Author Name Disambiguation: Report and Review

Anurag Nihal
Roll-19042
Data Science and Engineering, IISER Bhopal

Definition

Author name ambiguity[1] is a problem that occurs when a set of publication records contains ambiguous author names, i.e., the same author may appear under distinct names (synonymy), or distinct authors may have similar names (polysemy). This problem decreases the quality and reliability of information retrieved from digital libraries such as the impact of authors, the impact of organizations, etc. Therefore, author name disambiguation is a critical task in digital libraries.

General Approaches

There are two approaches to author name disambiguation:

1. grouping publication records of a same author by finding some similarity among them (author grouping methods)
2. directly assigning them to their respective authors (author assignment methods)

Github Repository Selected to Review

Arizona State University Digital Innovation Group's **diging** Repository

Repository's Approach

The Repository is seen to approach the problem of author name disambiguation by using first general approach and then dealing it based on Machine Learning Classification problem. It implements simple concepts such as cosine similarity or other preprocessings to obtain a processed dataset for modeling.

Installation Experience

The Repository needs Python2.7 for its operation. Though I successfully converted almost all files (except `PaperParser.py`) into Python3 syntax, however, due to dependency of `tethne` module in `PaperParser.py` file on Python2, I was not 100% successful in doing it.

[1] In literature some also call it as Record Linkage.

Also, since Python 2.7 reached the end of its life on January 1st, 2020, installing it on Windows 10 system with `tethne` module was not possible until depreciated MS VisualC++ 9.0 version[2] could be installed. We should note that this VisualC++ version is also no longer available in stable release in Microsoft website. Hence, I had to work through it and was able to get the `.msi` file link in StackOverflow.

`tethne` module has been extensively tested for MacOSX, but for Windows some people may face problem in Installation. Hence, creating a suitable *virtual environment* is recommended.[3]

```
//creating virtual environment using virtualenv
//goto the required project folder and open CommandLine
//and type following:
virtualenv your_venv_folder_name --python=2.7
```

Repository Files Summary

`PaperParser.py`: This class uses `Tethne` to parse WOS tagged-file data and write the output to a CSV file. Then we can use the `class DataAnalysisTool.py` on the output csv to perform data analysis.
`DataAnalysisTool.py`: This class has methods and tools to analyse a bunch of (World of Science) WOS papers objects. DataSet is read from a CSV. A CSV file of expected format can be easily created using the `class PaperParser.py`.
`DistanceMetric.py`: This defines various similarity metrics in this class.
`TrainingDataGenerator.py`: This class is responsible for generating Training records.

New Modules Explored

```
Parsing bibliographic metadata: tethne
Preprocessing Modules: pandas, re
Similarity Metrics Module: fuzzywuzzy
Visualisation Module: matplotlib
Modeling Modules: scikit-learn
Python syntax grammar Module: ast
```

`tethne` Module

`tethne` module provides tools for easily parsing and analyzing bibliographic data in Python. The primary emphasis is on working with data from the ISI Web of Science database, and providing efficient methods for modeling and analyzing citation-based networks. In the codefile `PaperParser.py`, it is used to parse WOS tagged-file data. `tethne` relies on `NetworkX` for graph classes, and leverages its network analysis algorithms. Some basic examples for understanding:

```
import tethne.readers as rd # imports tethne's readers.wos module
papers = rd.wos.read("/Path/to/savedrecs.txt")
```

[2] https://stackoverflow.com/questions/43645519/microsoft-visual-c-9-0-is-required
[3] // represents comments

`fuzzywuzzy` Module

`fuzzywuzzy` is a library of Python which is used for string matching. Fuzzy string matching is the process of finding strings that match a given pattern. Basically it uses Levenshtein Distance to calculate the differences between sequences. `fuzzywuzzy` library gives the score out of 100, that denotes two string are equal by giving similarity index. Some basic examples for understanding[4]:

```
from fuzzywuzzy import fuzz
from fuzzywuzzy import process

s1 = "I love GeeksforGeeks"
s2 = "I am loving GeeksforGeeks"
print "FuzzyWuzzy Ratio: ", fuzz.ratio(s1, s2)
print "FuzzyWuzzy PartialRatio: ", fuzz.partial_ratio(s1, s2)
print "FuzzyWuzzy TokenSortRatio: ", fuzz.token_sort_ratio(s1, s2)
print "FuzzyWuzzy TokenSetRatio: ", fuzz.token_set_ratio(s1, s2)
print "FuzzyWuzzy WRatio: ", fuzz.WRatio(s1, s2),'\n\n'

# for process library,
query = 'geeks for geeks'
choices = ['geek for geek', 'geek geek', 'g. for geeks']
print "List of ratios: "
print process.extract(query, choices), '\n'
print "Best among the above list: ",process.extractOne(query, choices)

Ouputs:
FuzzyWuzzy Ratio:  84
FuzzyWuzzy PartialRatio:  85
FuzzyWuzzy TokenSortRatio:  84
FuzzyWuzzy TokenSetRatio:  86
FuzzyWuzzy WRatio:  84


List of ratios:
[('g. for geeks', 95), ('geek for geek', 93), ('geek geek', 86)]

Best  among the above list:  ('g. for geeks', 95)
```

Metadata Mining and Parsing

In `PaperParser.py`, `tethne` is used for parsing WOS files. In the file, we have a `class PaperParser` which has two methods apart from `__init__` which initialises the parser object by taking parameters `inputlocation`, `outputlocation` and default parameter `output_filename` with value `None`:

[4]https://geeksforgeeks.org

1. `parseDirectory`: This object method searches recursively for WOS txt files and parses each one of them. It handles files and first writes the required headers. Then, it starts two `for` loops to open each `.txt` file and then using another 2 `for` loops, it writes rows for each author in each paper.

```
# modified the original location
parser = PaperParser('MBL History Data/1971/Albertini_David.txt',
                         'Dataset', output_filename='records.csv')
parser.parseDirectory()
```

2. `parseFile`: This object method parses a single WOS file passed in the input. The difference between this and `parseDirectory` is that it only parses single WOS txt file.

```
# modified the original location
parser = PaperParser('MBL History Data/1971/Albertini_David.txt',
                         'Dataset')
parser.parseFile()
```

Similarity Metrics

In the `DistanceMetric.py`, we define our similarity metrics. In this file, we have two functions:

1. `sentence_to_vector`: In this function, we tokenize the sentenceby using Collections.Counter() method to generate words in sentence as dictionary keys, and their counts are stored as dictionary values.

2. `cosine_similarity`: In this function, we take the tokens as vectors from the `sentence_to_vector` function. Then, we compute dot product and the magnitude and return the cosine value of the vectors made from sentences, only if `product_of_magnitude` is not zero. Otherwise, it will return zero.

Analysis of WOS paper objects

In `DataAnalysisTool.py`, we have `class DataAnalysisTool`, in which we have different methods to visualize and analyze the WOS paper.
It has an `__init__` which initialises the class object by taking `InputDataset` (pandas dataframe). Most of the methods are clear by their names only and are simple Exploratory Steps such as getting papers using either first names, last names or both.
We explain certain methods which are non-trivial in the procedure.

1. `drawHistogramForOverlapScore`: It takes papers dataframe and a fetaure name. Then firstly, this method initialises a numpy square matrix of dimensions as length of papers dataframe (rows). Then, we compute overlap score by checking how many of the feature

value for each paper overlaps with another paper and the ratio becomes the score. We store this score in the 2-D matrix and since, using only upper triangle of the matrix we get the list of desired values to be visualised in the histogram for every unique (i^{th}, j^{th}) papers.

2. `drawhistrogramForCosineScore`: It takes feature list as parameter. Then, using `sentence_to_vector`, it converts each string the fetaures to vectors and using `cosine_similarity`, we input similarity score in the matrix which as similar to `drawHistogramForOverlapScore` is used for histogram Visualisation.

3. `getInstituteNamesForAuthor`: It takes authorNames, currentAuthorPapers as parameter. It iterates through every row of currentAuthorPapers to get institute list of the authors by evaluating whether the resultant is no "[ ]". If it is, it returns nothing.

4. `getInstituteNamesForRandomRecords`: It is similar to `getInstituteNamesForAuthor` but, instead we take papers and just output list of institutes only.

Interesting point to notice: The author consciously chose `literal_eval` instead of `eval` to avoid evaluation of function if invalid input datatype is provided.

Training Data Generator

In `TrainingDataGenerator.py`, we first make some variations for name and declare a dictionary to map each (author specific)csv file corresponding to the possible firstnames and lastnames. These dictionaries will be used when we call the generate() method to generate the training data.
It also has `class TrainingDataGenerator`, which need the papers(pandas dataframe), to initialise and iterate on and generate the scores. Some of its methods are declared as static methods whereas some are declared as class methods.
The difference between class method and static method is that the class method can access class attribute where as static method can't.

1. `get_score_for *` methods: These methods return the overlap score between two features in a record. These are calculated for features such as email, authorkw, institute, etc.

2. `get_institute_name`: This method finds the institute name to which the author belongs. If there is no "[ ]" in the field, we return the string as it is. If there's "[ ]" then until and unless there is some mapping with author, we return None. Otherwise, we map the institute with author in "[ ]".

3. `compare_institute_name`: This method returns the cosine similarity of the two institutes by converting them to vector and then finding cosine value.

4. `calculate_scores`: This method compute scores for various features by either using previous methods or by using modules such as `fuzzywuzzy.fuzz`.

5. `generate_records`: It generates records to have all attributes as columns in the list to eventually become a pandas dataframe when every attribute and rows are appended. These records are of positive matches.

6. `generate_records_for_negative_cases`: Using the positive case records in `generator` method, we make records of negative cases by eliminating such pairs. In this method, we also call `DataAnalysisTool` class to analyze positive cases.

7. `generate`: This method using positive cases generated, calls to generate negative cases and then generates a training record dataframe and other with training record dataframe with only their attributes score. This generates two training files: (1) train.csv and (2) scores.csv .

Modeling using Scikit-learn Algorithms

Using `scores.csv` file as dataset, we start modeling different Machine Learning models for binary class Classification. The authors chose to test Support Vector Machine Classifier and Random Forest Classifier. The SVMClassifier suffers from the problem of dimension scaling i.e., we have to make sure all of the attributes are scaled to similar scaling. However, since, RandomForest-Classifier is a tree based classifier which does not produces much different result even if we don't scale the attributes. Also, generally, RandomForestClassifier is more resistant to overfitting than SVMClassifier.

Also, we notice that the author after modeling base models, chose to perform cross-validation which reduces overfitting in the resultant scoring.

We also notice that `pickle` file for RandomForestClassifier which means the selected model by the author(s) was RandomForestClassifier. However, I tried to implement the modeling part[5] using `linear models` and obtained a good results with less complexity than SVMs and RandomForest.

Things Learnt

1. Explored new libraries such as fuzzywuzzy, re, tethne
2. Explored preprocessing Techniques and implementation for obtaining data in required format
3. Handling exceptions and making wise choice in using code specific features.
4. Handling a project structure
5. Modeling an ML algorithm and getting good predictions.

[5]Modeling part in `trial.ipynb` file