Schema Matching Using BabelNet as Background Knowledge

Bilel Arfaoui

224618@studenti.unimore.it

University of Modena and Reggio Emilia

May 2024

Abstract

This article attempts to resolve the schema conflicts between two different datasets with the assistance of the API offered by BabelNet, which will help to find the relationship between label names. The study will comprehend an introduction to BabelNet, how it is structured and what services it provides. After that we will attempt the actual schema resolution by studying the schemas of the datasets and compare the performances between the traditional label and value based techniques and the approach based on the BabelNet API. The findings will shed light on the path of schema matching approach based on the grammatical relationship between labels with the assistance of Background Knowledge.

1 Introduction

This article delves into the exploration of resolving Schema conflicts that arise when integrating two datasets, leveraging the capabilities of the BabelNet[1] API[2]. Our primary objective is to investigate and evaluate the effectiveness of this approach compared to traditional methods that rely solely on label and value matching. We aim to assess various performance metrics, including accuracy, speed, costs, and other pertinent factors, to determine the potential advantages and limitations of utilizing the BabelNet API in Schema conflict resolution. The study will be executed in Google Colab[3] and the programming language of choice for this study will Python[4].

[1] https://babelnet.org
[2] https://babelnet.org/guide
[3] https://colab.research.google.com
[4] https://www.python.org

2 BabelNet

BabelNet is a multilingual semantic network developed at the NLP group of Sapienza University of Rome, it was created by linking Wikipedia[5] with Wordnet[6] the most popular English language lexicon. BabelNet will provide us with the necessary data for our study via the official HTTP API[7], the following API calls will be used:

- getSynsetIds: provides the associated synsets of the given the word/lemma
- getOutgoingEdges: given a synset we will provide the associated synsets and their relationship data

The data in BabelNet is structured as a graph composed of many to many relationships of the following nodes:

- Lemma/Resource: may contain information such as language and POS[8].
- Synset: Node that represents a concept/sense, identified by a unique identifier.

3 Study

disclaimer: all the codes that will be shown here is available in the following GitHub repository https://github.com/NakajimaAkemi/SchemaMatcherBI

Our study will focus on a label based schema matching by using a local graph built on the data fetched from the BabelNet API.

3.0 Considerations

Given the fact that this study is based on an API, the latency will be higher and the number of API calls per day will be limited. Considering this, we will make certain considerations in order to execute our experiments without problems and acceptable times. The issues that have been identified during the study are as follows:

- The number of daily API calls are limited with the free plan
- Automatically generated words are not recognized by BabelNet rendering the matching method ineffective
- During the matching process if the words are not present in the graph, a 0 as result may cause False Negative

[5] https://en.wikipedia.org
[6] https://wordnet.princeton.edu
[7] https://babelnet.org/guide
[8] https://en.wikipedia.org/wiki/Part_of_speech

- Words have multiple meanings, annotations may be needed to avoid False Positives
- The infrastructure on which this study is executed is limited and efficient memory management is advisable
- The available infrastructure in not able to handle the local indices, the use of the API is an alternative, but if possible the first option is advisable

With the purpose of tackling the issues, technical decisions and considerations have been taken such as the following:

- A boolean variable called alpha is defined and set to true, this will stop API calls of words with special characters or number
- A max number of synsets is defined to reduce the number of GetOutgoingEdges calls
- A max number of edges per synsets is set to keep the memory free from useless data
- A fudge factor is defined for the similarity measure, case one of the words is not in the graph, the similarity will be set to the fudge factor instead of 0
- An array of the approved relationship groups between synset, in order to manage the data more efficiently
- Array of language codes to tackle the problem better

3.0.1 Example of use case for annotations

Let’s make some examples with the table Person(Appellation,Address) and Individual(Name,Location).

In the case of the table Person:

- Appellation: is a noun that can refer to an identifying word, a nickname or even indicate a Geographical location where grapes for a whine are grown
- Address: may refer to a geographical point where an organization or person is residing, or may refer to a portion of memory in Disk

In the case of the Table Individual:

- Name: may refer to a identifying word or someone’s reputation
- Location: refers to geographical point or may even refer to the verb “Locating”

Annotations may smooth the process by using the specific synset, indicate the Point Of Speech etc.. reducing the ambiguity. In the case of our tables we can annotate them the following way:

Listing 1-2: Person and Individual tables

Person	
Appellation synset:**bn:00005032n** Pos: **Noun** Language: **English**	Address synset:**bn:00001303n** Pos: **Noun** Language: **English**

Individual	
Name synset:**bn:00056758n** Pos: **Noun** Language: **English**	Location synset:**bn:00051760n** Pos: **Noun** Language: **English**

3.1 Data annotation

The datasets may have ambiguous or automatically generated columns, potentially causing problems during the matching process and/or data fetching phase. To mitigate this problem we will provide annotations to the dataframe in the form of a dictionary.

Listing 3: Annotation function and a call example

```
def add_column_annotations(df,column_annotations):
    for col in df.columns:
        if col in column_annotations:
            df[col].annotations = column_annotations[col]
        else:
            df[col].annotations = {}  # Se non ci sono annotazioni
    return df



S3_annotations = {
        'Age2': {'lemma':'age','language': 'EN'},
        'Age4': {'lemma':'age','language': 'EN'},
        'City1': {'language': 'EN'},
        'Phone1': {'lemma':'phone','language': 'EN'},
        'Full Name':{'language':'EN'},
        'Phone3': {'lemma':'phone','language': 'EN'},
        'Phone4': {'lemma':'phone','language': 'EN'},
        'City3': {'lemma':'city','language': 'EN'},
        'City2': {'lemma':'city','language': 'EN'},
        'rec_id': {'lemma':'id','language': 'EN'}
}

SOURCES['S3']=add_column_annotations(SOURCES['S3'],S3_annotations)
```

For each column it will possible to give an annotation containing additional data such as:

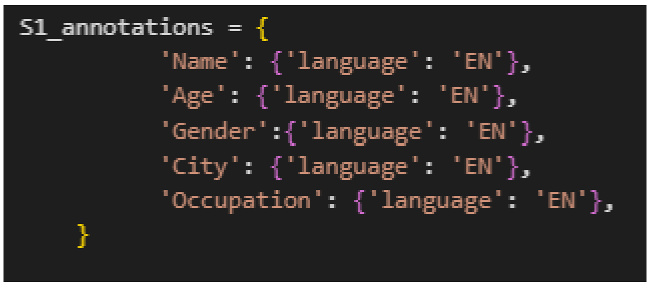
- **Synset**: unique synset code for disambiguation purposes
- **POS**: Part of speech
- **Language**: Target word's language
- **Lemma**: Word that will act as alias if defined

This way we can filter the data fetched from the API and be more precise with the matching and pathfinding in the graph.

3.1.1 DataFrame annotation by Pandas in detail

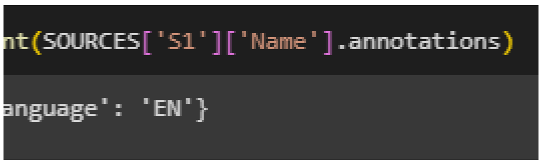
The addition of annotations is basically an addition of attributes to the dataframe, thanks to Pandas that designed DataFrame with the objective of allowing users to add attributes such as annotations to Pandas DataFrames enhances their flexibility and utility for data analysis. It enables customization, metadata storage, improves expressiveness, and aids in analysis and visualization. This flexibility empowers users to enrich and personalize their data, meeting specific analytical needs and facilitating better understanding and interpretation of the data.

Figure 1: Local Graph schema



```
S1_annotations = {
          'Name': {'language': 'EN'},
          'Age': {'language': 'EN'},
          'Gender':{'language': 'EN'},
          'City': {'language': 'EN'},
          'Occupation': {'language': 'EN'},
    }
```

Figure 2: Print of the annotations of the column 'Name'



```
nt(SOURCES['S1']['Name'].annotations)

anguage': 'EN'}
```

Thanks to this feature the annotations are perfectly encapsulated in the Dataframe and do change the overall structure of the data, and are also fetchable easily if present. This is a powerful tool for our experiment allowing us to pass the annotation inside the dataframe avoiding passing a list of annotations and rendering the code more complex.

3.2 Graph building

After the annotation we take the columns of the Datasets and an eventual GlobalSchema (if we're working on a Top Down approach), we take the data on hand and send a request to the API, after that we will add the data to a local graph provided by the library Networkx[9].

Listing 4: Definition of a empty Graph

```
import networkx as nx
V = nx.Graph()
```

The following function is the main function for building our graph, we take the word as input for our API and try to fetch its synsets, if there are no synsets, the word won't be added to the graph since it does not have any linkage and just occupies space without being used and unable to be evaluated. Another case worth mentioning is that if the synset is specified in the annotations, we won't call the API, because we already have the synset locally and we'll limit ourselves with finding the edges afterwards.

Listing 5: API call get synset ids

```
def fetch_synset_ids_for_graph(graph,word,lemma=None,POS=None,language=None):
   if lemma is not None:
      if is_lemma_present_with_synsets(graph, lemma):
        return []
   else:
      if is_lemma_present_with_synsets(graph, word):
        return []

   url = f"https://babelnet.io/v9/getSynsetIds"

   # Parametri della richiesta
   params = {
      'key': API_KEY,  # Inserisci la tua chiave API di BabelNet qui
   }

   if lemma is not None:
     params['lemma']=lemma
   else:
      params['lemma']=word

   if language is not None:
     params['searchLang'] = language
   else:
     params['searchLang'] = languages
```

[9] https://pypi.org/project/networkx/

```
    if POS is not None:
      params['POS']=POS

    response = requests.get(url, params=params)
    synset_ids = response.json()
    if response.status_code != 200:
      synset_ids=[]

    if len(synset_ids)>max_number_of_synsets:
        synset_ids=synset_ids[:max_number_of_synsets]
    return synset_ids
```

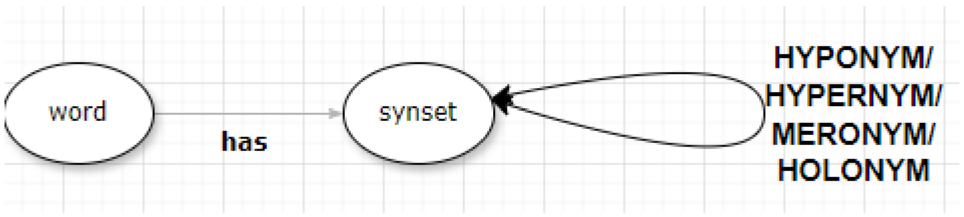
After fetching the synsets we will grab the outgoing edges in order to build relationships between the synsets in our local graph.

Listing 6: API call get the outgoing edges

```
def get_outgoing_edges(synset_id):
    url = f"https://babelnet.io/v9/getOutgoingEdges?id={synset_id}&key={API_KEY}"
    response = requests.get(url)
    if response.status_code != 200:
      return []
    edges = response.json()
    return edges
```

After fetching all the data from BabelNet our graph will be ready for the next phase of pathfinding and similarity match.

Figure 3: Local Graph schema



The graph takes as input all schemas and creates a single connected graph, each column/attribute has its own node.

3.2 Pathfinding and Integration in the matching process

With the local graph ready, the next phase will be to define a similarity measure method, our approach will be based on the distance and kind of relationship is present between the nodes. The similarity measure will vary between 0 and 1 and is calculated in the following way:

- If the words share a synset it means they are synonymous and the result given is 1
- If the synsets of our words share an Hyponym or Hypernym edge the result will be set to 0.8
- If the synsets of our words share an Meronym or Holnym edge the result will be set to 0.5

- If one or both words are not present in the graph, there is no result and will return the value of the fudge factor set by us, this is made in order to reduce the number of False Negatives on the final result.
- In case of no linking between the words, the result will be set to 0

The values returned have been decided with rule of thumb and can be modified for different needs and objectives, Meronym/Holonym cover cases where a label is part of another label such as ‘First Name’ and ‘Full Name’, Hypernym and Hyponym will allow us to associate labels with other labels that have a more generic meaning.

Listing 7: BabelNet similarity measure function

```
def babelnet_sim_score(searchWord,targetWord):
  if not is_node_present(V, searchWord) or not is_node_present(V, searchWord):
     return fudge_factor
   if find_common_nodes(searchWord, targetWord):
     return 1
   else:
     if find_paths_within_steps(searchWord, targetWord, 3,
approved_edges=['HAS','HYPONYM','HYPERNYM']):
       return 0.8
     if find_paths_within_steps(searchWord, targetWord, 3,
approved_edges=['HAS','MERONYM','HOLONYM']):
       return 0.5
   return 0
```

After defining the similarity measure function, it is now necessary to integrate a similarity table on which the similarity can be measured by giving the Pandas[10] Dataframe directly. If the input Dataframes are annotated, the lemma will be considered for the matching instead of the original label.

Listing 8: Annotated Similarity table function

```
def annotated_sim_table(TableA: pd.DataFrame, TableB: pd.DataFrame):
   A_columns = []
   B_columns = []
   for col in TableA.columns:
      if hasattr(TableA[col], 'annotations'):
            annotations=TableA[col].annotations
            lemma = TableA[col].annotations.get('lemma', col)
            if lemma is not None:
               A_columns.append(lemma)
            else:
               A_columns.append(col)
      else:
               A_columns.append(col)

   for col in TableB.columns:
      if hasattr(TableB[col], 'annotations'):
            annotations=TableB[col].annotations
```

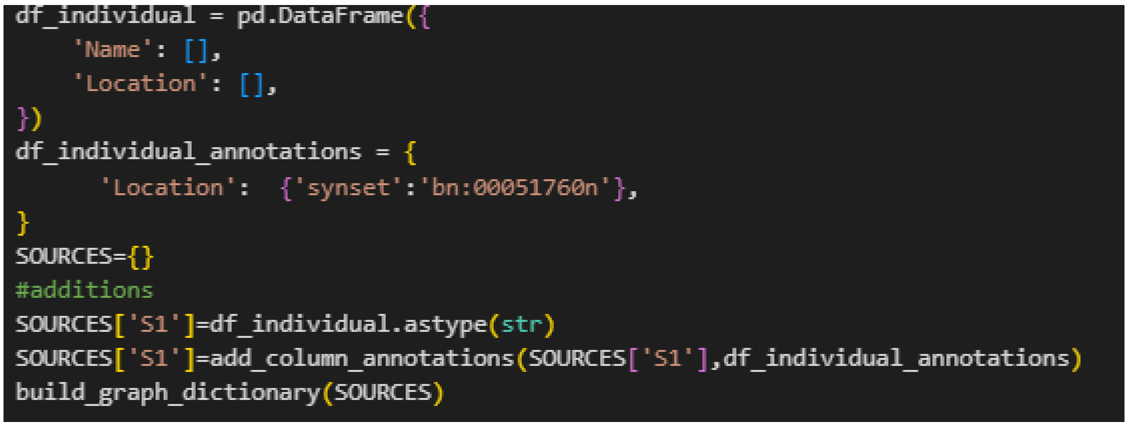
[10] https://pandas.pydata.org

```
            lemma = TableB[col].annotations.get('lemma', col)
            if lemma is not None:
              B_columns.append(lemma)
            else:
              B_columns.append(col)
      else:
              B_columns.append(col)
   A = pd.DataFrame({"A": A_columns})
   B = pd.DataFrame({"B": B_columns})
   S = A.merge(B, how='cross')
   return S
```

3.3 Graph building example

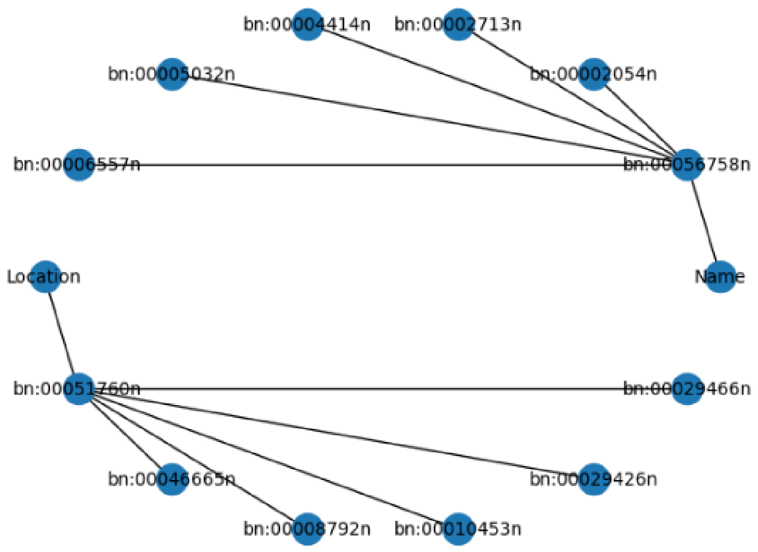
We will take the table Person(Name,Location) to build our graph

Figure 4: Custom made Dataset made for testing



```
df_individual = pd.DataFrame({
    'Name': [],
    'Location': [],
})
df_individual_annotations = {
      'Location':  {'synset':'bn:00051760n'},
}
SOURCES={}
#additions
SOURCES['S1']=df_individual.astype(str)
SOURCES['S1']=add_column_annotations(SOURCES['S1'],df_individual_annotations)
build_graph_dictionary(SOURCES)
```

Figure 5: Graph built from the example



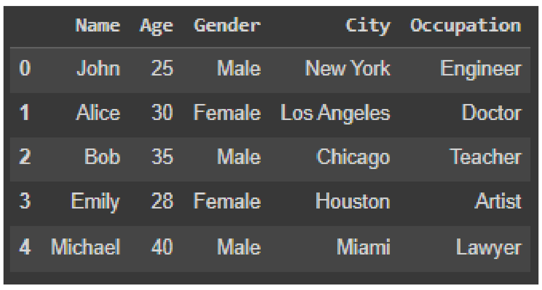
4 Schema matching using BabelNet case study

For this research endeavor, with the purpose to show more representative results, we will use the same datasets for all cases. Our focus will be to study the impact of the babelnet similarity measure over the typical approach, and conclude if it's worth it to apply it in real life use cases.

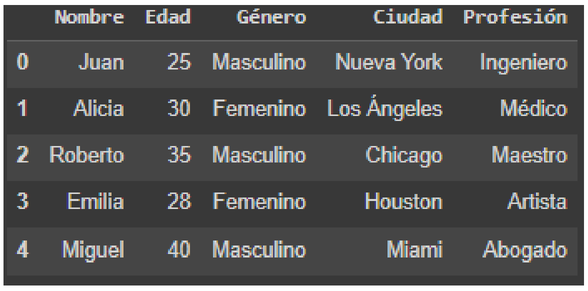
4.1 Babelnet schema matching experiment

For testing purposes we will create 2 datasets in different languages, the first one in english and the second in spanish. We will test if the API works with matching or not. The results are optimal and the only discrepancies from the gold standard are caused by ambiguity, easily fixable with additional annotations. The first experiment will be a Bottom Up schema matching label based strictly through BabelNet, the second one will be a Top Down schema matching attempt with a GlobalSchema with labels in French. We will also attempt to do schema matching with the typical methods and compare the results.

Figure 6-7: Custom made Dataset made for testing

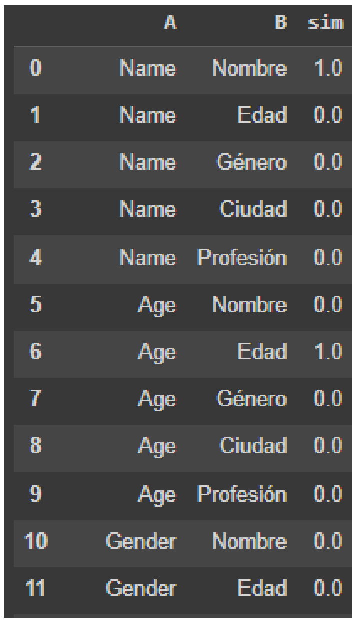


	Name	Age	Gender	City	Occupation
0	John	25	Male	New York	Engineer
1	Alice	30	Female	Los Angeles	Doctor
2	Bob	35	Male	Chicago	Teacher
3	Emily	28	Female	Houston	Artist
4	Michael	40	Male	Miami	Lawyer

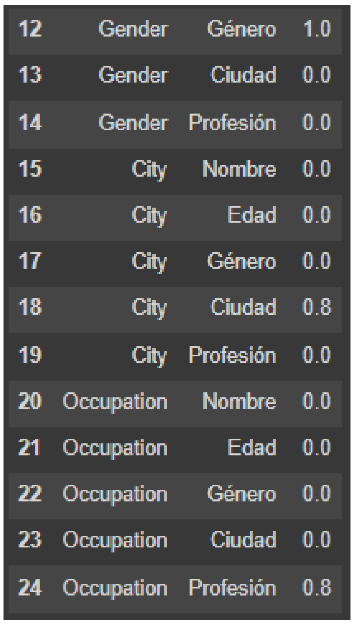


	Nombre	Edad	Género	Ciudad	Profesión
0	Juan	25	Masculino	Nueva York	Ingeniero
1	Alicia	30	Femenino	Los Ángeles	Médico
2	Roberto	35	Masculino	Chicago	Maestro
3	Emilia	28	Femenino	Houston	Artista
4	Miguel	40	Masculino	Miami	Abogado

Figure 8-9: Result obtained with matching strictly using BabelNet



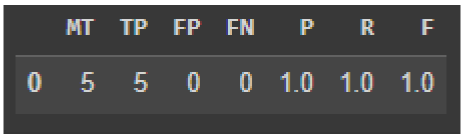
	A	B	sim
0	Name	Nombre	1.0
1	Name	Edad	0.0
2	Name	Género	0.0
3	Name	Ciudad	0.0
4	Name	Profesión	0.0
5	Age	Nombre	0.0
6	Age	Edad	1.0
7	Age	Género	0.0
8	Age	Ciudad	0.0
9	Age	Profesión	0.0
10	Gender	Nombre	0.0
11	Gender	Edad	0.0



12	Gender	Género	1.0
13	Gender	Ciudad	0.0
14	Gender	Profesión	0.0
15	City	Nombre	0.0
16	City	Edad	0.0
17	City	Género	0.0
18	City	Ciudad	0.8
19	City	Profesión	0.0
20	Occupation	Nombre	0.0
21	Occupation	Edad	0.0
22	Occupation	Género	0.0
23	Occupation	Ciudad	0.0
24	Occupation	Profesión	0.8

Worth noting that the matching between “City” and “Ciudad” is not a perfect match and it may be caused by an ambiguity and annotations may be needed to deal with it.

Figure 10-11: Result obtained with matching strictly using BabelNet

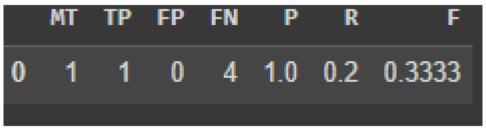


	MT	TP	FP	FN	P	R	F
0	5	5	0	0	1.0	1.0	1.0

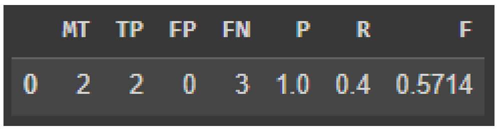
The results are optimal, and shows the capability of BabelNet to connect words and concepts of different languages together, and distinguish between relationship groups. Now we will attempt to match the schemas using label and value based methods, more precisely Levenshtein and Jaro similarity for labels and similarity join Jaccard for values, the results will combined with a weighted summation the weights we will apply for the study are the following:

- 0.5*Levenshtein + 0.2*Jaro + 0.3*Jaccard
- 0.3*Levenshtein + 0.2*Jaro +0.5*Jaccard
- 0.3*Levenshtein + 0.4*Jaro +0.3*Jaccard

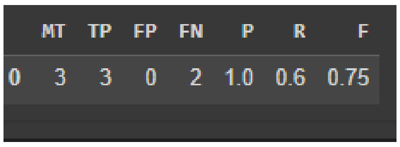
Figure 12-13-14: Results obtained with the typical methods combined with the weighted sum



	MT	TP	FP	FN	P	R	F
0	1	1	0	4	1.0	0.2	0.3333



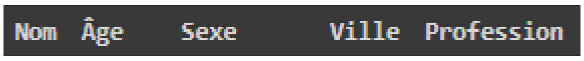
	MT	TP	FP	FN	P	R	F
0	2	2	0	3	1.0	0.4	0.5714



	MT	TP	FP	FN	P	R	F
0	3	3	0	2	1.0	0.6	0.75

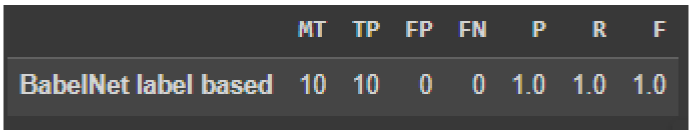
The second demonstration is the Top Down approach with a Global Schema in a different language and the matching is going to be strictly label based through BabelNet similarity measure. For simplicity’s sake we will use the same matching functions and weights for result combination.

Figure 15: GlobalSchema in French



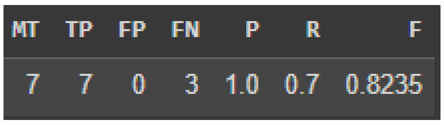
Nom	Âge	Sexe	Ville	Profession

Figure 16 : BabelNet top down schema matching results

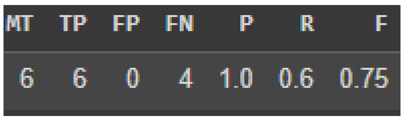


	MT	TP	FP	FN	P	R	F
BabelNet label based	10	10	0	0	1.0	1.0	1.0

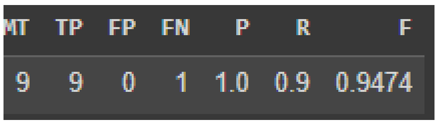
Figure 17-18-19: Results with the typical methods combined with the weighted sum



MT	TP	FP	FN	P	R	F
7	7	0	3	1.0	0.7	0.8235



MT	TP	FP	FN	P	R	F
6	6	0	4	1.0	0.6	0.75



MT	TP	FP	FN	P	R	F
9	9	0	1	1.0	0.9	0.9474

With these two cases we can see the full potential of Schema Matching through synonymity, Babelnet shines with words of the natural language, and is advised to be used in such use cases, label based approaches struggle with strings with big differences and value based approaches can also be limited by cases where the datasets have little to no similarity in the values.

4.1 Step by step demonstration

Another study worth exploring is to find the relationship between the following tables:

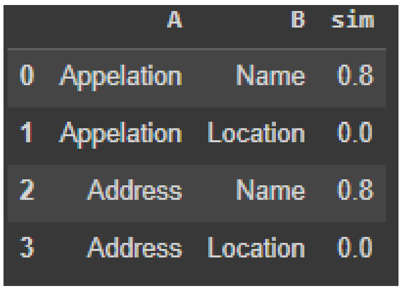
Listing 9-10: Person and Individual tables

Person	
Appellation	Address

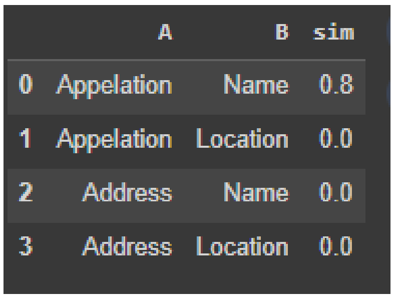
Individual	
Name	Location

We will apply our BabelNet API in order to match the two schemas, we will use two different approaches, the first one without annotations and the second with annotations. After the executing the matching the results are the following:

Figure 20-21: Results without annotations and with annotations



	A	B	sim
0	Appelation	Name	0.8
1	Appelation	Location	0.0
2	Address	Name	0.8
3	Address	Location	0.0



	A	B	sim
0	Appelation	Name	0.8
1	Appelation	Location	0.0
2	Address	Name	0.0
3	Address	Location	0.0

The results are not what we expected, why? For the first case we will explain each match:

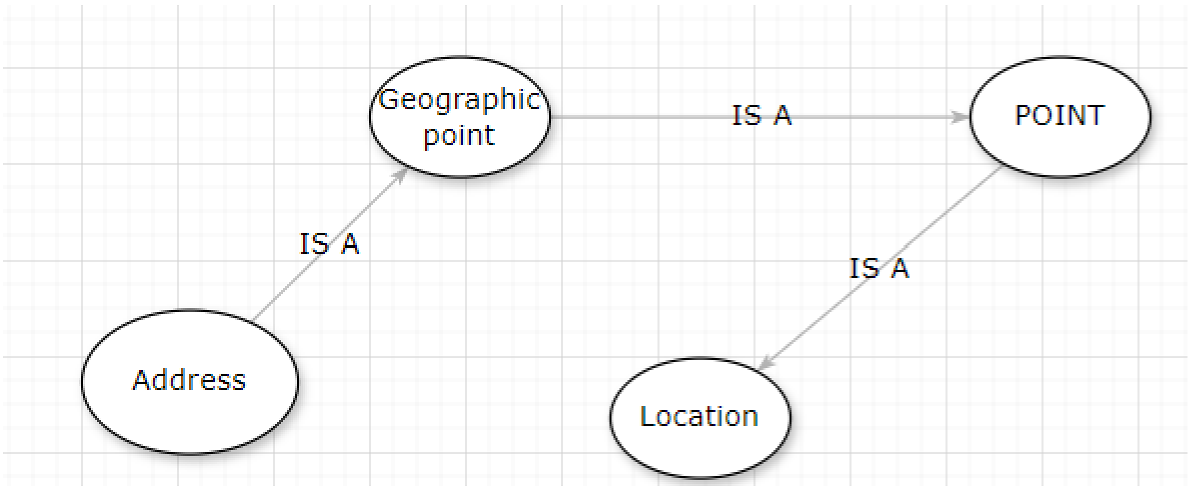
- Appellation-Name: The result indicates that ‘Name’ is the more generic form of appellation, and this explains why there is a Hypernimity/Hyponimity relationship
- Address-Name: In the english sense, Address is a name of a place where a person or organization resides, so a Hypernimity/Hyponimity relationship makes sense

The second case shows more False Negatives and only one match:

- Address-Name: the synset we annotated the Dataframe with has no relationship with the word Name or has a less direct relationship with it.
- Address-Location: Even here the relationship synsets are not directly linked, a deeper path must be taken for it to be found.

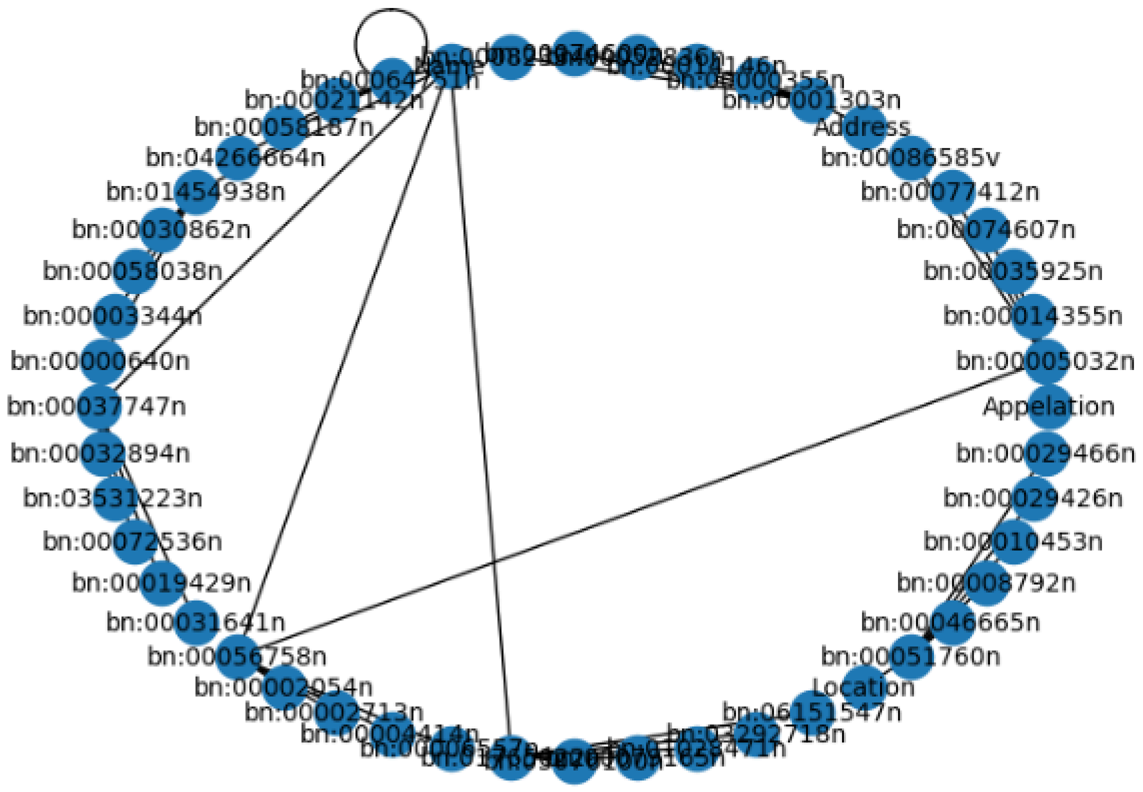
By investigating in BabelNet to search for the relationship between Address and Location not being found, a deeper study has been warranted, by manually calling the API starting from Address we find the following path.

Figure 22-23: Path between Address and Location



The path is much deeper than what our code is supposed to handle, the issue is caused by the structure of BabelNet itself being more sparse than expected. Our code does not find the relationship between “**GEOGRAPHIC POINT**” and “**POINT**”. Graph clustering might be a good solution but it’s really expensive, time consuming and the free API keys. Another aspect to be aware of is the growth of the graph as we gradually fetch data, filtering is advised since the graph can rapidly grow to a considerable size.

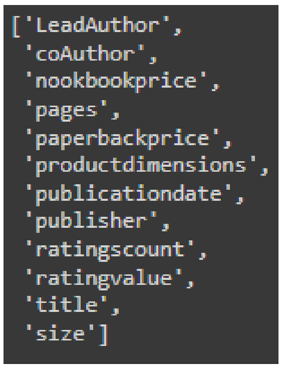
Figure 24-25: Graph built from this example



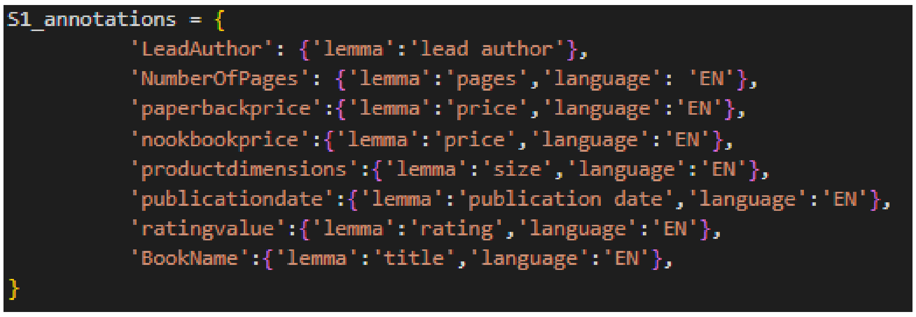
4.1 Top down attempt with real datasets

After testing with the generated datasets, it’s time to test the matching method on real datasets. In this case most of the column labels are not recognizable by BabelNet and for this reason we will annotate some columns and the matching will basically consist in a semi-automatic approach. The datasets have the following schemas and annotations:

Figure 26-27: Columns and annotations of the first dataset

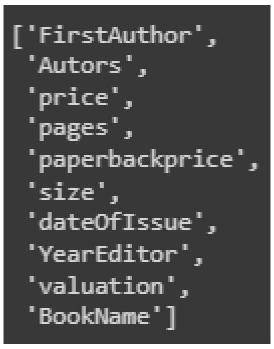


```
['LeadAuthor',
 'coAuthor',
 'nookbookprice',
 'pages',
 'paperbackprice',
 'productdimensions',
 'publicationdate',
 'publisher',
 'ratingscount',
 'ratingvalue',
 'title',
 'size']
```

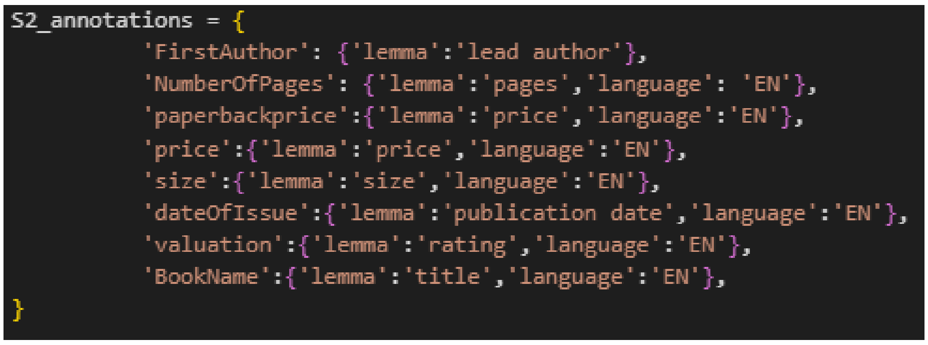


```
S1_annotations = {
         'LeadAuthor': {'lemma':'lead author'},
         'NumberOfPages': {'lemma':'pages','language': 'EN'},
         'paperbackprice':{'lemma':'price','language':'EN'},
         'nookbookprice':{'lemma':'price','language':'EN'},
         'productdimensions':{'lemma':'size','language':'EN'},
         'publicationdate':{'lemma':'publication date','language':'EN'},
         'ratingvalue':{'lemma':'rating','language':'EN'},
         'BookName':{'lemma':'title','language':'EN'},
}
```

Figure 28-29: Columns and annotations of the second dataset

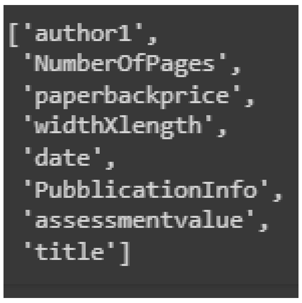


```
['FirstAuthor',
 'Autors',
 'price',
 'pages',
 'paperbackprice',
 'size',
 'dateOfIssue',
 'YearEditor',
 'valuation',
 'BookName']
```

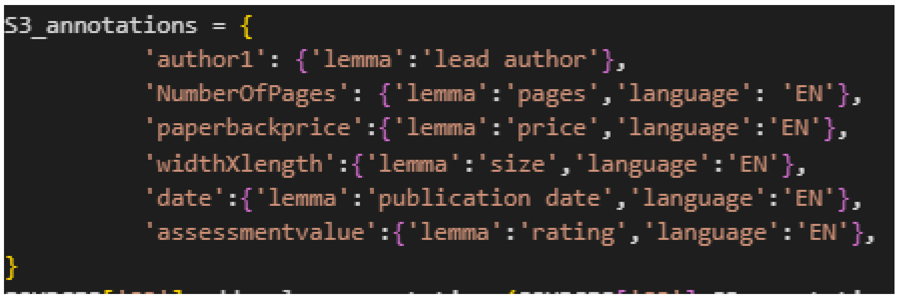


```
S2_annotations = {
         'FirstAuthor': {'lemma':'lead author'},
         'NumberOfPages': {'lemma':'pages','language': 'EN'},
         'paperbackprice':{'lemma':'price','language':'EN'},
         'price':{'lemma':'price','language':'EN'},
         'size':{'lemma':'size','language':'EN'},
         'dateOfIssue':{'lemma':'publication date','language':'EN'},
         'valuation':{'lemma':'rating','language':'EN'},
         'BookName':{'lemma':'title','language':'EN'},
}
```

Figure 30-31: Columns and annotations of the third dataset

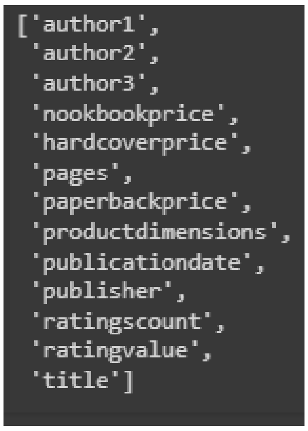


```
['author1',
 'NumberOfPages',
 'paperbackprice',
 'widthXlength',
 'date',
 'PubblicationInfo',
 'assessmentvalue',
 'title']
```

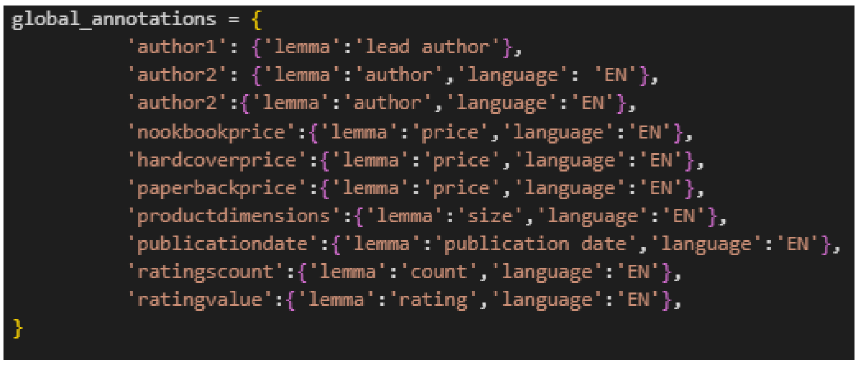


```
S3_annotations = {
         'author1': {'lemma':'lead author'},
         'NumberOfPages': {'lemma':'pages','language': 'EN'},
         'paperbackprice':{'lemma':'price','language':'EN'},
         'widthXlength':{'lemma':'size','language':'EN'},
         'date':{'lemma':'publication date','language':'EN'},
         'assessmentvalue':{'lemma':'rating','language':'EN'},
}
```

Figure 32-33: Columns and annotations of the GlobalSchema

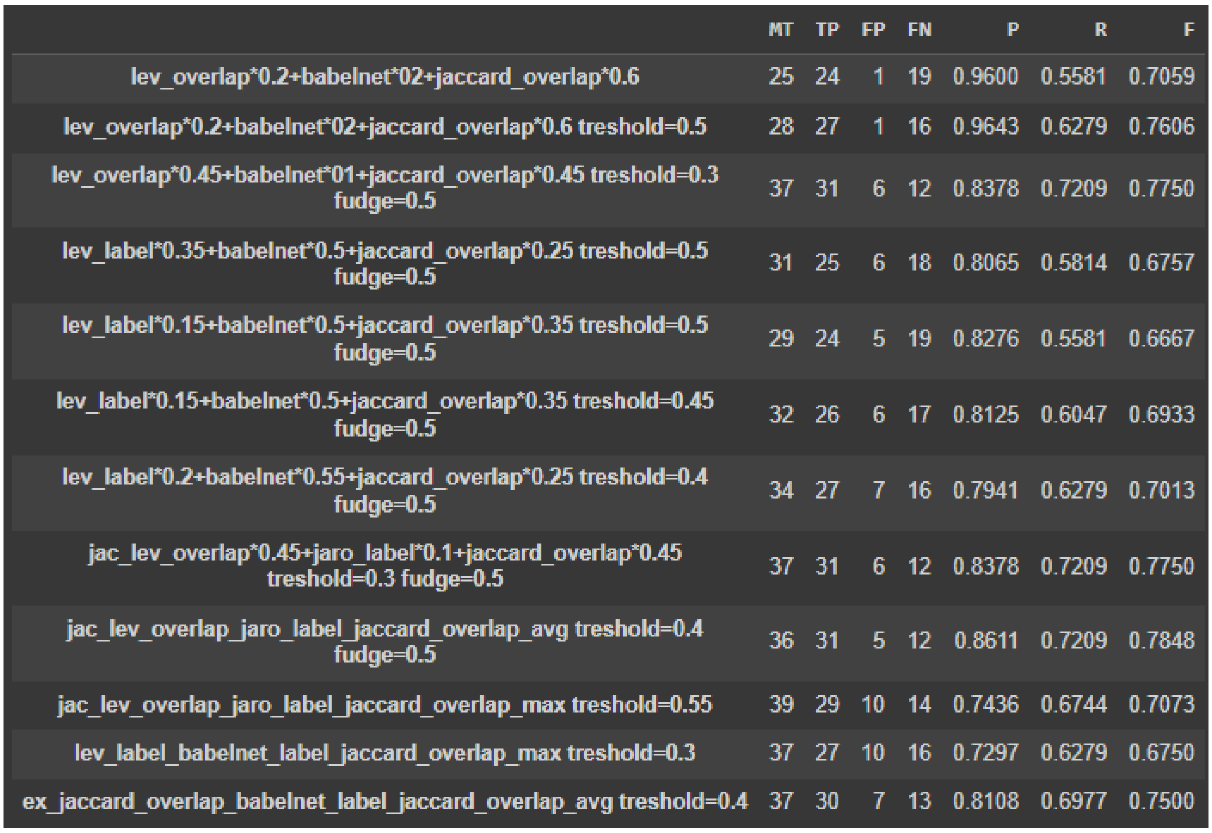


```
['author1',
 'author2',
 'author3',
 'nookbookprice',
 'hardcoverprice',
 'pages',
 'paperbackprice',
 'productdimensions',
 'publicationdate',
 'publisher',
 'ratingscount',
 'ratingvalue',
 'title']
```



```
global_annotations = {
         'author1': {'lemma':'lead author'},
         'author2': {'lemma':'author','language': 'EN'},
         'author2':{'lemma':'author','language':'EN'},
         'nookbookprice':{'lemma':'price','language':'EN'},
         'hardcoverprice':{'lemma':'price','language':'EN'},
         'paperbackprice':{'lemma':'price','language':'EN'},
         'productdimensions':{'lemma':'size','language':'EN'},
         'publicationdate':{'lemma':'publication date','language':'EN'},
         'ratingscount':{'lemma':'count','language':'EN'},
         'ratingvalue':{'lemma':'rating','language':'EN'},
}
```

Figure 34: Columns and annotations of the GlobalSchema



	MT	TP	FP	FN	P	R	F
lev_overlap*0.2+babelnet*02+jaccard_overlap*0.6	25	24	1	19	0.9600	0.5581	0.7059
lev_overlap*0.2+babelnet*02+jaccard_overlap*0.6 treshold=0.5	28	27	1	16	0.9643	0.6279	0.7606
lev_overlap*0.45+babelnet*01+jaccard_overlap*0.45 treshold=0.3 fudge=0.5	37	31	6	12	0.8378	0.7209	0.7750
lev_label*0.35+babelnet*0.5+jaccard_overlap*0.25 treshold=0.5 fudge=0.5	31	25	6	18	0.8065	0.5814	0.6757
lev_label*0.15+babelnet*0.5+jaccard_overlap*0.35 treshold=0.5 fudge=0.5	29	24	5	19	0.8276	0.5581	0.6667
lev_label*0.15+babelnet*0.5+jaccard_overlap*0.35 treshold=0.45 fudge=0.5	32	26	6	17	0.8125	0.6047	0.6933
lev_label*0.2+babelnet*0.55+jaccard_overlap*0.25 treshold=0.4 fudge=0.5	34	27	7	16	0.7941	0.6279	0.7013
jac_lev_overlap*0.45+jaro_label*0.1+jaccard_overlap*0.45 treshold=0.3 fudge=0.5	37	31	6	12	0.8378	0.7209	0.7750
jac_lev_overlap_jaro_label_jaccard_overlap_avg treshold=0.4 fudge=0.5	36	31	5	12	0.8611	0.7209	0.7848
jac_lev_overlap_jaro_label_jaccard_overlap_max treshold=0.55	39	29	10	14	0.7436	0.6744	0.7073
lev_label_babelnet_label_jaccard_overlap_max treshold=0.3	37	27	10	16	0.7297	0.6279	0.6750
ex_jaccard_overlap_babelnet_label_jaccard_overlap_avg treshold=0.4	37	30	7	13	0.8108	0.6977	0.7500

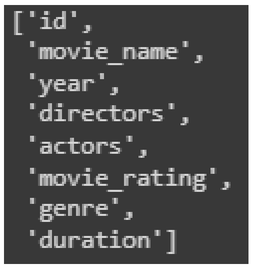
Given the results, two attempts made with BabelNet seem to have the higher precision and the second highest F1-measure. Other attempts come pretty close and some may have higher True Positives but at the cost of more False Negatives.

4.2 Bottom UP attempt with real datasets

For the Bottom Up use case we used datasets that represent a database for movies and their additional information such as actors, genre, release date etc..

In this case we need to come up with a schema of ours, some columns may not appear in other datasets and others may have ambiguous meanings. Also for this use case we will annotate some columns to smooth things up with the labels that BabelNet struggles to recognize.

Figure 35-36: Columns and annotations of the first dataset


```
['id',
 'movie_name',
 'year',
 'directors',
 'actors',
 'movie_rating',
 'genre',
 'duration']
```

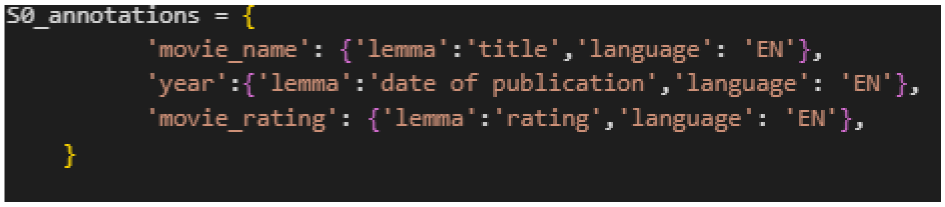
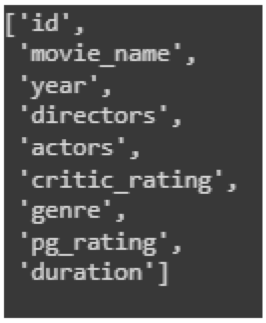

```
S0_annotations = {
        'movie_name': {'lemma':'title','language': 'EN'},
        'year':{'lemma':'date of publication','language': 'EN'},
        'movie_rating': {'lemma':'rating','language': 'EN'},
    }
```

Figure 37-38: Columns and annotations of the second dataset


```
['id',
 'movie_name',
 'year',
 'directors',
 'actors',
 'critic_rating',
 'genre',
 'pg_rating',
 'duration']
```

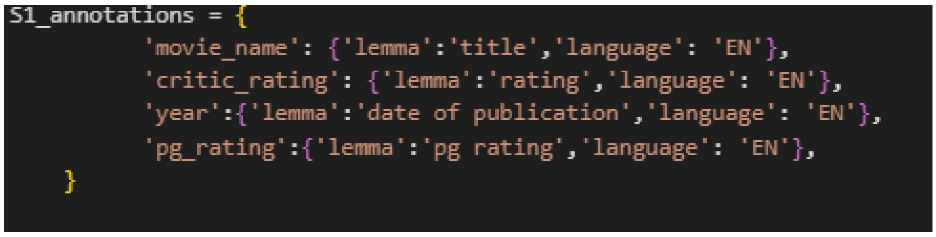
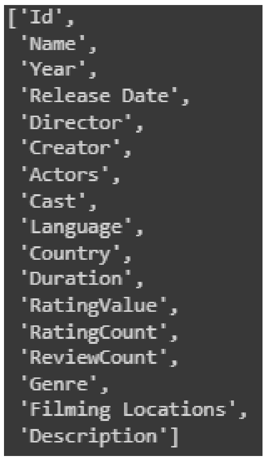
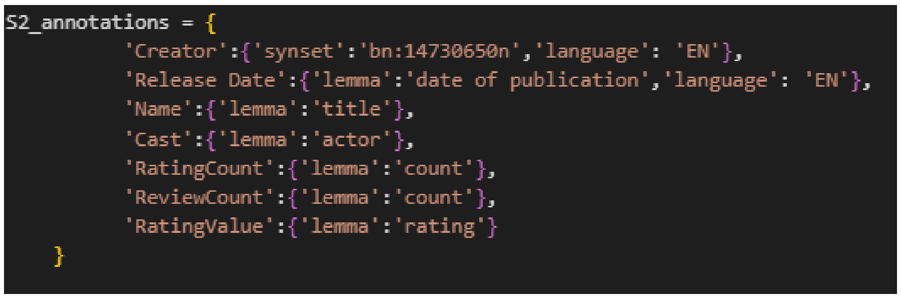

```
S1_annotations = {
        'movie_name': {'lemma':'title','language': 'EN'},
        'critic_rating': {'lemma':'rating','language': 'EN'},
        'year':{'lemma':'date of publication','language': 'EN'},
        'pg_rating':{'lemma':'pg rating','language': 'EN'},
    }
```

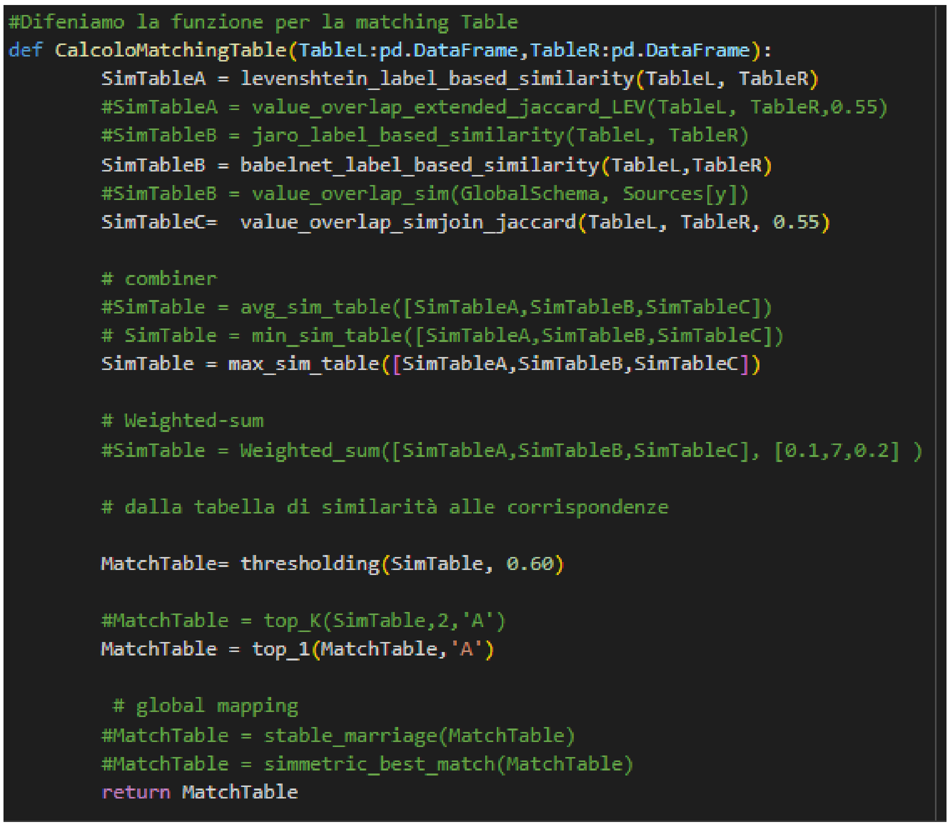
Figure 39-40: Columns and annotations of the third dataset


```
['Id',
 'Name',
 'Year',
 'Release Date',
 'Director',
 'Creator',
 'Actors',
 'Cast',
 'Language',
 'Country',
 'Duration',
 'RatingValue',
 'RatingCount',
 'ReviewCount',
 'Genre',
 'Filming Locations',
 'Description']
```


```
S2_annotations = {
        'Creator':{'synset':'bn:14730650n','language': 'EN'},
        'Release Date':{'lemma':'date of publication','language': 'EN'},
        'Name':{'lemma':'title'},
        'Cast':{'lemma':'actor'},
        'RatingCount':{'lemma':'count'},
        'ReviewCount':{'lemma':'count'},
        'RatingValue':{'lemma':'rating'}
    }
```

Our matching will be a mixed approach, being the following:

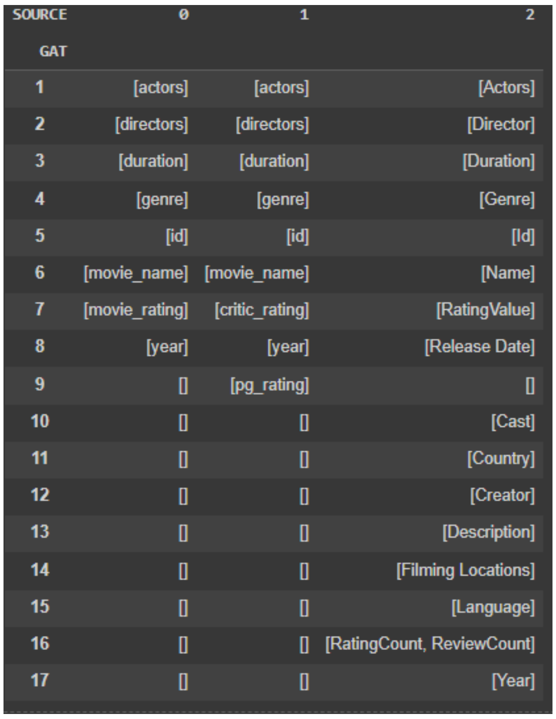
Figure 41: Matching function used for this study



```python
#Difeniamo la funzione per la matching Table
def CalcoloMatchingTable(TableL:pd.DataFrame,TableR:pd.DataFrame):
        SimTableA = levenshtein_label_based_similarity(TableL, TableR)
        #SimTableA = value_overlap_extended_jaccard_LEV(TableL, TableR,0.55)
        #SimTableB = jaro_label_based_similarity(TableL, TableR)
        SimTableB = babelnet_label_based_similarity(TableL,TableR)
        #SimTableB = value_overlap_sim(GlobalSchema, Sources[y])
        SimTableC=  value_overlap_simjoin_jaccard(TableL, TableR, 0.55)

        # combiner
        #SimTable = avg_sim_table([SimTableA,SimTableB,SimTableC])
        # SimTable = min_sim_table([SimTableA,SimTableB,SimTableC])
        SimTable = max_sim_table([SimTableA,SimTableB,SimTableC])

        # Weighted-sum
        #SimTable = Weighted_sum([SimTableA,SimTableB,SimTableC], [0.1,7,0.2] )

        # dalla tabella di similarità alle corrispondenze

        MatchTable= thresholding(SimTable, 0.60)

        #MatchTable = top_K(SimTable,2,'A')
        MatchTable = top_1(MatchTable,'A')

         # global mapping
        #MatchTable = stable_marriage(MatchTable)
        #MatchTable = simmetric_best_match(MatchTable)
        return MatchTable
```

The results of our matching are the following:

Figure 42: Matching results



SOURCE	0	1	2
GAT			
1	[actors]	[actors]	[Actors]
2	[directors]	[directors]	[Director]
3	[duration]	[duration]	[Duration]
4	[genre]	[genre]	[Genre]
5	[id]	[id]	[Id]
6	[movie_name]	[movie_name]	[Name]
7	[movie_rating]	[critic_rating]	[RatingValue]
8	[year]	[year]	[Release Date]
9	[]	[pg_rating]	[]
10	[]	[]	[Cast]
11	[]	[]	[Country]
12	[]	[]	[Creator]
13	[]	[]	[Description]
14	[]	[]	[Filming Locations]
15	[]	[]	[Language]
16	[]	[]	[RatingCount, ReviewCount]
17	[]	[]	[Year]

The results are satisfactory, showing great accuracy in the matching, the only downside is inability to match Year of the third DataFrame with the other “year” columns. This may be caused by the Top1 filtering, and transitioning to Top2 will create a matching of only False Positives.

5 Conclusions

The outcomes demonstrate the efficacy of BabelNet in connecting words and concepts across different languages, while also discerning between various relationship groups. BabelNet particularly excels with words from natural languages and is recommended for such scenarios. In contrast, label-based approaches encounter challenges when dealing with strings exhibiting significant disparities, while value-based approaches may be constrained when datasets lack similarity in values. Additionally, BabelNet encounters difficulties with automatically generated labels, as they often do not adhere to natural language conventions. Furthermore, utilizing the API for data retrieval may not be optimal due to latency issues, with retrieval times ranging from 10-100ms and potentially escalating significantly with an increase in the number of columns. In conclusion, BabelNet emerges as an optimal tool for label-based schema matching, complementing other conventional matching methods. It is essential to consider its application alongside other methods to cater to diverse use cases effectively.
This study also shows that some continuations may be possible such as programming a fetcher with the paid API key, build the graph on a dedicated Graph Database such as Neo4j and run algorithms on it for further insights and better results for different use cases.