Annotation of Biological data

OLEKSANDR KHOROSHEVSKYI*, VALÉRIO DA SILVA FIORI*, and LEON INGELSE*, University of Lisbon, Portugal

In this paper we present a method for the Wikidata annotation of tabular data. Specifically, we apply the method on a biological data set. First, our method classifies the different columns of the table as distinct Wikidata classes. After that, it uses this classification to extract the different cell values of the table. The results are promising, but there is still much that can be improved in future work.

Additional Key Words and Phrases: data annotation, Wikidata, knowledge graph, tabular data

ACM Reference Format:
Oleksandr Khoroshevskyi, Valério da Silva Fiori, and Leon Ingelse. 2021. Annotation of Biological data. 1, 1 (May 2021), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Annotation of structured data is an often-recurring issue within Bioinformatics. An effort to forward the technologies of annotating structured data were SemTab 2019 [2] and 2020 [3]. These were the first and second editions of the Semantic Web Challenge on Tabular Data to Knowledge Graph Matching (SemTab).

As the name suggests, SemTab poses the challenge of annotating tabular data with a knowledge graph (KG), in this case WikiData. SemTab's main motivation is generalising the "gaining semantic understanding" of tabular data, in order to simplify all sorts of data integration, machine learning, and data mining tasks [3]. As identified by [3], the challenge can be split up in 3 separate issues; "(i) [table] cell to KG entity matching; (ii) [table] column to KG class matching; and (iii) [table] column pair [relation] to KG property matching".

In this paper we will introduce our method of tabular data annotation with WikiData data. Our programme is written in Python [7]. We used a Windows 10 machine with 8 logical processors (7th generation i7). After introducing our method and implementing this method, this paper will guide the reader through our results, and finally present a conclusion.

2 METHODS

Given an annotation-lacking input table, our method can be summarized into the following steps. First, for (part of) the input data, each cell is searched for on Wikidata. In section 2.1 we look at the class of these results, and decide on the KG class of that column, equivalent to point (ii), as defined above. Fortified with this information, we continue with classifying all cells of the dataset, equivalent to point (i) above. In sections 2.2 and 2.3, for every cell in every row,

*All authors contributed equally to this research.

Authors' address: Oleksandr Khoroshevskyi, fc57297@alunos.fc.ul.pt; Valério da Silva Fiori, fc55782@alunos.fc.ul.pt; Leon Ingelse, fc55969@alunos.fc.ul.pt, University of Lisbon, Lisbon, Portugal.

Manuscript submitted to ACM

we search an entity in the KG and for relations between these entities. Based on this information we choose the most suitable Wikidata representative for each row in section 2.4.

2.1 Column to KG class matching

The first step in our method is a column to KG class matching. For this we use Algorithm 1. For every column, this algorithm finds classes that match the column well. First, it goes through all the cells of the column and searches each cell on Wikidata with the search_wiki function, and records the class of the results. Then it divides into two methods:

- Method 1A: We take into account all the classes found for each entity. These are counted and then sorted based on the number of appearances.
- Method 1B: Only if the search returns a single class, it is taken into account. For each column, all these classes are counted and then sorted. This algorithm is written in pseudo code in Figure 1.

```
Algorithm 1B: Column to KG class matching

Input:    Table T with c columns and n rows
Output:   List L of length c with scores for classes
L <- empty list
for each column of T do
    L' <- empty list
    for each row of T do
        results <- search_wiki(T[col][row])
        if length(results) = 1 then
            L'.append(results)
        end
    end
    L[col] <- sort(count(L'))
end
return L
```

Fig. 1. Algorithm 1B

The most expressed class will be the one with the most counts. Therefore this will be the most likely column match. As both methods give different results, we decided to test both methods on our data set. In section 3.2 we will discuss these results.

2.2 Possible entities selection

The second step is obtaining the possible Wikidata entities, for each row of the data. This algorithm is written for a single row in pseudo code in Figure 2. For each column try to find the Wikidata pages of that column with the search_wiki_page function. We only keep the n first results.

```
Algorithm 2: Possible entities selection

Input:    A row r with c columns, and a limit n
Output:   List R with possible entities from Wikidata
possible_entities <- empty list
for each column of row do
    pos_entities <- search_wiki_page(column)
    possible_entities <- pos_entities[1:n]
end
return possible_entities
```

Fig. 2. Algorithm 2

2.3 Combine entities

The third step entails the combining of different entities through their relations. Wikidata gives information about possible relations, which we exploit by looking whether the different possible entities of every column are related and in what way. In this way we

```
Algorithm 3: Combine entities

Input:    List R with possible entities from Wikidata
Output:   List R with linked entities from Wikidata
possible_combinations <- empty list
for every combination of columns do
    columns_combination
    for every two possible entities do
        if exist a relation between the two entities then
            columns_combination.append(entity1,relation,entity2)
        end
    end
    possible_combination.append(columns_combination)
end
return possible_combinations
```

Fig. 3. Algorithm 3

end up with a list of possible entity pairs. This algorithm is written for a single row in pseudo code in Figure 3. After this, not shown in 3, we combine the different pairs and look for combinations that encompass the complete row. In this way we only get rows that are actually linked on Wikidata.

2.4 Choose the most suitable Wikidata result

The last step is choosing the most suitable of the results found in the previous step. Similar to the first algorithm, we have two possibilities here. Either we know the columns' classes (Algorithm 4A), or we don't (Algorithm 4B). In Figures 4 and 5. Note that, if Algorithm 4A doesn't give a result, the default Algorithm 4B is used.

```
Algorithm 4A: Classes are known - Choose the most suitable Wikidata result

Input:      List R with possible entities from Wikidata
Output:     Row of most suitable entities
1:          for every possible answer do:
2:              |   If possible answer and classes are equal then
3:              |       |   return possible answer
4:              |   end
5:              |   If possible answer and classes are equal but shuffled then
6:              |       |   possible answer <- sorted(possible answer)
7:              |       |   return possible answer
8:              |   end
9:          If nothing returned then:
10:             |   Continue with algorithm 4B
11:         end
```

Fig. 4. Algorithm 4A

```
Algorithm 4B: Classes are not known - Choose the most suitable Wikidata result

Input:      List R with possible entities from Wikidata
Output:     Row of most suitable entities
1:          R <- sort_on_number_of_gaps(R)
2:          row <- first(R)
3:          If some cells of row are empty then:
4:              |   Fill cells with answers of other rows
5:          end
6:          return row
```

Fig. 5. Algorithm 4B

Currently, the algorithms base their choice on very simple criteria: whether the found classes match the specified classes, and if there are any gaps in found data, we simply try to fill it with the cells of another found answer. These criteria could be extended. Some ideas would be to filter out rows that had the same result for multiple cells, or to give more importance to the match of certain columns. These methods were out of scope for this project.

2.5 The test data set

As a test data set, we were given 3 CSV files with input data, and 3 corresponding ground truth data sets with Wikidata entities. Little research showed that the data was biological data.

2.6 Scoring measure of accuracy

To score our programme, we created a slightly-adjusted-accuracy scoring measure. In stead of using the standard accuracy measure as we know it, we decided to weigh wrong classifications differently,

```
Scoring method of classified data

Input:      row r of classified data and row gt of ground truth
Output:     Score between 0 and 1
1:          if r equals gt then
2:              |   return 1
3:          elif first two columns of r switched in comparison to gt do
4:              |   return 0.75
5:          elif first two columns equal, rest of column different do
6:              |   return 0.5
7:          elif first two columns switched, rest of column different do
8:              |   return 0.25
9:          else
10:             |   return 0
11:         end
```

Fig. 6. Scoring measure

Column: 1	Column: 2	Column: 3	Column: 4	Column: 5
- Q8054 --> 62.91% **- Q7187 --> 31.77%** - Q417841 --> 1.38% - Q13442814 --> 0.61% - Q67015883 --> 0.09% - Q14860489 --> 0.05% - Q929833 --> 0.02%	**- Q8054 --> 76.79%** - Q7187 --> 18.9% - Q417841 --> 2.16% - Q13442814 --> 0.59% - Q0 --> 0.56% - Q898273 --> 0.42% - Q81505329 --> 0.42% - Q47461827 --> 0.12% - Q67015883 --> 0.05%	**- Q2996394-->59.25%** - Q0 --> 19.53% - Q13442814 --> 3.02% - Q8054 --> 1.59% - Q14860489 --> 0.89% - Q84467700 --> 0.77% - Q24017414 --> 0.35% - Q67101749 --> 0.28% - Q5058355 --> 0.23% - Q81505329 --> 0.19% - Q898273 --> 0.09% - Q417841 --> 0.07% - Q7644128 --> 0.07% - Q4915012 --> 0.02%	**- Q14860489 → 61.88%** - Q0 --> 14.09% - Q8054 --> 6.54% - Q2996394 --> 1.06% - Q13442814 --> 0.66% - Q81505329 --> 0.59% - Q898273 --> 0.59% - Q5058355 --> 0.28% - Q417841 --> 0.07% - Q67101749 --> 0.05%	**- Q5058355-->58.01%** - Q0 --> 10.04% - Q2996394 --> 9.43% - Q84467700 --> 2.13% - Q14860489 --> 1.15% - Q13442814 --> 0.07% - Q898273 --> 0.05% - Q81505329 --> 0.05%

Fig. 7. Results Algorithm 1A

Column: 1	Column: 2	Column: 3	Column: 4	Column: 5
- **Q7187 --> 100.0%**	- **Q8054 --> 98.76%** - Q66826848 --> 0.93% - Q417841 --> 0.31%	- **Q2996394 --> 98.89%** - Q4915012 --> 0.44% - Q30612 --> 0.22% - Q210973 --> 0.22% - Q20732156 --> 0.22%	- **Q14860489 → 99.77%** - Q67015883 --> 0.23%	- **Q5058355 --> 88.06%** - Q78155096 --> 8.96% - Q67015883 --> 1.49% - Q67101749 --> 1.49%

Fig. 8. Results Algorithm 1B

to incorporate the difference of importance of the classification of different columns. In Figure 6 this measure is given in pseudo code.

3 IMPLEMENTATION

The code we created is free to use and downloadable from https://github.com/Khoroshevskyi/Annotation-of-biological-data.

3.1 Technologies

All our programme was run in Python [7]. For searching on Wikidata we made extensive use of Pywikibot [6]. We made some use of Pandas [8] to organise our data. Both Pandas and NumPy [4] were used for some data manipulation. MatPlotLib [5] was used to visualize our results.

3.2 Algorithm 1A and 1B comparison

As discussed in section 2.1, for the column-to-KG-class-matching step, we created two different algorithms, 1A and 1B. In Figures 7 and 8, you can see the results. For each column it shows the percentage of Wikidata classes found for the instances in that column. Highlighted is the selected class representative.

What directly stands out is that the classes for Algorithm 1A are far more varied, and more widely distributed. This is because, for every cell, the algorithm returns all top 5 classes found. As the first and second column are mostly classified as genes and proteins, and as genes and proteins often carry the same names, we see a lot of ambiguity. Columns 1 and 2 even have the same most expressed class. Algorithm 1B gives more unambiguous results. The lowest percentage given is 88 %, which is not bad if seen as a certainty. Mainly because of this lack of ambiguity, we decided to use Algorithm 1B as a standard.

4 RESULTS

As discussed in section 2.4, we created two algorithms, one, Algorithm 4B, for when the classes of the columns are given (for example by Algorithm 1), and another, Algorithm 4A, for when this information is not available.

In Figure 9 you can see the accuracy of both methods plotted together for the three different input data files. The blue bar represents Algorithm 4A, and the orange Algorithm 4B. To start, an accuracy around 63 % is a promising first attempt. This attempt has many improvement possibilities, which we will discuss in section 4.1.

On average, the accuracy is the same for both methods. This was not expected. Probably, the column classification was not specific enough to aid the algorithm in obtaining significantly-better results. For the run time of the algorithm, there was a slight improvement. Where Algorithm 4A ran for an average of 215.3 minutes, Algorithm 4B showed higher performance, with 199.7 minutes, a speed up of 7 %.

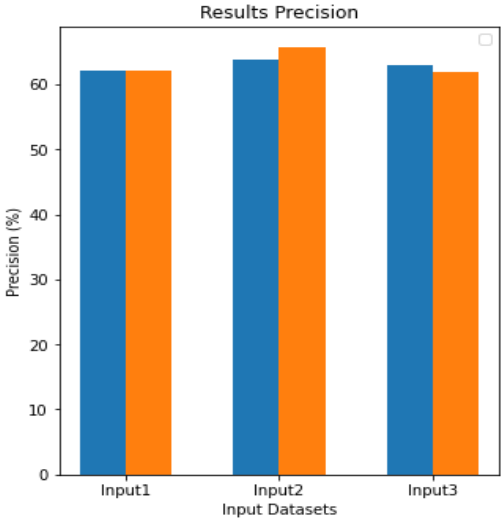

Fig. 9. Results comparison

4.1 Discussion and future work

Two parts stand out in our results. Firstly, our code is very slow. Secondly, the classification of the columns did not allow us to exploit some of the KG's underlying information to obtain improved results.

Most importantly, our method's accuracy should be improved. As discussed in section 1, there are three issues to classifying tabular data, of which two were addressed in this project, column classification and cell entity matching. One was not, namely column relationship classification. In the same way that we tried to use the column classification to improve our programme's accuracy, we could use column relationship classification with the same aim. In our particular case, we often noticed that the gene in the first column was found, but the protein in the second not, showing some defect in our programme. Adding a column relationship classification would help to solve this issue.

Also some data cleaning could enhance performance. For example, some cells contained quotation marks, which made resulted to a failed matching. Fixing such errors would be fruitful, but wouldn't improve accuracy by a whole lot.

To speed up our code, searching methods on Wikidata should be further explored. Currently, all of Wikidata is searched for a single cell, and then the search results are filtered based on the classification method. Using SPARQL [1], we would be able to specify the class within the query, significantly decreasing the search domain.

Another promising improved could come from parallelising the programme. The algorithms are easily parallelisable, as the search results of the each row are independent of the other rows. First results showed a reduction to 140 minutes of runtime for a single file. Here, 8 cores were used, but only a single Wikidata API and internet connection, so there is room for higher speed ups.

5 CONCLUSION

Tabular data annotation revealed to be a hard task, especially with biomedical data. Nevertheless, our code showed to get good results despite the hard challenges that arise in this domain. Exploitation and the use of Wikidata's information

on entity relations is a promising possible improvement step. Furthermore, dealing with the noise in the input data would be helpful to obtain a higher accuracy. To improve the running time of the algorithms, we could start using SPARQL, and further experiment with the parallelisation of our code.

REFERENCES

[1] Andy Seaborne Eric Prud'hommeaux. 2008. SPARQL Query Language for RDF. (January 2008). https://www.w3.org/TR/rdf-sparql-query/

[2] Vasilis Efthymiou Jiaoyan Chen Ernesto Jiménez-Ruiz, Oktie Hassanzadeh and Kavitha Srinivas. 2020. SemTab 2019: Resources to Benchmark Tabular Data to Knowledge Graph Matching Systems. *Extended Semantic Web Conference (ESWC)* (2020).

[3] Vasilis Efthymiou Jiaoyan Chen Kavitha Srinivas Ernesto Jiménez-Ruiz, Oktie Hassanzadeh and Vincenzo Cutrona. 2020. Results of SemTab 2020. *CEUR Workshop Proceedings* 2775 (2020).

[4] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'10, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. *Array programming with NumPy.* https://doi.org/10.1038/s41586-020-2649-2

[5] J. D. Hunter. 2007. *Matplotlib: A 2D graphics environment.* https://doi.org/10.1109/MCSE.2007.55

[6] The Pywikibot Team. 2021. *Pywikibot.* https://github.com/wikimedia/pywikibot

[7] G. van Rossum. 1995. *Python tutorial, Technical Report.*

[8] Wes McKinney. 2010. *Data Structures for Statistical Computing in Python.* https://doi.org/10.25080/Majora-92bf1922-00a