paper.

2 Preliminaries

In this section some concepts will be explained that are necessary to comprehend the rest of this paper. First we will talk about how we compute distances (and, accompanying those, alignments) between two character strings. After that, we explain how we extend this method to n strings.

2.1 Levenshtein distance and alignment

Modifying a string can be done in different ways. When considering the Levenshtein edit distance [6] there are three possible operations: substitutions (one character morphs into another), insertions (a character is inserted) and deletions (a character is deleted). When we consider two strings s_1 and s_2 of respective lengths m and n we can compute the minimal amount of these operations we need to convert s_1 into s_2.

The standard Levenshtein edit distance algorithm constructs an $n \times m$ matrix d in which the minimal distance can be read in cell $d[n, m]$. We can also perform a traceback through this distance matrix to determine which path leads to the optimal result. Using this traceback we can build an alignment between s_1 and s_2 as shown in Example 1 (mentioned indices refer to the example):

- When considering a substitution: nothing changes (e.g. indices 12 and 20),
- When considering a deletion in s_1 we have to make sure to add an *alignment character* at that position (e.g. index 14),
- When considering an insertion in s_1 we have to add an *alignment character* at the same position in s_2 (e.g. index 1).

2.2 N-Dimensional variation

The standard Levenshtein algorithm is the basis for the string alignment we need. However, to accomodate the merging of more than two strings we have

Example 1 Levenshtein alignment for s_1 (top string) and s_2 (bottom string), with character indices underneath

```
arnie morton'*s of 6hicago
*rnie mortow'Qs of chicago
--------------------------
12345678901234567890123456
```

implemented an n-dimensional variant, based on the three-dimensional alignment using the sum-of-pairs score described in [4].

The pseudocode to construct the n-dimensional distancematrix d can be seen in Algorithm 1, where we start by filling out the 0-dimensional cell (situated in $(0, 0, ..., 0)$) and the 1-dimensional cells where only one index is nonzero. Using these we can fill in the 2-dimensional cells (where 2 indices are nonzero), after which we proceed with the rest of the matrix. We use the n-vector di as an index to access d, where $di_{i=1,j=2}$ indicates that the i^{th} element of di has a value of 1 and the j^{th} element has a value of 2, while all the other elements have value 0. If we then use di to access d we note it as $d[di_{i=1,j=2}]$, which will result in the cell that is situated on the first level of the i^{th} dimension, the second level of the j^{th} dimension and the zeroth level of the rest of the dimensions.

The n strings that are aligned with the algorithm are denoted as $s_1, s_2, ..., s_n$, while the k^{th} character of the l^{th} string is denoted by $s_l[k]$. The different penalties that can be assigned are p_{miss}, p_{space} and p_{match} (respectively the penalties to have a character mismatch, to introduce a space in one of the strings and to have a correct match between characters). Within the scope of this paper we use respectively 1, 1 and 0 as the values for these variables.

As a final note on how to understand Algorithm 1: lines 31 to 37 show the construction of $r = \binom{n}{2} - 1$ variables, using induction. There are $r = \sum_{k=0}^{n-1} \binom{n}{k}$ possible combinations of indices to access d, excluding the combination where all the indices are present. For every one of these combinations we take the chosen indices and take their values as they are (as opposed to the standard subtraction by 1). For example, for v_2 the chosen combination was $\{i_n\}$, so only the value of this index remains as it is, whereas the values of all the rest are subtracted by 1. A special case is v_1, where the chosen combination of indices is empty.

Traceback As with the standard Levenshtein algorithm we keep track of which path results in the minimal cost alignment so we can trace back our steps and see how we came to that optimal result. When tracing back through the distancematrix we rebuild all the strings, going from the end to the beginning, resulting in aligned strings $as_i, 1 \leq i \leq n$ with exactly the same length. We start at the end of the matrix: the value in cell $d_{l_1,l_2,...,l_n}$. We know which other cells could have resulted in this value so we pick one of these and see which indices have changed. If an index i_k has changed we know that a step has been made in the related string s_k so we append the relevant character to as_k. For the string /

index combinations that have remained the same we append an alignment character instead. This ensures that, for every step we make in the distance matrix, the aligned strings grow with 1 character, either by appending an alignment character or a character from the source string.

3 Record linkage

As mentioned in the introduction, the problem at hand is that of *record linkage*([3], [2]), where two collections A and B of objects are given. The goal of record linkage is to find couples of objects $(a, b) \in A \times B$ such that a and b are coreferent or duplicate. Winkler [9] states this is only possible when the following five conditions are met:

- The data should contain more than 5 percent of matches,
- The matching pairs should differentiate themselves sufficiently from the other pairs so they can be properly recognized,
- The amount of typographical errors should be low,
- There is sufficient redundancy in the data to compensate for the errors,
- Estimates computed under the assumption of conditional independence result in a good classification. Within the context of this paper, we shall soften the third condition. More specific, the assumption that objects in A and B are of relative high quality will be dropped. Instead, it will be assumed that, for each object in A or B, we can easily collect a collection of coreferent objects. For example, several pictures of the same person, who do not need to be of high quality. We will show that by merging these collections, the problem of low quality (i.e. high error rate) can be solved. For reasons of simplicity, we shall not illustrate our reasoning with pictures, but with strings, because the contamination of strings is easily controllable.

4 Proposal

In this paper we investigate how generic objects can be merged that describe the same entity, while still differing from each other (due to some contamination of the data). We use our running example to illustrate the problem: if we have a restaurant's name (e.g. *arnie morton's of chicago*), it can be contaminated with random substitutions, insertions and deletions:

```
Vrnie morton's of Ycicago
arnWe ZmoPton'sGof chicago
```

These different descriptions of the same entity still contain a lot of data that are similar to each other. This data could thus be used to merge the different contaminated objects to try and recover the original description of the entity.

As a testing platform we focus on strings in particular, which we extracted from the widely used (e.g. [1, 5, 7]) database containing restaurants. This database was authored by Sheila Tejada [8] and can be retrieved freely from the site of the

Algorithm 1 N-Dimensional distance matrix

$d_{0,0,...,0} := 0$
1-dimensional cells
for $i := 1 \rightarrow n$ **do**
  $d[di_{i=1}] := i$
end for

2-dimensional cells
for $iindex := 0 \rightarrow n-2$ **do**
  for $jindex = iindex + 1 \rightarrow n-1$ **do**
    for $i = 0...l_{iindex}$ **do**
      $v_1 := d[di_{iindex=i-1,jindex=j-1}] + p_{miss}$
      $v_2 := d[di_{iindex=i,jindex=j-1}] + 1$
      $v_3 := d[di_{iindex=i-1,jindex=j}] + 1$
      $d[di_{iindex=i,jindex=j}] = min(v_1, v_2, v_3) + ((i_{iindex} + i_{jindex}) * p_{space}$
    end for
  end for
end for

Non-boundary cells
for $i_1 := 1 \rightarrow l_1$ **do**
  for $i_2 := 1 \rightarrow l_2$ **do**
    ...
    for $i_n := 1 \rightarrow l_n$ **do**
      for $iindex := 1 \rightarrow n-1$ **do**
        for $jindex := iindex + 1 \rightarrow n$ **do**
          $c_{iindex,jindex} := p_{miss}$
          if $s_{iindex}[i_{iindex}] = s_{jindex}[i_{jindex}]$ **then**
            $c_{iindex,jindex} := p_{match}$
          end if
          $r = \sum_{k=0}^{n-1} \binom{n}{k}$
          $v_1 := d_{i_1-1,i_2-1,...,i_n-1}$
          $+c_{i_1,i_2} + c_{i_1,i_3} + ... + c_{i_{n-1},i_n}$
          $v_2 := d_{i_1-1,i_2-1,...,i_{n-1}-1,i_n}$
          $+c_{i_1,i_2} + c_{i_1,i_3} + ... + c_{i_{n-2},i_{n-1}} + (n-1) * p_{space}$
          $v_3 := d_{i_1-1,i_2-1,...,i_{n-1},i_n-1}$
          $+c_{i_1,i_2} + c_{i_1,i_3} + ... + c_{i_{n-2},i_n} + (n-1) * p_{space}$
          ...
          $v_{2+n} := d_{i_1-1,i_2-1,...,i_{n-1}-1,i_n-1}$
          $+c_{i_1,i_2} + c_{i_1,i_3} + ... + c_{i_{n-3},i_{n-2}} + (n-1) * p_{space}$
          ...
          $v_r := d_{i_1-1,i_2,...,i_{n-1},i_n} + (n-1) * p_{space}$
          $d_{i_1,i_2,...,i_n} := min(v_1, v_2, ..., v_r)$
        end for
      end for
    end for
  end for
end for

RIDDLE project[1]. Within this set only the restaurants that originated from the website *Fodor's* were used. First tests focus on just comparing the restaurant's names (from hereon described as strings). We use an n-dimensional distance matrix to compute the minimal cost of transforming the strings to each other, thus allowing us to also compute an optimal alignment of all the strings (MSA[2] solution). By aligning the characters that are most likely to describe the same original (in the clean, non-contaminated string) character we can then make a decision regarding which of the contaminated characters is the correct one. This leads to a merged, cleaner version of the string.

4.1 Methodology

Generating data Using the database of restaurant names we generate a couple of datasets:

- Clean dataset: the original list of restaurant names,
- Dirty dataset: the clean dataset, but randomly contaminated with substitution, insertions and deletions,
- Dirty merge dataset: equivalent to the dirty dataset, but now we generate a list of dirty strings for every clean string in the clean dataset.

When randomly contaminating strings we take into account a number of possibilities: every character has a p_{dirty} chance of being transformed. In the case of a transformation we have three possibilities (that sum up to 1): p_{insert}, $p_{deletion}$ and $p_{substitution}$. These different possibilities can be given preference over each other to facilitate further testing.

Matching results In our tests we compare a list of clean data with a list of dirty data (merged or not). One of the constraints on the data is that we know that there has to be a correct mapping from every clean item to the appropriate dirty item and that these items exist.

We compare every clean item to every dirty item, thus creating a list of m^2 couples. Out of these m^2 couples, m are the correct ones. This list of couples is sorted according to a similarity measure $sim_{d,c}$ we calculated, using the dirty and clean string as input. If $d_{c,d}$ is the Levenshtein distance between a clean string s_c and a dirty string s_d and l_c and l_d are the lengths of the clean and dirty string respectively we can write $sim_{d,c}$ as:

$$sim_{d,c} = \frac{d_{c,d}}{max(l_c, l_d)} \tag{1}$$

By comparing the merged dirty strings to the originals we can find out how accurate we can still do an $m - to - m$ mapping, thus giving us a measure of how much correct information we obtain after merging contaminated objects.

[1] RIDDLE - Repository of Information on Duplicate Detection, Record Linkage, and Identity Uncertainty
http://www.cs.utexas.edu/users/ml/riddle/data.html

[2] Multiple sequence alignment

4.2 Examples

In this section we will give a quick merging example. We present a clean string, together with its dirty versions and merged result. We use the same strings that were used in the tests whose results are reported in Section 4.3. The clean string

```
Arnie morton's of chicago
```

was contaminated with random operations which resulted in a list of dirty versions:

```
awnie mortoDn's of nchicago
arnie morton's of 6hicago
rnie mortow'Qs of chicago
arnie Zmorton's of chicagk
```

Using the n-dimensional distance matrix calculated for these strings an optimal alignment was found (we use the character "*" as an alignment character when needed):

```
awnie *mortoDn'*s of nchicago
arnie *morto*n'*s of *6hicago
*rnie *morto*w'Qs of *chicago
arnie Zmorto*n'*s of *chicagk
```

For every index in the strings we check the most occurring character. For index 0 this is a, for index 1 it is r, etc... After merging the aligned strings by constantly choosing the most occurring character for every index we get the clean string as a result: the settings used for contamination and the amount of strings used were sufficient to provide us with a perfect merge that gave us the original string as a result:

```
Arnie morton's of chicago
```

4.3 Results

In this section we present some experimental results that were obtained using the restaurant dataset.

As a first experiment we took the clean, dirty and merged datasets and calculated how precise we could match the clean strings with the dirty and the merged strings respectively. To match the m clean strings with the m dirty strings we calculated the m^2 Levenshtein distances for every combination of two strings. We sorted these in ascending order and then made an $m-to-m$ mapping from clean strings to dirty strings.

For the merged strings we used different dimensions (2, 3 and 4 dirty strings for every clean string). These dirty strings were merged and matched in the same way we did with the single dirty strings. Results are reported in Figure 4.3, where the x-axis denotes the chance for every character to be modified. It

is clear to see that merging two sources lowers the quality of the match, as we cannot make informed decisions for the relevant characters, whereas more than two strings provide more information about a certain character position which makes it possible to improve upon the result obtained by matching a single dirty string with a clean string.

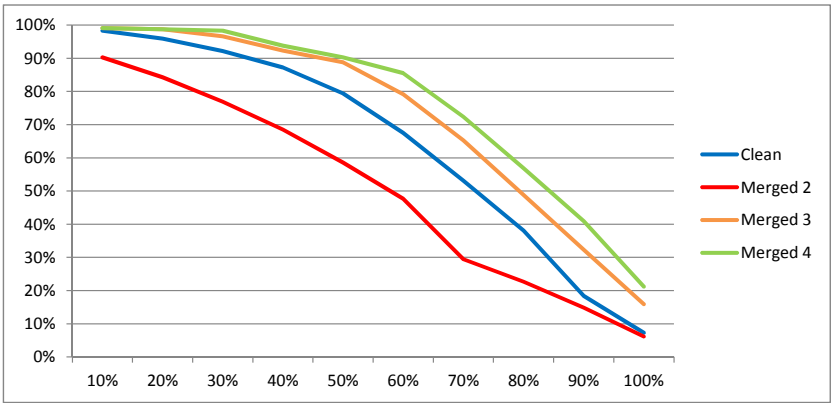

Fig. 1. Percentage of correctly matches strings for given *dirtyChance* settings.

In Figure 4.3 we present the precision - recall curves we generated for the different *dirtyChance* settings (see legend), going from 10% (close to perfect strings) to 100% (completely scrambled strings). The x-axis represents the recall value and the y-axis represents the precision for the given recall. The precision remains high for recall values of up to 0.4, when considering strings that have been made dirty with $dirtyChance \leq 40\%$.

5 Future work

In this section we discuss future steps we will take regarding this research, specifically the current iteration in which we merge strings.

5.1 Improvements

The current workflow makes use of a naïve implementation of an n-dimensional Levenshtein distance matrix. As described in [4] there are a number of improvements to be made when dealing with distances matrices with number of dimensions > 2. These improvements would allow us to perform experiments on a larger scale.

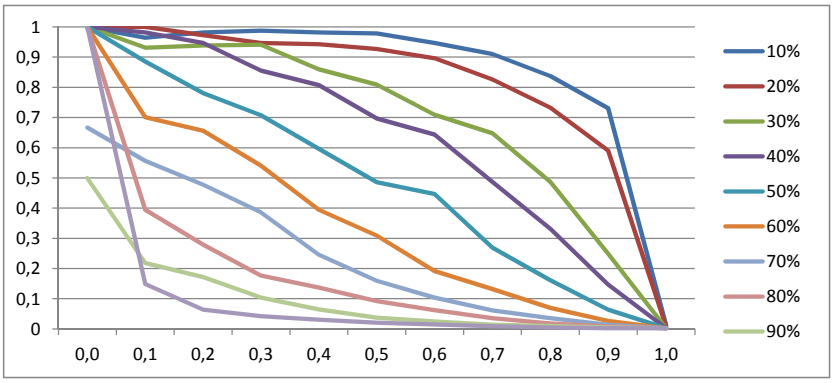

Fig. 2. Precision recall curves for the different *dirtyChance* settings.

5.2 Different alignment methods

Instead of aligning all the strings at once there are ways of aligning them pairwise. The first two strings are aligned and merged, after which this merged version is aligned with the third string and so on. Early experiments with this approach give reasonable results but are not yet error proof. It is a more efficient way of merging strings though, so improvements with regards to the quality of the merging would make the algorithm useful.

5.3 Dataset

This paper focuses on a synthetic database of restaurant names and their randomly modified versions. Future tests will be performed on real-world databases to test the influence of various error rate models.

6 Conclusion

In this paper we presented a way of dealing with incomplete information. More specifically, we investigated the merging of strings that, ideally, should have been identical to each other and of which the quality has been degraded. This can, for example, be applied to the deduplication of a database that is populated with coreferent objects. In that case, deduplication can only be achieved succesfully if the objects are of reasonably good quality. By merging the coreferent objects into a new object of higher quality we make this requirements less immediate, as we can eliminate most of the errors that are present. We showed that, when using the information of three or more of these strings, the merged string resembles the original string better than the individual ones. This initial research on merging strings will allow us to better understand the concept of merging generic coreferent objects.

7 Acknowledgements

We would like to thank FWO (Fonds Wetenschappelijk Onderzoek / Research Foundation Flanders) for their support and making this project possible.

References

1. Bilenko, M., Mooney, R.J.: Learning to combine trained distance metrics for duplicate detection in databases. Tech. rep. (2002)
2. Bronselaer, A., Hallez, A., De Tré, G.: Extensions of fuzzy measures and the sugeno integral for possibilistic truth values. International Journal of Intelligent Systems 24(2), 97–117 (2009)
3. Fellegi, I., Sunter, A.: A theory for record linkage. American Statistical Association Journal 64(328), 1183–1210 (1969)
4. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York, NY, USA (1997)
5. Lehti, P., Fankhauser, P.: Probabilistic Iterative Duplicate Detection. In: Meersman, Robert and Tari, Zahir (ed.) On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, Lecture Notes in Computer Science, vol. 3761, pp. 1225–1242. Springer Berlin / Heidelberg (2005)
6. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Tech. Rep. 8 (1966)
7. Ravikumar, P., Cohen, W.W.: A hierarchical graphical model for record linkage. In: Proceedings of the 20th conference on Uncertainty in artificial intelligence. pp. 454–461. UAI '04, AUAI Press, Arlington, Virginia, United States (2004)
8. Tejada, S., Knoblock, C.A., Minton, S.: Learning object identification rules for information integration. Information Systems 26, 607–633 (2001)
9. Winkler, W.E.: Methods for record linkage and bayesian networks. Tech. rep., Series RRS2002/05, U.S. Bureau of the Census (2002)