A low-resource approach to SemTab 2022

Laurent Mertens[1]

[1]KU Leuven, De Nayer Campus, Dept. of Computer Science
J.-P. De Nayerlaan 5, 2860 Sint-Katelijne-Waver, Belgium
and
Leuven.AI - KU Leuven Institute for AI, B-3000 Leuven, Belgium

Abstract

Mapping tabular data to a Knowledge Graph is a common task that often involves the use of considerable resources in terms of disk space, system memory and processing power. This paper introduces a barebones system that needs only modest computer hardware and only relies on Python, yet still achieves 77.0% F1 and 84.6% F1 on the SemTab 2022 Round 1 CTA and CEA tasks respectively. In its current form, the system can only solve specific kinds of cases, but pointers are provided as to how it could be expanded to become more versatile. Our source code has been made available online.

Keywords

Entity Linking, Cell Entity Annotation, Column Type Annotation, Column Property Annotation, Table to Knowledge Graph

1. Introduction

Storing data in tabular format is popular, as the format combines a clear structure imposed by its rows and columns, with a compact size. This compactness often comes at the price of not clearly describing the properties of, and relations between the elements in the data, which can be a hindrance when attempting to interpret the data correctly. This gave rise to “Tabular data to Knowledge Graph matching” (T2KG), the task of automatically mapping tabular data to appropriate entries in an external knowledge graph (KG) such as Wikidata or DBpedia.

The SemTab challenge was introduced in 2019 [1] as a means of collecting benchmark T2KG datasets and systematically evaluating T2KG systems. This year sees the fourth edition of the challenge, which consists of two tracks: a Dataset track which invites participants to submit new benchmark datasets, and an Accuracy track aimed at evaluating T2KG system performance. The Accuracy track consists of three Tasks: CTA (assigning a semantic type to a column), CEA (matching a cell to a KG entity) and CPA (assigning a KG property to the relationships between two columns). These tasks are organized in three separate rounds, with each round using a more challenging benchmark dataset.

This paper describes our submission to Round 1 of the Accuracy track using Wikidata as

ISWC2022, October 23–27, 2022, Hangzhou, China
laurent.mertens@kuleuven.be (L. Mertens)
0000-0001-5175-2673 (L. Mertens)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3320/paper10.pdf

external KG. In the remainder of this text, we will refer to a Wikidata article, designated by a unique Q-ID, as an *entity*.

The novelty of our approach lies not in its algorithm, which essentially aims at maximizing the property overlap between 1) the cells within a key column and 2) a pair of cells taken from the key column and one other column, and which shares many similarities with existing approaches such as, e.g., [2, 3, 4]. Rather, existing systems either make intensive use of web APIs to query the KG [5, 6] or locally hosted third-party search engines such as Elastic Search [7, 8]. Our goal is not to obtain the highest results, but rather to provide a prototype for an alternative to these complex and resource-intensive systems with a barebones system that is local, fully self-contained and only requires Python. We do not use an external DB API or search engine, but instead opted to create custom data indexes and build our own API on top of those. The results reported in this paper were obtained using a system that ran on a i7-8850H laptop with 32GB of RAM and needed as little as ∼40GB of hard drive space, specs that put the system within reach of edge computing applications. The source code is available at https://gitlab.com/EAVISE/lme/semweb2022.

In what follows we will first sketch the general approach taken to the problem in Section 2. In Section 3 we describe our data pre-processing approach, followed by a detailed description of the table parsing algorithm in Section 4. Results are reported and discussed in Section 5, with pointers for future work given in Section 6, and a summary and final conclusions given in Section 7.

2. Generic algorithm

Our system follows the general approach of first identifying a table column, called *key column*, whose entries can all be mapped onto KG entities and, specifically for Wikidata, these entities share a common category-like property named *instance of*. E.g., in Wikdata, the entity "Robert Smith" (Q491252) is an "instance of" (Wikidata property P31) the concept "human" (Q5). At this stage, it is possible that multiple sets of KG entities match the key column. Subsequently, the system will attempt to match the other columns to properties of the entities in these entity sets, enforcing the constraint that all mapped onto properties for values in a non-key column, if matchable to a set of entities, should share the same property tag. E.g., if the key column identifies entities of category "human", all values in some other column might be matchable as values of the property "age" of the entities the key column is mapped onto. If on the other hand, say, one of the column values can be mapped onto the property "retirement age" instead of "age", whilst all the others can be mapped onto "age", this particular mapping will be considered invalid. The set of key-column entities that allows to best match the other columns is the one that is ultimately put forward as the winning set.

To this end, we need to be able to perform the following KG queries: retrieve candidate entities for a given string; retrieve all "instance of" values of a given entity; and finally, retrieve all RDF triplets pertaining to a specific entity from the compressed KG. In the following sections we explain how we set this system up and use it to parse tables.

3. Data preparation

3.1. Splitting of the Knowledge Graph

The SemTab 2022 challenge uses a Wikidata dump dated 21th of May, 2022. Compressed, the data is 33.4 GB. Uncompressed, this amounts to a considerable 1TB+ of textual data representing RDF triples, which are then typically ingested into some type of structured database such as MongoDB[1], or a framework that supports the SPARQL query language.

We opted to work on the compressed data directly, without using any third party database or search engine, hereby greatly reducing the necessary hard drive space. However, querying the original compressed file directly is not an option, knowing that parsing it to the end (equivalent to fully uncompressing it) takes no less than a couple of days on a typical laptop. To circumvent this issue, we split the original data into chunks of 10,000 articles, each identified by a unique Q-ID, by uncompressing the main file on the fly[2] whilst filling a 10,000 article buffer. Note that the buffersize is customizable, with lower buffersizes resulting in smaller files, and hence, increased query speed later on. Once full, the buffer is compressed to disc, emptied, and filled up again. This results in 10,084 chunks. Besides dividing the data up into manageable chunks, it also adds the benefit of allowing to process the data in parallel later on.

After splitting the data, we perform two extraction phases, each with a different focus:

- First, we extract all unique tokens making up the names of the Wikidata entities. We will come back to what constitutes an entity's name in §3.2.
- Second, we reparse the data, using the unique tokens extracted in the previous step to map the entity names to a numerical representation by replacing each token with its index in the sorted list of unique tokens. Simultaneously, we extract for each entity its "instance of" property (which might have multiple values) and generate a table of contents (TOC) that describes in what chunk and at what position in the chunk each entity's article (i.e., its RDF triplets) is to be found.

In the following subsections, these steps are discussed in more detail.

3.2. Unique token extraction

The goal of this step is to gather a sorted list of unique tokens that make up all the entity names in the database. This list is used as a basis for storing the names in an efficient way by simply storing the indices of the tokens making up the names, instead of the tokens themselves. Efficient index look-up is achieved by using binary search.

Wikidata entity descriptions do not contain an explicit "name" property, so we heuristically assembled a list of properties that are considered namelike by browsing through a list of all occuring properties sorted by occurence frequency, and inspecting typical values for some random entities. We selected the Wikidata properties (P-IDs in parentheses) "Wikimedia commons category" (P373), "official name" (P1448), "short name" (P1813),

[1] https://www.mongodb.com/

[2] So we still fully uncompress the data once, but we do not save the uncompressed data to disk.

"label" (P2561) and "kbpedia id" (8408), as well as the non-Wikidata property "schema name". All values of these properties for a specific entity are interpreted as names of that entity.[3] Potential names are processed as follows:

- If the literal property value contains a '@' value delimiting a language specification (e.g., `"George"@en`), then this language specification is removed.
- Names are filtered using the following RegEx that defines the allowed characters: "`[À-žß\w\s.\-()\[\]{}]+`". Names containing other characters are ignored. Mostly, this aims at filtering out names containing non-Latin characters, with the exception of some special characters such as brackets. This entails that as of now, our system does not support resolving entities using non-Latin names.

Valid names are split into tokens simply by splitting on whitespaces. By repeating this process over all entities and keeping track of all unique tokens, we obtain the desired set of all unique tokens making up all valid names, which is then sorted alphabetically and saved to disc as a binary file, storing one token per line. For the work described in this paper, this amounted to 48,645,067 unique tokens stored in a 916MB file.

3.3. Name, "instance of" and TOC extraction

During the second parse, all entities are again parsed in order to extract their names using the same method as described in §3.2, as well as their "instance of" IDs. Entities for which no valid names can be extracted, either because they do not have any specified names or because their names are filtered out, are ignored. Extracted names are converted to a numerical representation by replacing each constituent token with its index in the sorted list of unique tokens, a process that can be done efficiently using binary search.

Finaly, the list of names and "instance of" IDs for each entity are saved to disk, creating one new file per chunk, using the following binary file format:

- First, 4 bytes are used to store the entity's Q-ID.
- Next are 2 bytes representing the number of extracted names for this entity.
- This is followed by, per name, a sequence of 2 bytes representing the number of tokens in this particular name, followed by (number of tokens)*4 bytes encoding the sequence of token IDs constituting this name.
- After the listing of names, 2 bytes are used to store the number of "instance of" values for this entity, followed by (number of "instance of" IDs)*4 bytes encoding all "instance of" Q-IDs for this entity.

This format allows to very efficienty store all the extracted data. File size varies from chunk to chunk, but is typically in the order a tens of KBs, to a few hundreds of KBs for the larger files.

Simultaneously, a TOC is created by for each chunk appending the relevant information to the single binary TOC file (contrary to the "one file per chunk"-approach for the names), using the following format:

[3] Note that an entity can have multiple values sharing a same property, e.g., more than one "official name".

- First, 4 bytes encode the ID of the chunk file.
- The next 4 bytes encode the number of entities in the chunk.
- This is then followed for each entity by a sequence of 4 bytes encoding its Q-ID, 4 bytes encoding the start position of the data pertaining to this Q-ID in the chunk file and another 4 bytes encoding the length of this Q-ID's data.

So, in order to retrieve the data for a specific Q-ID, one needs to open the corresponding (compressed) chunk file, put the reader head at the starting position described in the TOC and read out the appropriate number of bytes, as described in the TOC. The number of entities left after filtering is 96,809,376. The corresponding TOC file is 619MB.

3.4. A note on multiprocessing

As noted in 3.1, a bonus advantage of dividing up the data in chunks is that it can be processed in parallel. One should be careful however how to approach this in Python. The most straightforward way is to divide the data chunks up into batches, and process each batch using a `multprocessing.Pool` object, but this risks running out of RAM, because each process comes with a full copy of the data contained in the main process that spawns it. Moreover, the time it takes to process a chunk varies greatly from chunk to chunk, with larger chunks[4] taking up considerably more time. As long as a single process is running it will block the entire pool from finishing, leaving the other processes to idle until the entire pool is finished and the next batch of chunks can be processed.

To alleviate both issues at once, we use a double multiprocessing queue system in which one queue is used to launch several parallel processes that each process a single chunk, and each of those processes then puts its results on the second queue which digests the results and writes them to disk. This way, whenever a process has finished parsing a chunk, it can immediately be reused to process a different chunk.

4. Table processing

Processing the CSV tables requires the possibility to query the extracted information described in Section 3 in an online way, i.e., without having to reload the data for each CSV, which would be inefficient to the point of making the system inoperable. To make this possible, we opted to use a local Python-based web server. Once the data is loaded into RAM and can be queried, the CSV tables can be processed. In §4.1 the server setup is described in detail, followed by a description of the table processing algorithm in §4.2.

4.1. Local server setup

As local web server, we use CherryPy[5]. This allows us to load all necessary data into memory, and make it queryable through local URL requests using custom API endpoints

[4] "Larger" in terms of disk space, not in terms of the number of entities. All chunks, except for the last one, describe 10,000 entities. However, the size of entity descriptions varies greatly, with more data being known for some entities than for others, resulting in varying chunk sizes.

[5] https://cherrypy.dev/

and Python's built-in `urllib` package.

Loading the data into RAM happens as follows.[6] The process might look a bit convoluted, but essentially, it all revolves around storing data into RAM in such a way that it can be conveniently and efficiently searched using binary search.[7]

First, we parse all "names and instance of" files created in §3.3, extracting the information contained therein. To make things slightly more efficient, we parse the files in batches, processing all gathered results per batch at once. During the parsing of the files, we fill up two arrays: one containing the entity IDs, and one containing the "instance of" IDs. The important point is that both arrays are filled up in the same order, i.e., position x in the "instance of" array corresponds to the entity stored in position x of the entity IDs array. At this stage, we also write out all unique entity names to a temporary file on disk, together with all the entity IDs to which they correspond (a name can be shared by many entities). These unique files are written away per processed batch of files, and may hence contain duplicates; the names are unique within their respective file batch.

Once all files have been parsed this way, we sort the entity IDs array, and let the "instance of" array follow the same order. This way, one can easily look up the index of a specific entity ID using binary search on the entity IDs array, and use this index to retrieve the entity's "instance of" values in the corresponding array. To make things more memory efficient, we actually use two arrays to store the "instance of" IDs. After sorting, we convert the array of IDs to a contiguous bytearray, and store the ending positions of the data chunks corresponding to each ID in a second array. In other words, position x in the second array tells us at which position in the first (contiguous) array the data chunk corresponding to the entity ID stored in position x in the entity IDs array ends (exclusive). To get the start position of the data chunk, one simply needs to look at position $x - 1$. In case $x = 0$, the start position is also 0. This is represented schematically in Figure 4.1. Similarly, the entity IDs array is converted to a contiguous bytearray, where each ID takes up 4 bytes. Binary search can be performed on this contiguous array by taking positions that are multiples of 4 as pivot points.

Once this part is done, we proceed to collect all unique names saved to the temporary file generated in §3.3, ignoring the saved entities at this point. We essentially apply the same trick as for the "instance of" IDs, where the names, or to be precise their corresponding token IDs, are stored alphabetically in a contiguous array of bytes, with a second "parallel" array containing the end positions of each name. This allows us to use binary search on the names array, using the end positions as pivot points.

Once we have this information, we then proceed to parse the temporary file a second time, this time focusing on the entities each name belongs to. From this, we create two mappings: one that maps from each entity ID to the indices in the unique names array of its corresponding names, and inversely a map that for every unique name maps onto those entity IDs to which it belongs. The ordering of both maps follows the ordering of

[6] Note that this approach can be cached, i.e., the resulting Python objects saved to disk, so that they can be loaded directly at later times instead of having to restart the process from scratch.

[7] On a technical note, it is imperative to use Python `bytearrays` and Numpy[8] arrays instead of Python structures such as `lists` and `dictionaries` whenever possible, as the Python structures are simply too memory consuming.

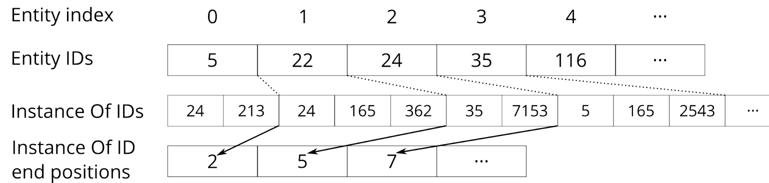

Figure 1: Schematic illustration of indexing mechanism. As an example, to retrieve the "instance of"-ids for entity Q24, first binary search for target "24" is performed on array "Entity IDs", which returns 2. So the start and end positions of the relevant "instance of"-ids are located in the "Instance Of ID end positions"-array at positions $x - 1$ (start) and x (end), with $x = 2$, which gives start=5, end=7. Hence, the "instance of"-ids are [35, 7153].

Table 1
Toy example of a table to be solved.

col 1	col 2
Robert De Niro	Greenwich Village
Maggie Smith	Ilford

the entity ID array and unique names array respectively. Both maps are again stored using the "two parallel arrays"-approach.

The CherryPy API offers four endpoints to query the data:

- An endpoint to query all entity IDs relating to a specific name.
- An endpoint to query all known names relating to a specific entity ID.
- An endpoint to query all "instance of" IDs relating to a specific entity ID.
- An endpoint to request the raw data (i.e., the RDF triplets as stored in the Wikidata dump) for a specific entity ID.

4.2. Table processing proper

Now that all data has been loaded into RAM and we are able to query it using our API, we are ready to process the CSV files.

4.2.1. Finding a key column

First, we attempt to find a key column in the table. For this, we start with the first entry in the first column, and request all known entities for this entry. There is an interal conversion from string to token indices taking place. If the column value can not be mapped onto known token indices, or no entities can be linked to this name, we move on to the next column. If entities are retrieved, we retrieve the "instance of" IDs for each retrieved entity.

We then move on to the next entry in the column, and retrieve its corresponding entities. Again, we construct a set of all the unique "instace of" IDs relating to these entities, and for each entity consider the entities gathered for the previous column entry, and look at

the intersection of the sets of "instance of" IDs relating to both. If this intersection is empty, then we know that these two entities do not share a common "instance of" ID, and we further ignore this sequence of entities. If however the intersection is not empty, we keep this sequence of entities as a potential *explanation* of the column so far.

This process then continues on until the end of the column is reached. All surviving entity sequences are considered potential explanations of the column. If the end of the column was not reached or no sequences survive, we move onto the next column until a key column is found. If no key column can be found, the table can not be solved.

Considering the toy example depicted in Table 1, "col 1" is a key column for which the cell value "Robert De Niro" can be mapped onto either Q36949 or Q951321 and "Maggie Smith" onto Q172653, Q19666077 and a few others. This results in explanations (Q36949, Q172653), (Q951321, Q172653), (Q36949, Q172653), etc. Each explanation shares the "instance of" ID Q5, which is "human". In words: each explanation represents a tuple of two humans, one named "Robert De Niro", the other named "Maggie Smith". To figure out which of these explanations is the most probable, we need to look at the other information present in the table.

4.2.2. Use the key column to explain the other columns

Assuming a key column and matching explanations were found using the steps explained in §4.2.1, we can move on to the next step: try to match the other columns by checking if we can find the values in these other columns within the data pertaining to the entities in the (explanations for the) key column.

In order to do so, we take as our starting point the explanations previously found. For each of these explanations, we loop over all columns other than the key column, and check that not only can the values in these columns be found back in the raw RDF triplet data for the entities in the explanation, but moreover the matching triplets over all rows (i.e., across all entities in the explanation) share the same predicate. Only then will we consider the column as *solved* by this particular key column explanation. If all non-key columns can be solved this way, we say that the explanation solves the entire table.

The way we go about doing this can be summarized as follows. For each entry in the (non-key) column to be parsed, i.e., for each row, we check whether entities can be found for which this entry is a known name. If so, we also take their corresponding Q-IDs into consideration in what follows. This gives us a set of *targets*: the original cell value plus any possible corresponding Q-IDs.

Next, we retrieve the raw RDF triples (this is essentially a chunk of text consisting of multiple strings, each encoding an RDF triplet) for the entity in the explanation under consideration that corresponds to the current table row. For each target we then check whether we can find it back in the object part of an RDF triplet by means of string matching. For each match, we retrieve the predicate of the triple and add it to the set of predicates over all matched targets for this particular column entry and explanation entity. We then take the intersection of this predicate set with the set of predicates so far shared amongst previously parsed column entries. If this is the first entry to be parsed, its predicate set will serve as starting point for this iterative process. If at some point during

the processing of column entries this shared predicate set becomes empty, this either means that for the currently processed column entry/explanation entity combination no target value could be found in the entity's raw data, or one or more targets could be found, but their predicates do not overlap with the predicates of the previously matched targets for previously processed column entries.

If when having parsed all column entries, starting from a key column explanation, there remain shared predicates amongst the matched targets, the column is solved by the explanation.

To make this all a bit more concrete, let us refer back to the toy example in Table 1. §4.2.1 left us with, a.o., the explanations (Q36949, Q172653), (Q951321, Q172653) and (Q36949, Q172653).

Let us consider (Q36949, Q172653) first. The first entry in column 2 is "Greenwich Village". The targets under consideration are the original cell value "Greenwich Village", but also the string encoded entity IDs "Q205380" and "Q5604897", of which "Greenwich Village" is a known name in our database. We then retrieve all RDF triples for entity Q36949, and try to string match the aformentioned targets. It turns out we find a match for "Q205380", namely in the following triple:

```
<http://www.wikidata.org/entity/Q36949>
<http://www.wikidata.org/prop/direct/P19>
<http://www.wikidata.org/entity/Q205380> .
```

From this triple, we extract the predicate "P19", which is "place of birth", and add it to the set of shared predicates $S = \{\text{P19}\}$. We then turn to the next entry in this column, which is "Ilford". The corresponding entity for this explanation is Q172653. The name "Ilford" can be mapped onto entities Q2297044, Q5997730, Q5997725 and Q5997728, so we have five targets to look for. In the raw triples data for entity Q172653 we find a string match for "Q2297044":

```
<http://www.wikidata.org/entity/Q172653>
<http://www.wikidata.org/prop/direct/P19>
<http://www.wikidata.org/entity/Q2297044> .
```

Hence, the set of predicates for entity/value combination Q172653/"Ilford" is $D = \{\text{P19}\}$. We update the set of shared predicates by doing $S \leftarrow S \cap D$, and are left with the non-empty predicate set $S = \{\text{P19}\}$. In other words, for the explanation (Q36949, Q172653) the values in column 2 can be mapped onto the same property with predicate P19, i.e., "place of birth". Put even differently, it appears that column 2 encodes the place of birth of the entities (Q36949, Q172653). Hence, explanation (Q36949, Q172653) solves column 2, and since column 2 is the only non-key column in this table, also solves the table itself.

Moving on to explanation (Q951321, Q172653), we find that we can not find back any of the "Greenwich Village" targets in the triples for entity Q951321. Hence, explanation (Q951321, Q172653) does not allow to solve column 2. Analogously, we find that all the other explanations are unable to solve column 2.

Table 2
Results for the SemTab 2022 Round 1 CTA and CEA tasks.

Team	CTA			CEA		
	Avg. Prec.	Avg. Rec.	Avg. F1	Avg. Prec.	Avg. Rec.	Avg. F1
Laurent	0.785	0.755	0.770	**0.972**	0.749	0.846
KGCODE-Tab	0.944	0.940	0.942	0.916	0.871	0.893
JenTab	0.940	0.936	0.938	0.946	0.944	0.945
s-elBat	**0.961**	**0.952**	**0.957**	0.964	0.926	0.945
DAGOBAH	–	–	–	0.955	**0.952**	**0.954**
Kepler-aSI	0.944	0.944	0.944	–	–	–
SemInt	0.794	0.794	0.794	–	–	–

5. Results

The results of our system on the SemTab 2022 Accuracy Challenge Track, Round 1 CTA and CEA tasks are shown in Table 2. These results were taken from the official website[9]. We also submitted a submission for the CPA task, but unfortunately our submission contained duplicate rows due to the results being appended to an earlier aborted run, making our submission invalid. The submitted run took 3h50min to complete.

Our system achieved an F1 score of 77.0% on the CTA task and 84.6% on the CEA task. Current system performance is markedly inferior to the top-performing systems (95.7% for CTA, 95.4% for CEA), yet managed to squeeze in a best performance in precision for CEA at 97.2%.

6. Future Work

The biggest drawback of our system at present is its lack of support for fuzzy matching. This means that a table cell entry can only be matched to a KG entry if its value exactly matches a known name of the KG entry. Similarly, cell values in non-key columns are currently only checked literally against the candidate entities' known properties.

In the following subsections we will provide some pointers as to how these issues could be resolved.

6.1. Fuzzy name matching

Adding fuzzy name matching to our system is somewhat tricky, mainly because of the self-imposed limit to have the system be self-contained and run on a 32GB machine. Currently, only about 10GB of free RAM is needed to load all necessary data into memory, but on top of that comes the memory requirements during table parsing. Data pertaining to candidate entities is buffered into memory, which in case of many candidates quickly runs into several GBs of RAM as well, leaving only so much wiggle room.

[9] https://sem-tab-challenge.github.io/2022/

After submitting our competition results, we experimented with using n-gram based matching to allow for fuzzy matching. Two paths were explored, one focused on token matching, the other focused on full string matching. Both paths proved unsatisfactory, albeit for different reasons. We will first discuss both, and present a possible way forward.

The idea behind n-gram based matching is that one creates an index mapping strings onto n-grams, analogous to how indices are created mapping documents to salient terms for document retrieval purposes. Such an index is typically stored in a sparse matrix, with rows representing the indexed terms, and columns the values to be indexed on. For our purposes this is a matrix with rows representing names and columns representing n-grams. To perform fuzzy matching, one then simply has to convert the search query (i.e., the name to be matched) to a vector representing its decomposition in n-grams, and compute the cosine similarity (or other preferred metric) with the index matrix. The matrix rows with the highest cosine similarity values are then considered the best matches to the search query. The issue with this, of course, is the potential size of the index matrix. Specifically, we are dealing with 48,645,067 unique tokens making up 94,714,857 unique names, resulting in gigantic index matrices in terms of memory, even using sparse encoding[10]. This is compounded by the fact that matrix multiplication consumes heaps of memory as well, further limiting the acceptable size of the index matrix.

Our first experiment was indexing the tokens rather than the full names, resulting in a smaller index matrix, and performing fuzzy matching on a per-token basis. This way a query string is first split into tokens, and each token is fuzzy matched against the index. The resulting lists of matches for each token are subsequently combined to form possible candidates. E.g., consider the query string "Robaert Smith". Fuzzy matching would possibly return matches "Robert" and "Robart" for the first token, and some variations on "Smith" for the second token. Potential candidates would then be "Robert Smith", "Robart Smith", and all combinations with the "Smith"-variations. For each of these candidate strings the corresponding KG candidates then have to be retrieved (potentially none; e.g., "Robart" is a known surname, but not a known firstname, so "Robart Smith" would return no results). The main issue with this approach is the quickly exploding number of candidate entities, making this potential solution unworkable.

The second experiment was to index the full names, albeit after heavily filtering them[11]. Moreover, n-grams were binned, such that, e.g., the 100 most common n-grams would map to the same index. Unfortunately, despite experimenting with several n-gram binnings, we did not manage to get the system to work. When entering an exactly matchable query, the top result would be the exact match, as expected, followed by some variations in casing when present. But whenever a query with a typo would be presented, the returned results would not contain the correct name.

A step in the right direction might be to return to the token-based setup, and (quite heavily) filter the result combinations. For starters, one might begin by only keeping those combinations that fall within a certain edit distance from the full query term. This would also allow to greatly increase the speed of the result product taking, as entire

[10]The sparse index matrix for the unique names is ∼10GB.

[11]E.g., we removed all names containing brackets.

branches could be cut off early on. E.g., for any name with more than one token, all combinations whereby the match for the first token already violates the “within edit distance” contstraint can be ignored.

6.2. Fuzzy property matching

To address the issue of not being able to fuzzy match non-key column values, we propose the following approach. When retrieving the raw KG triplets for an entity, the tuples (or at least their predicate/object pairs; the subject is of course the same) should be buffered. At this stage, one could already make a distinction between numerical and textual values. One could imagine a global variance threshold being defined that explicits how much a cell value and KG property value may differ in order to be explored as candidate match. For textual values, this could be a maximum edit distance, potentially as a function of length (i.e., longer strings could be allowed to differ by more than smaller strings), whilst for numerical values the logical choice would be a percentage. Finding candidate matches would then simply be a matter of going through the candidate KG entry’s properties, and seeking out those values that are within the allowed variance bounds from the cell’s value. From there on out, the further processing would remain the same. Specifically, all column value matches should still have the same predicate in order to be accepted.

6.3. Property relations extraction

We actually used a third extraction phase in which property relations were extracted. Concretely, for each article we extracted the values for the following relations (Wikidata P-ID in parentheses): “subclass of” (P279), “said to be the same as” (P460), “subproperty of” (P1647), “related properties” (P1659) and “next lower rank” (P3729). Currently, this information is not used, but it could be used to look for property matches between entities with “instance of” IDs that are related, rather than equal.

7. Conclusion

We presented a low-resource system to map tabular data to Wikidata, and validated it on Round 1 of the Accuracy Challenge Track of SemTab 2022. Our system differs from existing solutions in that it is completely self-contained, relies only on Python and works directly on a compressed Wikidata dump, hence needing only modest computer resources. It achieved 77.0% and 84.6% average F1 on the CTA en CEA tasks respectively. Currently, our system does not support fuzzy matching, putting an inherent limit on obtainable performance. Pointers are provided on how the system could be modified to circumvent this issue. Although we only used our system together with Wikidata, it is in principle usable with any database in RDF triplet format.

8. Acknowledgments

We are grateful to Prof. Joost Vennekens for helpful remarks regarding this paper.

References

[1] E. Jiménez-Ruiz, O. Hassanzadeh, V. Efthymiou, J. Chen, K. Srinivas, Semtab 2019: Resources to benchmark tabular data to knowledge graph matching systems, in: A. Harth, S. Kirrane, A.-C. Ngonga Ngomo, H. Paulheim, A. Rula, A. L. Gentile, P. Haase, M. Cochez (Eds.), The Semantic Web, Springer International Publishing, Cham, 2020, pp. 514–530.

[2] G. Limaye, S. Sarawagi, S. Chakrabarti, Annotating and searching web tables using entities, types and relationships, PVLDB 3 (2010) 1338–1347. doi:10.14778/1920841.1921005.

[3] D. Deng, Y. Jiang, G. Li, J. Li, C. Yu, Scalable column concept determination for web tables using large knowledge bases, Proceedings of the VLDB Endowment 6 (2013) 1606–1617. doi:10.14778/2536258.2536271.

[4] P. T. Nguyen, N. Kertkeidkachorn, R. Ichise, H. Takeda, Mtab: Matching tabular data to knowledge graph using probability models, in: Semantic Web Challenge on Tabular Data to Knowledge Graph Matching 2019, CEUR Workshop Proceedings, 2019.

[5] W. Baazouzi, M. Kachroudi, S. Faiz, Kepler-asi : Kepler as a semantic interpreter, in: Semantic Web Challenge on Tabular Data to Knowledge Graph Matching 2020, CEUR Workshop Proceedings, 2020.

[6] W. Z. G. Z. C. J. T. H. P. W. Xinhe Li, Shuxin Wang, Kgcode-tab results for semtab 2022, in: Semantic Web Challenge on Tabular Data to Knowledge Graph Matching 2022, CEUR Workshop Proceedings, 2022.

[7] J. L. R. T. Yoan Chabot, Thomas Labbe, Dagobah: An end-to-end context-free tabular data semantic annotation system, in: Semantic Web Challenge on Tabular Data to Knowledge Graph Matching 2019, CEUR Workshop Proceedings, 2019.

[8] M. Cremaschi, R. Avogadro, A. Barazzetti, D. Chieregato, E. Jiménez-Ruiz, Mantistable se: an efficient approach for the semantic table interpretation, in: Semantic Web Challenge on Tabular Data to Knowledge Graph Matching 2020, CEUR Workshop Proceedings, 2020.