JenTab: Matching Tabular Data to Knowledge Graphs

Nora Abdelmageed[1,2][0000−0002−1405−6860] and Sirko Schindler[2][0000−0002−0964−4457] ⋆

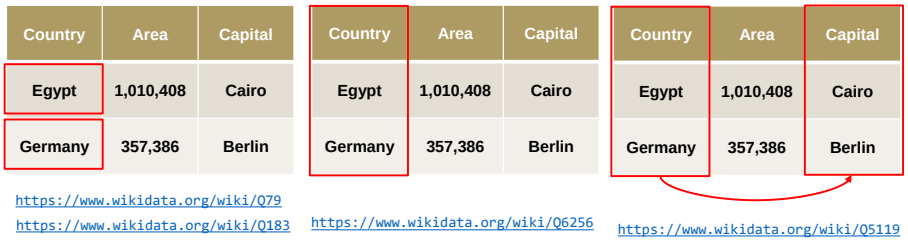
[1] Computer Vision Group
Michael Stifel Center Jena, Germany
Friedrich Schiller University Jena, Germany
[2] Heinz Nixdorf Chair for Distributed Information Systems
Friedrich Schiller University Jena, Germany
nora.abdelmageed@uni-jena.de, sirko.schindler@uni-jena.de

Abstract. A lot of knowledge is traditionally captured within tables using free text entries. Due to the inherent issues of free text like typos and inconsistent naming, integrating that knowledge with other data is seriously hindered. Using semantic techniques to annotate the individual parts of a table can alleviate this task and support access to this vast reservoir of knowledge. However, converting legacy tables into a semantically annotated representation is a non-trivial challenge due to the scarcity of context and the ambiguity and noisiness of the available content. In this paper, we report on our system "JenTab" developed in the context of the Semantic Web Challenge on Tabular Data to Knowledge Graph Matching (SemTab2020). "JenTab" tries to create as much as possible of semantic annotations for table parts. Then, iteratively reduce these candidates by levering different levels of information to reach the most specific solution.

1 Introduction

Tabular data such as CSV files are a common way to publish data and represent a precious resource. However, it is hard to access and use directly due to the associated metadata's frequent lack and incompleteness. Furthermore, the data itself is often noisy and ambiguous. Thus, integrating available tabular data oftentimes becomes a labor-intensive task of cleaning the data and manually mapping across the different representations of identical concepts.

One promising solution often referred to as semantic table annotation, is to exploit the semantics of a widely recognized Knowledge Base (KB). Here, the task is to link individual table components like cells, columns, and their respective relations to resources from KB such as classes (categories), entities (elements), and properties (relations). Achieving such a semantic understanding benefits data integration, data cleaning, data mining, machine learning, and other knowledge discovery tasks.



(a) CEA (b) CTA (c) CPA

Fig. 1: SemTab tasks summary

The Semantic Web Challenge on Tabular Data to Knowledge Graph Matching (SemTab2020)[3] channels the efforts towards this goal. In its second year, it features a series of four rounds. Each round consisting of thousands of raw tables [2] to be annotated with concepts from Wikidata [5]. In each round, participating systems are encouraged to annotate the provided tables and submit their results via AICrowd[4] for evaluation. Annotations themselves are split into three tasks, namely Cell Entity Annotation (CEA), Column Type Annotation (CTA), and Column Property Annotation (CPA).

Given a data table and Wikidata as the target KB, CEA (cf. Figure 1a) links a cell to an entity in KB. CTA (cf. Figure 1b) is the task of assigning a semantic type (i.e., a Wikidata class) to a column. Finally, CPA assigns a semantic relation (predicate) between a column pair in the KB (cf. Figure 1c).

In this paper, we present JenTab, our approach to semantic table annotation. It uses a collection of building blocks that either generate sets of candidates for a given task (create), removes highly improbable candidates (filter), or chooses the most likely one as the respective solution (select). These blocks are arranged in multiple sequences to account for differences in their performance as well as chances of success.

The remainder of this paper is structured as follows. Section 2 describes our pipeline. Section 3 discusses our results for all four rounds where available. Finally, Section 4 concludes the paper and gives an overview of our future work.

2 Approach

We base our approach on collecting mostly independent building blocks for each task that follows Create, Filter and Select (CFS) pattern. The individual building blocks differ in what information they use and how accurate their results are. They further differ in their performance characteristics: On the one hand, this refers to the time needed to execute them. On the other hand, creation and filtration blocks vary in the number of candidates they output.

We maintain sets of candidates on different levels: For each cell, we maintain both the candidates for the respective CEA task as well as those induced by this cell for

[3] http://www.cs.ox.ac.uk/isg/challenges/sem-tab/
[4] https://aicrowd.com/

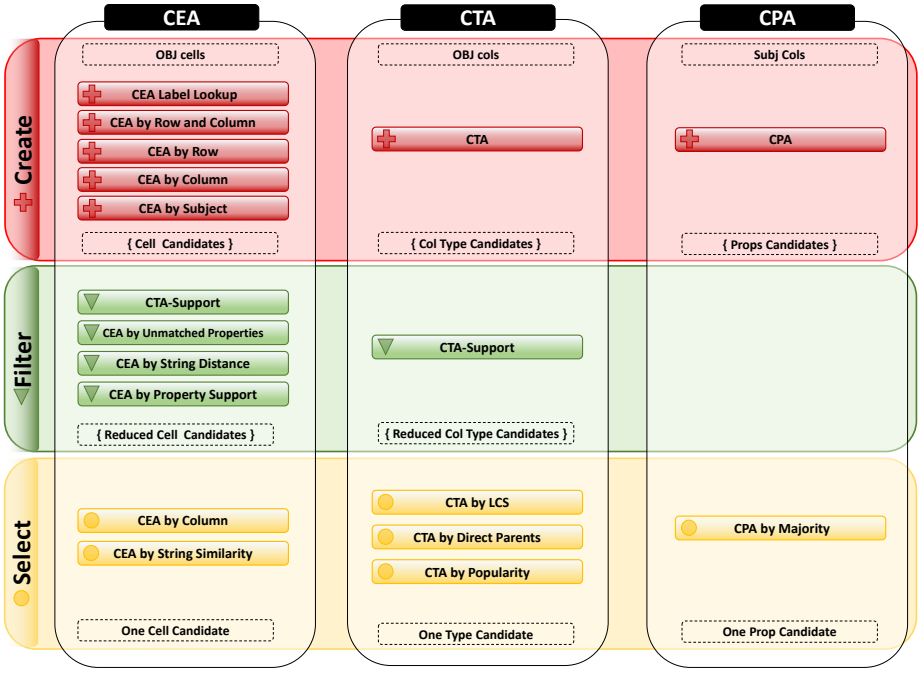


Fig. 2: Pool of building blocks and their assignment to CFS-stage and task. *Create* is indicated in red with a plus icon, *Filter* is represented in green and a triangle sign and *Select* is shown in yellow with a circle symbol.

the corresponding CTA task. For each pair of cells in the same row, there is a set of candidates for the respective property connecting both cells contributing to the CPA task. The same is mirrored on the column level. Here, we keep the onset of candidates for the CTA task and CPA candidates for the combinations of columns.

Building blocks usually pertain to only one task as well as one CFS-stage: *Create* blocks generate candidates of possible solutions for the respective task. *Filter* blocks reduce given sets of candidates. Here, we take a rather conservative approach and only remove candidates that will not be a solution with a very high probability. Finally, *Select* blocks pick a solution of a given set of candidates. Figure 2 summarizes the developed building blocks, which will be described in detail below. Unless noted otherwise, the building blocks only apply to columns of datatype `OBJECT` as classified in the preprocessing. Further, to query the KB we rely on the official SPARQL endpoint of Wikidata[5], which imposes specific rate and execution time restrictions on our queries.

Preprocessing: Before the actual pipeline, each table is run through a preprocessing step. This step has two main goals: First, we apply a series of steps to fix spelling mistakes and other typos in the source data. We start with using *ftfy*[6] to fix any encoding issues encountered. Next, we use a regular expression to split up terms that

[5] `https://query.wikidata.org/`
[6] `https://github.com/LuminosoInsight/python-ftfy`

are missing spaces like in "*1stGlobal Opinion Leader's Summit*". Similarly, we remove certain special characters like parentheses. The result of these steps are stored as a cell's "clean value". Finally, we also apply an off-the-shelf spell checker, *autocorrect*[7], to fix typos resulting in an "autocorrected value" per cell.

Second, we determine the datatype of each column. While the system distinguishes more datatypes, we aggregate to the ones having a direct equivalent in Wikidata, i.e. `OBJECT`, `DATE`, `STRING`, and `NUMBER`. `OBJECT`-columns represent entities, whereas `DATE`- and `NUMBER`-columns correspond to literals. We change the column datatype to `STRING` if it was classified as `OBJECT` but not found among the given targets.

2.1 Create

In the following, we describe different building blocks that generate candidates for individual tasks.

CEA Label Lookup: The Label Lookup is the foundation of the pipeline. It does not depend on any prior information other than the cell's content and their datatype. We apply a series of strategies to retrieve candidates based on the cells' initial, clean, and autocorrected value. As each strategy succeeds in different cases, they are applied in sequence until candidates are retrieved by one. All but the Generic Strategy use the Wikidata Lookup Service[8] to resolve from a given label to Wikidata entities.

- *Generic Strategy* compares the initial cell values with all Wikidata labels and aliases using the Jaro-Winkler distance [6]. We iteratively[9] lower the selection threshold from 1 (exact matches) to 0.9 until a set of candidates is returned.
- *Full Cell Strategy* uses the initial cell values to query the lookup service.
- *All Tokens Strategy* splits a cleaned cell value into tokens removing all stopwords. The lookup service is then queried with all possible orderings of these tokens.
- *Selective Strategy* removes any addition in parenthesis and uses the remainder to query the lookup service.
- *Token Strategy* splits the cleaned cell value into tokens and queries for each token in isolation.
- *Autocorrection Strategy* uses the autocorrected value from the preprocessing to query the lookup service.

CEA by row and column: This approach applies in cases where we could determine candidates for at least some cells of datatype `OBJECT` within a row but failed for the subject cell. Furthermore, for the subject column, existing candidates are required. If all conditions are met, we retrieve entities from the KB that are instances of a subject column candidate and have a connection to at least one of the other candidates in the same row. Subsequently, these entities are filtered such that the remaining ones have a connection to each object cell in the same row. Finally, we compute the string distances[10] between the remaining labels and the initial cell value and discard all that exceed a certain threshold. Finally, we add the remaining entities as candidates to the subject cell in question.

[7] `https://pypi.org/project/autocorrect/`

[8] https://www.wikidata.org/w/api.php

[9] To speed up the process, we use a many-to-many implementation for calculation [3].

[10] Here, we use the Levenshtein distance [4] as implemented by edlib (`https://pypi.org/project/edlib/`).

CEA by row and **CEA by column**: These two approaches work in a similar fashion as *CEA by row and column*, but drop one of its preconditions respectively. In *CEA by row*, a candidate does not have to be an instance of the current column's CTA-candidates. On the other hand, we apply *CEA by column*, when there are no other cells of datatype `OBJECT` in the same row, or those cells have no candidates either.

CEA by subject: This approach is the inverse to *CEA by row*. Assuming that the subject cell of a row has a set of candidates, but any other cell of datatype `OBJECT` does not, it will fetch all entities from the KB that are directly connected to a subject cell candidate. We filter the resulting entities again by their string-distance to the initial cell value before adding them as candidates.

CTA: To find candidates for each column of datatype `OBJECT`, we first retrieve the types for each cell individually based on its current list of CEA-candidates. Here, a type denotes a combination of *instanceOf* (P31) and *subclassOf* (P279) relations. Following experiences in earlier rounds, we also include the sequence P31/P279. Such that, it provided correct candidates that would otherwise have been missed. The column's type candidates are then given by the set of types appearing for at least one cell in this column.

CPA: Candidates for column pairs' relations are generated by first retrieving candidates for the connections between cells of each row. We assume that there is a single subject column; thus, all other cells have to be connected in some way to the cell of that column.

First, we retrieve all properties for a subject cell's candidates, including both literal and object properties. Second, we try to match individual properties to the values of other cells in the same row. If we have found a match, we add the respective property as a candidate for the corresponding cell pair. Object properties are rather easy to match. Here, we rely on previously generated CEA candidates of the respective target and merely compare those with the object properties retrieved.

On the other hand, literal properties require more care. For them, we only consider matches to a cell whose datatype has been determined as either `DATE`, `STRING`, or `NUMBER` in the preprocessing. If we can not establish an exact match for a cell's value, we resort to fuzzy matching strategies depending on the corresponding datatype. For `DATE`-properties (RDF-type: `dateTime`), we try parsing the cell value using different date format patterns. If the parsing succeeds and both dates share the same day, month, and year, we consider this a match. We omit time and timezones for this comparison. In case of `STRING`-properties (RDF-type: `string` and `langString`), we extract words from the given value and the retrieved Wikidata label. Then, we count how many words are overlapped between the two string values. We consider a match if the overlap is above a certain threshold. For `NUMBER`-properties (RDF-type: `decimal`) we tolerate a 10% deviation to still be considered a match according to Equation 1. Such that, *cell_value* is the table cell value and *property_value* is the retrieved property label.

$$Match = \begin{cases} true, & \text{if } |1 - \frac{cell\_value}{property\_value}| < 0.1 \\ false, & \text{otherwise} \end{cases} \tag{1}$$

Once candidates for each pair of cells are determined, we aggregate them to retrieve candidates on the column-level. This initial generation corresponds to the union of all candidates found on the row-level for cells within the respective columns.

2.2 Filter

Once we generate candidates for a particular task's solution, we apply filter-functions to sort out highly improbably candidates. For create-functions depending on previously generated candidates, this can substantially reduce the queries required and overall running time.

CTA-support: This filter works separately on each column of datatype `OBJECT`. First, it calculates the support of all current CTA candidates concerning the cells of a column. A cell supports a given CTA candidates if any of its current candidates has the corresponding type[11]. This filter neglects all cells that are either empty or have no CEA-candidates at the moment. Next, we remove all CTA-candidates from the respective column that do not have support by at least 50% of the cells in this column. Finally, we remove all CEA-candidates from the corresponding cells, which have no types in the remaining CTA-candidates.

CEA by unmatched properties: After generating the properties for all cells on a row-level, some CEA-candidates will have no connection to any other cells in the same row. This filter removes these candidates, leaving only those that have a connection to at least one cell in their respective row. It applies to all cells of datatype `OBJECT`.

CEA by property support: This filter applies only to subject cells. We compute the support of a candidate as the number of cells in the same column it can be connected to[12]. We determine the maximum support for each of a cell's candidates and remove all those with lower support.

CEA by string distance: Some create-functions generate a relatively large number of CEA-candidates. This filter reduces that number by removing candidates whose label is too distant from the initial cell value. We rely on a normalized version of the Levenshtein distance [4], which uses the length of the involved strings as a normalizing factor. To keep any valid candidate, we resort to a rather conservative threshold, thus retain all candidates with a value of at least 0.5 for any of its labels.

2.3 Select

The challenge calls for a specific solution for each task. Hence, at some point, we have to select a single entry from the remaining candidates. As some of the methods below cannot distinguish between the candidates in a certain situation, we apply them in sequence until we find a solution. If only one candidate is left after the previous filter steps, we pick it for an obvious reason.

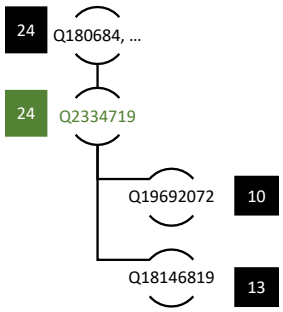
CEA by string similarity: For the remaining candidates, we calculate the string distance to the original cell value using the Levenshtein distance [4]. If there are multiple candidates with the same distance, we break those ties using the "popularity" of the respective candidates. We define popularity as the number of triples the respective candidate appears in them. The intuition is that if there was no other way to distinguish among the candidates in previous filter-steps, using the popularity results in the highest probability of selecting the correct candidate.

[11] Kindly refer to the definition of "type" in this context as given in the generation of CTA-candidates before.

[12] For non-subject cells, this value can only be of 0 or 1, depending on whether they could be matched to the respective subject cell. This case is already covered in a different filter and thus excluded here.

Col0	P31
Spaziano . Florida	Q18146819, Q19692072
Smith v/ Maryland	Q18146819, Q19692072
SEC v. Texas Gulf Sumphur Co.	Q2334719
Reieer v. Thompso	Q18146819, Q19692072
Reed v. Pennsylvania Railroad Compan|	Q18146819, Q2334719
Building Service Employees International Union Local 262 v/ Gazzam	Q18146819, Q19692072
Ramspeck v. Federal Trial Exainers Conference	Q18146819, Q2334719
Budk v. California	Q18146819, Q19692072
Cowma Dairy Company v. United States	Q18146819, Q2334719
Noswood v. Kirkpatrick	Q18146819, Q19692072
Mongomery Building & Construction Trades Council v. Ledbetter Erection Company	Q18146819, Q19692072
Southern Pacfic Company v. Gileo	Q18146819, Q19692072
Colgate-Palmolive-Peft Company v. National Labor Relations Board	Q18146819, Q19692072
Unitee States v. United States Smelting Refining	Q18146819, Q19692072
Poizzi v. Cowles Magazies	Q18146819, Q19692072

(a) Cell values and corresponding types.



(b) Hierarchy of retrieved types.

Fig. 3: Example for CTA selection by LCS, the selected type is highlighted in green.

CEA by column: Sometimes filter-functions accidentally remove the correct solution from consideration. As those cases are quite rare, thus affects a small number of cells in a column. Further, the value of non-subject cells is often not unique throughout their column. In case no candidate is left for a cell, this function looks for occurrences of the same value in other cells of the same column. If we find a match, we apply their solution to the current cell.

CTA by Least Common Subsumer (LCS): The candidates for the CTA task do not stand in isolation but are part of a class hierarchy. Given an example of *Court Cases* in R3, cell types are shown in Figure 3a. We first expand this hierarchy for all remaining CTA-candidates, as shown in Figure 3b. Next, we compute the support for all candidates similar to the respective filtering-function. We remove all candidates with support less than the maximum. We choose the one with the longest path from the root node of the hierarchy as a solution from the remaining candidates.

CTA by Direct Parents: This function selects the CTA by a majority vote. It will fetch the type for all remaining CEA-candidates of a column and then select the one appearing most often. In contrast to the previous definition of type, it only considers the direct connections of an entity, i.e. *instanceOf* (P31) and *subclassOf* (P279) but not their combination (P31/P279).

CTA by Popularity: In case the other methods failed to produce a result due to ties, this function will break those ties by using the candidates' popularity. Again, this is given by the number of corresponding triples the candidate appears in the entirety of the KB. As there is no semantic justification for this selection method, it is only used as a matter of last resort.

CPA by Majority Vote: We compute the support for a given CPA-candidate. Here, this refers to the number of remaining cell-candidate-pairs that use the respective property. We subsequently select the candidate with the highest support as a solution.

2.4 Sequence of Execution

Figure 4 shows the current state of our pipeline. However, this is only a snapshot and subject to repeated additions and adaptations. The current order of blocks is the result of experimentation using the available input tables as a source. After each run, we scanned the results for tables lacking some mappings and investigated causes and possible solutions. We aggregate the individual building blocks into groups for the sake of brevity in the following description.

Group ① forms the core of our pipeline and is responsible for most of the mappings. Based on the *CEA Label Lookup*, it generates candidates for all three tasks and removes only the most unlikely candidates from further consideration.

Group ② represents a first attempt to find missing CEA-mappings based on the context of a row and a column. Be kindly reminded that all create-blocks only work on cells that currently have no candidates attached. So both blocks here only apply to cells that got no mappings from Group ①. *CEA by Row and Column* is put before *CEA by Row*, as it relies on a narrower context and thus will provide more accurate results. However, it might fail, e.g., when the current CTA-candidates do not include the proper solution. In such cases, *CEA by Row* will loosen the requirements to compensate. If either of these attempts provided new candidates, we re-execute the creation of CTA and CPA candidates afterwards in Group ③.

Group ④ is our first attempt at selecting solutions. After another filtering step on CEA-candidates using the row context, we continue to select high-confidence solutions. As hinted before, this might fail to produce proper mappings for a fraction of CEA-targets. Groups ⑤ and ⑦ try to fill in the gaps for at least some of them. If new candidates are found, groups ⑥ and ⑧ will filter and select from them.

Finally, Group ⑨ represents our means of last resort. Again, they only apply to targets that for which we could not generate a solution before. Here, we assume that not only parts of the context are wrong, but doubt every part of the context. The used blocks will reconsider all candidates discarded in the previous steps and attempt to find the best solution among them.

3 Experiences and Results

The different strategies building blocks and strategies described before reflect our continuous efforts in improving the system. The chosen modular approach allows us to easily exchange individual components or provide backup solutions if the existing failed to account for specific situations. A prime example is the evolution of strategies for retrieving CEA-candidates based only on a cell's content. For example, when we faced spelling mistakes in datasets. We started by using only off-the-shelf spellcheckers. However, their results were not reliable and failed in particular for proper names. We used Levenshtein-distances using a set of all Wikidata labels, which turned out to be time-consuming. The Jaro-Winkler-distance and, in particular, the used implementation allows us to compare individual labels against a multitude of values quickly. However, it overemphasizes differences at the beginning of the string, which caused it to fail for some labels. The presented solution is the current status of our efforts keeping in mind both the effectiveness as well as the resource consumption of the respective strategies.

A reoccurring source of issues was the dynamic nature of Wikidata. Users enter new data, delete existing claims, or adjust the information contained. On several occasions, we investigated missing mappings of our approach only to find that the respective

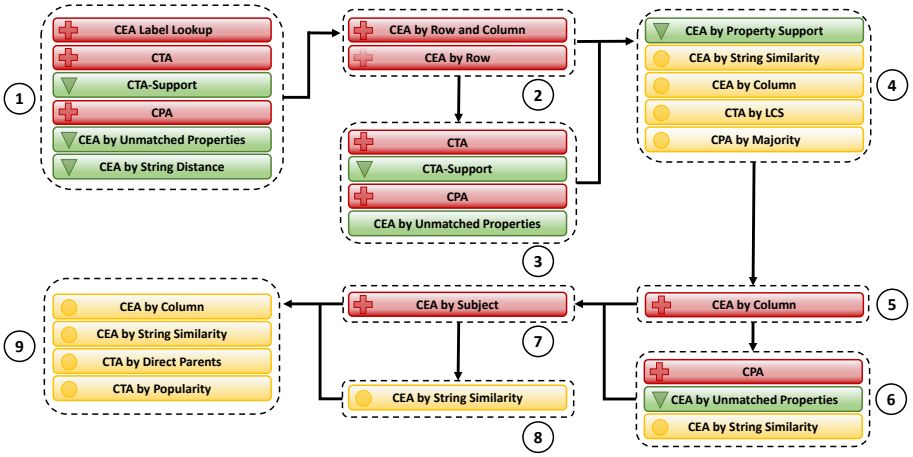


Fig. 4: JenTab: Arrangement of building blocks into a pipeline.

Table 1: Primary and Secondary scores for JenTab at SemTab2020-challenge

	CEA		CTA		CPA	
Rounds	F1-Score	Precision	AF1-Score	APrecision	F1-Score	Precision
Round 1	0.828	0.906	0.574	0.626	-	-
Round 2	0.953	0.969	0.904	0.906	0.955	0.965
Round 3	0.935	0.952	0.859	0.863	0.917	0.928
Round 4	0.973	0.975	0.930	0.930	0.994	0.994
2T	0.374	0.541	0.624	0.669	NA	NA

entity in Wikidata had changed. The challenge and ground truth were created at one point in time, so using the live system will leave some mappings unrecoverable.

Table 1 reports our results as reported by AIcrowd. As our corresponding implementation was not ready in time, we could not submit a CPA solution in the first round. We discovered at a later stage that the CTA solution has a bug in R1, we missed the P31 from the query, we have retrieved only the generic parent. That's why we have a lower score in this task. One more difference about R1, it does not include feedback from other tasks (filter concept). For example, the CEA solution relies only on the string similarity between the cell value and the retrieved label and not considering the semantic context by column types or properties. Moreover, we considered Autocorrect since R3, for tackling the spelling mistakes in the dataset. We have computed the generic lookup database given the unique cell values in the dataset against the Wikidata labels via an optimized Jaro-Winkler Similarity implementation. We expect higher scores if we run the current version of our pipeline on datasets of the rounds again. The last row represents our scores at the Tough Tables [1] known as (2T), 180 special tables are added to Round 4. Scores are only available for the first two tasks.

4 Conclusions & Future Work

In this paper, we have introduced our contribution to the SemTab2020-challenge, "JenTab". We have tackled three posed tasks, CEA, CTA, and CPA. We base our solution purely on the publicly available endpoints of Wikidata, namely the Lookup API and the SPARQL endpoint. Our system relies on the CFS pattern: It generates a pool of candidates for each task, applies various filters given the feedback from other tasks, and finally picks the most suitable candidate.

We see multiple different areas for further improvement. First, we would like to improve the accuracy of the datatype prediction. Here, we still see some misclassifications, especially regarding the `DATE`-datatype. Furthermore, certain components currently require substantial resources, either due to the number of computations necessary or to a lacking performance of the SPARQL endpoint. While we can address the latter by rewriting queries or re-designing the approach, the former offers plenty of opportunities to accelerate the system.

Acknowledgment

The authors thank the Carl Zeiss Foundation for the financial support of the project "A Virtual Werkstatt for Digitization in the Sciences (P5)" within the scope of the program line "Breakthroughs: Exploring Intelligent Systems" for "Digitization - explore the basics, use applications". We would further like to thank the following people for the fruitful discussions throughout the challenge: Kobkaew Opasjumruskit, Sheeba Samuel, and Franziska Zander. Last but not least, we would thank Birgitta König-Ries and Joachim Denzler for their guidance and feedback.

References

1. Cutrona, V., Bianchi, F., Jiménez-Ruiz, E., Palmonari, M.: Tough Tables: Carefully Evaluating Entity Linking for Tabular Data (Nov 2020). https://doi.org/10.5281/zenodo.4246370
2. Hassanzadeh, O., Efthymiou, V., Chen, J., Jiménez-Ruiz, E., Srinivas, K.: SemTab 2020: Semantic Web Challenge on Tabular Data to Knowledge Graph Matching Data Sets (Nov 2020). https://doi.org/10.5281/zenodo.4282879
3. Keil, J.M.: Efficient bounded Jaro-Winkler Similarity based search. BTW 2019 (2019). https://doi.org/10.18420/BTW2019-13
4. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Doklady. Akademii Nauk SSSR **163**(4), 845–848 (1965)
5. Vrandečić, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Communications of the ACM **57**(10), 78–85 (sep 2014). https://doi.org/10.1145/2629489
6. Winkler, W.E.: String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. (1990)