Duplicate record detection in a data set

Alexander Hamedinger
Data Wrangling With MongoDB
OTH Regensburg
Regensburg, Germany
alex1.hamedinger@st.oth-regensburg.de

***Abstract*—This report describes how to be searched for duplicates in the test data set "restaurant data". The task here was to identify as many correct duplicates as possible and to keep the number of incorrectly identified duplicates as low as possible. Finally, the results will be compared to the gold standard (all specified duplicates), various key figures will be evaluated and the results will be discussed . This was done using the programming language python and a MongoDB database. The source code can be viewed on github at the following repository: AlexHamedinger/DMDB.**

I. Introduction and data structure

The restaurant data record and a list of all gold-duplicates is available in tab separated value format. The restaurants file has six columns: id, name, address, city, phone and type, while the gold standard file has two columns, id_1 and id_2, which represent the ids of a duplicate pair. There are two python files, one with the main function and one with several other provided functions.

II. Step 1: Load the data into the database

A. Load data from .tsv file (Fig. 1)

For a better handling and to make sure that they are stored persistently, the data records should first be loaded into a MongoDB database. To do this, the Python program first calls the function get_data which reads data from a .tsv file and loads it into a list.

B. Create connection to a database collection (Fig. 2)

Now a connection to the MongoDB database must be established. For this purpose, the get_collection function is called. It imports MongoClient from pymongo and passes it a connection string, previously read from the database.conf file. The name of the collection to connect to is then passed to the client. If this collection does not yet exist, it is automatically created.

C. Store data in the database (Fig. 3)

In the third called function, store_in_database, the data from the passed data-list is now to be stored in the just created collection. The column id is renamed to _id so that the database automatically recognizes it as an Id column. Since MongoDB stores its data in field-value pairs, so-called documents, the list must first be converted into dictionaries, as this is the python equivalent of a document. The dictionaries are created by iterating through the data-list. The first line of the data-list, the field names, and a current line, the values, always form one dictionary. The dictionaries can now be stored as documents in the database.

```
def get_data(file_adress, my_delimiter):
    data = []
    with open(file_adress, 'r') as csvFile:
        reader = csv.reader(csvFile,
                 delimiter = my_delimiter)
        for row in reader:
            data.append(row)
    return data
```

Fig. 1. Shows the get_data function

```
def get_collection(db_name, collection_name):
    from pymongo import MongoClient
    connection_string = ""
    with open("database.conf") as config:
        for line in config:
            connection_string = line.strip()

    client = pymongo.MongoClient(connection_string)
    db = client[db_name]
    db_collection = db[collection_name]
    return db_collection
```

Fig. 2. Shows the get_collection function

```
def store_in_database(db_collection, data):
    db_collection.drop()
    #if "id" exists, it becomes "_id"
    if(data[0][0] == "id"):
        data[0][0] = "_id"
    dataIterator = iter(data)

    #reading the first row (contains header)
    header = next(dataIterator)

    #writing every row of the list to MongoDB
    for row in dataIterator:
        #building a dictionary
        new_document = (dict(zip(header, row)))
        #storing the dictionary in the DB
        db_collection.insert_one(new_document)
```

Fig. 3. Shows the store_in_database function

III. Step 2: Clean the data

The next step is to clean the data that is now in the database. Since the fields refer to different types of data and meet different requirements, each field must be checked and cleaned individually.

A. Name

The name field has been left unchanged, as a simple cleanup would not produce positive results, but a more accurate cleanup of the data would require too much effort for the scope of this project. As an example the typewritter distance could be implemented (Hier Verweis einfügen). In order to see the negative effects we make an example, removing all closing words in brackets. Let's have a look at the records "87 - lespinasse", "88 - lespinasse (new york city)", "689 - michael's (los angeles)" and "776 - michael's (las vegas)". As you can guess, the pair 87 - 88 is a real duplicate, which would be recognized by removing the closing parenthesis. However, the pair 689 - 776 would be a type 1 error (false positive) (Hier Verweis einfügen), which means that a false duplicate would have been detected.

B. Address

The Address field usually begins with a leading postcode followed by a street name, for example "747 - 840 second ave". This field is normalized in four steps (Fig. 4).

- First the address string is split into the number part and the street part. Since an address does not always have to contain a postcode, but can also simply be "central park west", the house number part is created and set to 0 if it does not exist.
- In the second step the street name is edited. The street name sometimes starts with a leading "e.", "s.", "w." or "n.", which stands for east, south, west or north. This letters must be removed, because it can result in non-recognition of duplicates, as in the example "21 - n. hillburst ave." and "22 - hillburst ave.", which are in fact duplicates.
- When the leading direction letter is removed, street names that are numbers, for example "203 - 4th st.", are converted to their written form (fourth st.).
- The fourth and last step is to create a key from the postcode and the first five letters of the street name, separated by a dash.

C. City

When looking through the city field, it is quite noticeable that often different names were registered for the same city. For example LA and Hollywood stand for Los Angeles or New York City stands for New York. The field is scanned for several of these paraphrases and then combined under one main term (Fig. 5).

D. Phone

With this fields there are no difficulties to unify them. There are only two different ways of how phone numbers are stored. Either it starts with three digits followed by a "-" or it starts with three digits followed by a "/". The field has been purged to variant one (Fig. 6).

E. Type

For the Type field, again two steps are taken. First, all closing parentheses are removed, as already considered for the name field. Unlike the name field, this is unproblematic here, since the brackets only contain various adjectives, for example "new" or "classic". In the second step, some values are replaced, as in the city field, because they mean the same thing. Here, for example, "bbq" is replaced by "american" or "chinese" by "asian".

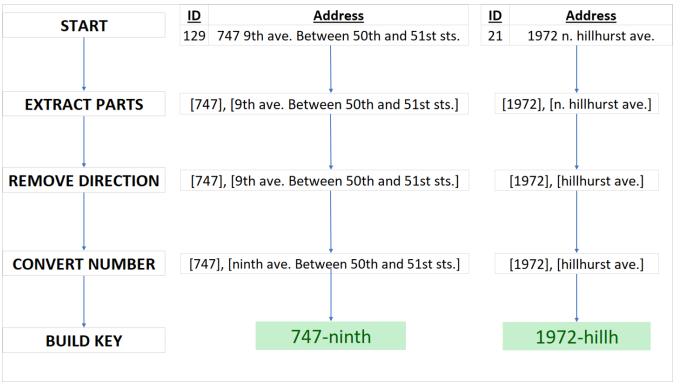

Fig. 4. Shows the way the address field is getting unified

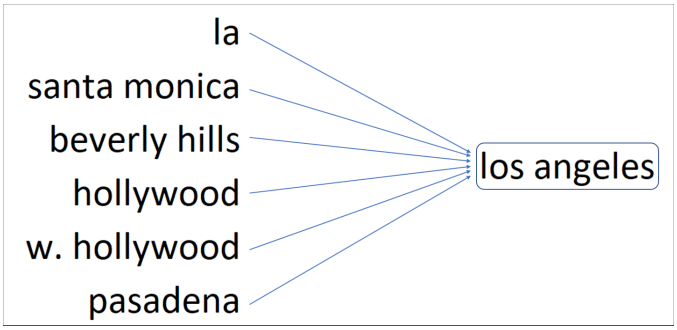

Fig. 5. Shows the way the city field is getting unified

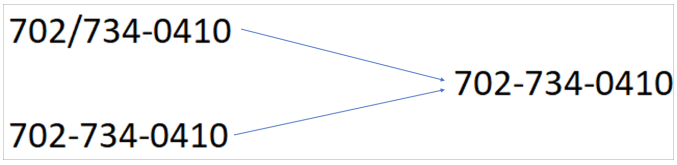

Fig. 6. Shows the way the phone field is getting unified

IV. Step 3: Find duplicates

After the records have been cleaned properly, the duplicate search can now be started.

For this purpose, different field combinations, which should be considered duplicates if they are equal, are combined to a list of several field combinations (Fig. 7). Now for all these combinations the data is searched for duplicates. A duplicate is considered as such if all fields of a combination match the other (Fig. 8).

```
[["name", "address", "city"],
 ["address", "phone", "type"],
 ["name", "city", "phone"],
 ["address", "city", "phone"]]
```

Fig. 7. Example for a field combination

id	name	address	city	phone	type	duplicates
21	restaurant katsu	1972-hillh	los angeles	213-665-1891	asian	NO
22	katsu	1972-hillh	los feliz	213-665-1891	asian	
21	restaurant katsu	1972-hillh	los angeles	213-665-1891	asian	NO
22	katsu	1972-hillh	los feliz	213-665-1891	asian	
21	restaurant katsu	1972-hillh	los angeles	213-665-1891	asian	NO
22	katsu	1972-hillh	los feliz	213-665-1891	asian	
21	restaurant katsu	1972-hillh	los angeles	213-665-1891	asian	YES
22	katsu	1972-hillh	los feliz	213-665-1891	asian	

Fig. 8. Illustration of how the duplicate search works

The search technically works via a database query. For each field to be checked, a pipeline is executed (Fig. 10). The restaurant data are first grouped by the field name, sumed up the number of matching entries and then all entries with a sum greater than one are output.

Next, for each entry of this result the matching documents are loaded from the database. The IDs of these (duplicate) documents are stored as pairs in a list where the the lower ID always comes first. This avoids duplicate entries for the same pair (Reference to Searching for Gold essay). The classified duplicates are stored in the database in the classifier collection.

Then the next field combination passed is examined and the search for the corresponding duplicates is started again. The duplicates found are compared with the duplicates previously found (Fig. 9) and if one or more of the duplicates are not found yet, the classifier collection is updated.

old duplicates (3)		recent duplicates (3)	
id_1	id_2	id_1	id_2
21	22	57	58
57	58	61	62
139	140	139	140

new duplicates (4)	
id_1	id_2
21	22
57	58
61	62
139	140

Fig. 9. The way new duplicates are stored

```
find_multiple_entries(db_collection, field_name):

pipeline = [
            {'$group': {'_id': '$' + field_name,
                        'count': {'$sum': 1}}},
            {'$match': {'count': {'$gt': 1}}},
            {'$sort': {'count': -1,
                       '_id': 1}},
            {'$project': {'_id': 0,
                          field_name: '$_id',
                          'count': 1}}
           ]

return list(db_collection.aggregate(pipeline))
```

Fig. 10. The pipeline that is executed to find multiple entries

V. Step 4: Classification and Indicators

After all passed field combinations have been checked for duplicates and the result has been saved in the database, the compare_to_gold function is called. Here, the pairs are categorized in a first step and then various key figures are determined and displayed in a second step.

A. Classification

First, the duplicates found and the gold duplicates provided are loaded into a list. Now both lists are iterated over and their entries are compared. The data pairs are then sorted into one of the four following categories (Fig. 11).

a) True-Positives (TP): This is a correctly recognized duplicate. They result from the intersection of the gold-standard duplicates and the classifier duplicates, therefore if the pair is in both lists.

b) False-Positives (FP): If a result is false-positive, this means that a duplicate was detected by mistake. That is, when the pair is only in the classifier list.

c) False-Negatives (FN): In case a duplicate remains unnoticed, it is placed in the false negative category. You can tell by the fact that the pair only appears in the gold list.

d) True-Negatives (TN): A true-negative result correctly identified one pair as non-duplicates. Since non-duplicate pairs are not explicitly searched for, this value is calculated from the difference of all non-duplicate pairs N and all incorrectly recognized duplicates (1).

$$TN = N - FP \tag{1}$$

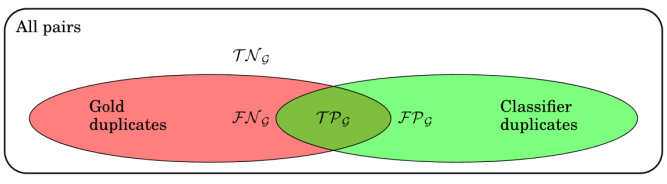

Fig. 11. Categories into which the data is divided

B. Indicators

Once all pairs have been assigned to one of the four categories, it is possible to calculate different key figures.

a) Precision (2)*:* Intutitively, precision is a measure for the correctness of the result or what percentage of result are True-Positives. A precision of 100% means that every pair found is a duplicate from the goldstandard.

b) Recall (3)*:* Recall can also be referred to sensitivity or true positive rate, that means it is a measure for the ratio of how many of the duplicates were detected. A recall of 100% means that all gold-standard duplicates were identified.

c) F-Score (4)*:* A measure that combines precision and recall is the harmonic mean of precision and recall, the F-score. It is approximately the average of the two when they are close.

d) Balanced Accuracy (5)*:* The balanced accuracy can be described as a fair measurement between all four categories, as it includes each of them.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$F = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4}$$

$$bAcc = \frac{Recall + \frac{TN}{TN+FP}}{2} \tag{5}$$

C. End of program

An overview is displayed at the end of the program (Fig. 12). You can see the field combinations which have been checked for duplicates, how many pairs have been classified in which category and of course the calculated key figures. This is done for all entered field combinations.

```
Found 74 duplicates for input ['name', 'address']
Found 81 duplicates for input ['name', 'city']
Found 80 duplicates for input ['name', 'phone']
Found 112 duplicates for input ['address', 'phone']

recognized duplicates: 122

correctly recognized duplicates (TP): 108

incorrectly_recognized_duplicates (FP): 14

unrecognized_duplicates (FN): 4


========================================
Precision: 88.52% (very low)
Recall: 96.43% (high)
F-Score: 92.31% (low)
Balanced Accuracy: 98.21% (very high)
========================================
```

Fig. 12. Example output to the console

VI. Discussion

Last but not least I would like to discuss some results.

For this I have chosen two example field combinations, which I would like to discuss briefly. The first one forms all possible pairs of combinations of three (example), the second one all possible pairs of four (example 2).

As you can see on the console output (Fig. 13), example 1 has a recall of over 97 percent. That means that almost all duplicates could be found. But for the price of a rather low precision level. On closer consideration this is not surprising, because the more duplicates you find, the higher is the probability that there are correct ones. You can also see that the F-Score is not the average of Precision and Recall, but rather approaches the lower value. The Balanced Accuracy of almost 99 percent is very high.

A completely different picture is presented in example 2 (Fig. 14), where the combination of several fields has resulted in a much more accurate hit image, which is reflected in an almost perfect precision. On the other hand, a recall of only three quarters of found duplicates is rather poor, which also makes the F-Score look bad and pulls the balanced accuracy down. An exact statement which of the two field combinations is the better one can hardly be made, since this always has to be determined specifically for the particular use case, and of course individual combinations can be dropped, added or mixed.

```
recognized duplicates: 128
correctly recognized duplicates (TP): 109
incorrectly_recognized_duplicates (FP): 19
unrecognized_duplicates (FN): 3
========================================
Precision: 85.16% (low)
Recall: 97.32% (high)
F-Score: 90.83% (average)
Balanced Accuracy: 98.66% (very high)
========================================
```

Fig. 13. Results for combinations of three different fields

```
recognized duplicates: 88
correctly recognized duplicates (TP): 87
incorrectly_recognized_duplicates (FP): 1
unrecognized_duplicates (FN): 25
========================================
Precision: 98.86% (very high)
Recall: 77.68% (very low)
F-Score: 87.0% (low)
Balanced Accuracy: 88.84% (average)
========================================
```

Fig. 14. Results for combinations of three different fields