The 800 Pound Gorilla in the Corner: Data Integration Assignment 1: Schema Matching

May 2018

GROUP L

Henrik Dichmann	358361
Fiona Wille	376585
Ariane Ziehn	358027

1 General information

Given DB schemas:

Imdb(Id; Name; YearRange; ReleaseDate; Director; Creator; Cast; Duration; RatingValue; ContentRating; Genre; Url; Description)

rotten tomatoes(Id; Name; Year; Release Date; Director; Creator; Actors; Cast; Language; Country; Duration; RatingValue; RatingCount; ReviewCount; Genre; FilmingLocations; Description)

set of actual correspondences :
G = {(Imdb.Name; rt.Name); (Imdb:YearRange; rt:Year); (Imdb:ReleaseDate; rt:"Release Date"); (Imdb:Director; rt:Director); (Imdb:Creator; rt:Creator); (Imdb:Cast; rt:Cast); (Imdb:Duration; rt:Duration); (Imdb:RatingValue; rt:RatingValue); (Imdb:Genre; rt:Genre); (Imdb:Description; rt:Description)

2 Label-Based Schema Matching

Task: Here, we want to find the correspondences between the columns from the two datasets with the help of **only** schema headers.

1. Provide an algorithm. Specify the input, output, similarity function, and time complexity.

2. Implement the algorithm and report the results. Is there any parameter that affects the results?

3. What is the upsides and downsides of this method? When does it work and when not?

To solved the task we used Java in combination with the similarity functions of the SimMetrics library.

2.1 The algorithm, input, output, similarity function and time complexity

- INPUT:
  - source and targetfile as CSV
- OUTPUT:
  - Identified Matches between Source and Target
  - Number of Identified Matches between Source and Target
  - Distinct Attributes of Target
  - Distinct Attributes of Source
  - FINAL SCHEMA
  - Time of calculation in ms

The algorithms read in both file headers. The source schema is taken as baseline and integrates the target into it. The *readSchema*-function cleans the attribute names of both files to lowercase and removes all whitespaces and quotation marks. Further, the *cleanSchema* merges the headers using two for-loops. If both attribute names are exactly the same or both similarity functions, MongeElkan and Levenshtein, are larger than 0.8 and 0.4, the attributes are identified as match. Else they are treated as different values.

Listing 1: Main Function

```
sourceSchema;
targetSchema;
matching{
        readSchema(sourcePath);
        readSchema(targetPath);
        List<String> matches = cleanSchema();

        target.removeAll(matches);
        schema.addAll(target);
    }
```

Listing 2: CleanSchema Function

```java
ArrayList<String> discarded = new ArrayList<>();

        for (int i = 0; i < schema.size(); i++){
            String one = schema.get(i);
            for (int j = 0; j < target.size(); j++){
                String two = target.get(j);

                    if(one.equals(two)){
                        discarded.add(two);
                        discarded.add(one);
                    }
                    else {
                        MongeElkan metricME = new MongeElkan( );
                        Levenshtein metricL = new Levenshtein( );
                        float resultME = metricME
                            .getSimilarity(one, two);
                        float resultL = metricL
                            .getSimilarity(one, two);

                        if (resultME > 0.8 && resultL > 0.4) {
                            discarded.add(two);
                            discarded.add(one);
                        }
                    }
            }
        }
```

2.2 Results of Label-Based Schema Matching

- Identified Matches between Source and Target :[id, id, name, name, year, yearrange, releasedate, releasedate, director, director, creator, creator, cast, cast, duration, duration, ratingvalue, ratingvalue, genre, genre, description, description]
- Number of Identified Matches between Source and Target : 11
- Distinct Attributes of Target :[actors, language, country, ratingcount, reviewcount, filminglocations]
- Distinct Attributes of Source :[contentrating, url]
- FINAL SCHEMA : [id, name, yearrange, releasedate, director, creator, cast, duration, ratingvalue, contentrating, genre, url, description, actors, language, country,

ratingcount, reviewcount, filminglocations]

$$\textbf{precision} = \frac{\text{Number of the discovered correspondences that are in G}}{\text{Number of all the discovered correspondences}} = \frac{10}{11}$$

$$\textbf{recall} = \frac{\text{Number of the discovered correspondences that are in G}}{\text{Number of all the actual correspondences} = 10} = \frac{10}{10}$$

- Time : 63 ms

2.3 Pros and Cons of Label-Based Schema Matching

PROS	**CONS**
Very fast	Entire column represent by one word
Clear and intuitive	Semantic sensitive
Simple	No quality comparison of the data

Label based matching works extremely well if the attribute naming is made with similar semantic. As no instances are known, the problem in this case is the column "id" as each DB commonly contains its own unique identifiers. Similar problem may appear for synonym and homonym naming of attributes. As the instances are not considered at all, another disadvantage is that the quality of both columns is not considered at all.

3 Instance-Based Schema Matching

Task: Here, we want to find the correspondences between the columns from the two datasets with the help of **only** data values.

1. Provide an algorithm. Specify the input, output, similarity function, and time complexity.
2. Implement the algorithm and report the results. Is there any parameter that affects the results?
3. What is the upsides and downsides of this method? When does it work and when not?

To be able to use the two provided datasets we had to preprocess the datasets. Two steps were needed. The first one, was to replace all semicolons with $% and the second step was to replace all separation commas with semicolons.

3.1 The algorithm, input, output, similarity function and time complexity

- INPUT:

  - Source and target file as preprocessed CSV
  - 10 % random reservoir samples of original as additional input files (created with the command-line tool subsample)

- OUTPUT:
  - List of correspondences

Listing 3: Cleaning of dataset

```java
String attribute = split[counter].trim().toLowerCase()
    .replace('"', ' ').replaceAll(" ", "")
    .replaceAll("_", "").replaceAll("-","");

if(!attribute.equals("") && !attribute.equals(" ")) {
    if(attribute.contains(",")){
        String[] split2 = attribute.split(",");}
```

We applied a two phase algorithm: in the first phase, we analyze the whole original input files (imdb.csv and rotten_tomatoes.csv) consisting of 6408 and 7390 lines, respectively. Therefore, we preprocess the input, such that all letters are converted to lower case letters and commas are used as word separators.

The strings are then stored as keys in a hashmap together with the number of the column they were extracted from as values. The keys of this initial hashmap are then compared to the second likewise created hashmap ("rotten_tomatoes.csv") and if an exact match is found, it is written to another hashmap, with the column of the imdb hashmap as key and another hashmap as value containing the string found in the rotten_tomatoes hashmap as key and the counter of how often it was found in this column as value. If exact matches were found in more than one column a threshold of 400 matches was applied. If exceeded, the corresponding columns are again saved to a hashmap and regarded as probable matches. Moreover, we sort out the identified column matches to reduce the input for the second phase even more.

For the second phase we use the reduced sample, since the time complexity of this algorithm is $O((x * y)^2)$ (four encapsulated loops in total) where x and y are the input sizes of the two different datasets. Moreover, size of the compared strings influences the processing time to a great extent (due to the string comparisons).

This algorithm takes the first attribute of the first column of the first input file as input and compares it to every attribute of the second input file. The comparison is done by applying the Monge Elkan and the Levenshtein similarity function that calculates a similarity score for two strings (1 would be a perfect match, 0 denotes no match at all). This iteration is done until every possible similarity score is calculated and saved to a hashmap alongside the corresponding column of the attribute. After a full iteration over one column, the average similarity score for one column with another of the other input file is calculated. We apply a threshold of 0.21 for Monge Elkan and 0.15 for Levenshtein distance to identify matches. This might appear low, but since we only apply little preprocessing, there are still some disturbances like hyphens that reduce the calculated similarity.

As a result of the first phase we got five matches ({0={1=728}, 1={1=1350}, 4={4=810}, 5={5=563}, 12={16=9}}). Those five matches are already filtered out to shorten the time needed for the second phase:

[Column Imdb] = {[Column Rotten_Tomatoes]= [Counter]}	Actual correspondence
0 ={1=728}	False
1 ={1=1350}	Right
2 ={2=0}	Right
3 ={3=0}	Right
4 ={4=810}	Right
5 ={5=563}	Right
6 ={6=0}	Right
11 ={8=0}	False
12={16=9}	Right

Table 1: Output (final matches), 609s

3.2 Results of Instance-Based Schema Matching

$$\textbf{precision} = \frac{\text{Number of the discovered correspondences that are in G}}{\text{Number of all the discovered correspondences}} = \frac{7}{9}$$

$$\textbf{recall} = \frac{\text{Number of the discovered correspondences that are in G}}{\text{Number of all the actual correspondences} = 10} = \frac{7}{10}$$

- Time: 353s

3.3 Pros and Cons of Instance-Based Schema Matching

PROS	**CONS**
Versatile	Extremely time consuming
	Knowledge of dataset is necessary for choosing appropriate similarity measure