LinkingPark: An Integrated Approach for Semantic Table Interpretation

Shuang Chen[1,2⋆], Alperen Karaoglu[3⋆], Carina Negreanu[3], Tingting Ma[1,2⋆], Jin-Ge Yao[2], Jack Williams[3], Andy Gordon[3], and Chin-Yew Lin[2]

[1] Harbin Institute of Technology, Harbin, China
[2] Microsoft Research Asia, Beijing, China
[3] Microsoft Research Cambridge, Cambridge, UK
{t-shuche, t-alkara, t-caneg, v-tinma, jinge.yao, t-jowil, adg, cyl}@microsoft.com

Abstract. In this paper, we present LinkingPark, our system for Semantic Web Challenge on Tabular Data to Knowledge Graph Matching (SemTab 2020). LinkingPark is an integrated approach for semantic table interpretation. Our system includes a cascaded pipeline for candidate generation, an iterative coarse-to-fine entity disambiguation algorithm, a multi-pass property linking algorithm, and a type inference algorithm tackling the issue of loose ontology in Wikidata. Results on SemTab 2020 demonstrate the effectiveness of our approach.

1 Introduction

Semantic table interpretation could be realised by adding annotations over structured data based on the semantic knowledge stored in a knowledge base. Aiming at benchmarking systems built for this purpose, the 2020 edition[4] of the Semantic Web Challenge on Tabular Data to Knowledge Graph Matching (SemTab-2020) consists of three sub-tasks: Column Type Annotation (CTA), Cell Entity Annotation (CEA) and Columns Property Annotation (CPA).

Formally, the specified input is a relational *table* $T = \{\{t_{11}, \ldots, t_{1n}\}, \ldots, \{t_{m1}, \ldots, t_{mn}\}\}$ that contains a matrix of m rows and n columns of cells, where $r_i = \{t_{i1}, \ldots, t_{in}\}$ denotes the i-th row and $c_j = \{t_{1j}, \ldots, t_{mj}\}$ denotes the j-th column. The content of each cell is a string, denoted as t_{ij}, which is usually used as a textual mention of a named entity and henceforth called a *entity mention*. The knowledge base (KB) in this paper can be described with $(\mathcal{E}, \mathcal{T}, \mathcal{P}, \mathcal{F})$. Here, $\mathcal{E} = \{e_1, \ldots, e_{|\mathcal{E}|}\}$ is the set of all *entities* in the KB. $\mathcal{T} = \{\tau_1, \ldots, \tau_{|\mathcal{T}|}\} \subseteq \mathcal{E}$ is the set of *types* in the KB. The types are connected with the `subclass of` relation to form a *type ontology*. $\mathcal{P} = \{p_1, \ldots, p_{|\mathcal{P}|}\}$ is the set of possible *properties* to

⋆ Work conducted during Shuang and Tingting's internship at Microsoft Research Asia and Alperen's AI residency program at Microsoft Research Cambridge.

[4] http://www.cs.ox.ac.uk/isg/challenges/sem-tab/2020/index.html

describe key attributes of an entity. $\mathcal{F}$ is the set of *facts* which consists of a set of RDF triples $\langle s, p, o \rangle$, where s denotes a *subject* (an entity $e \in \mathcal{E}$), $p \in \mathcal{P}$ is a *property* (also known as *predicate* or *relation*) and o denotes an *object* (an entity e, or a data value, e.g. number, time, string etc.). The target knowledge base of SemTab-2020 is Wikidata.[5]

The three matching tasks of SemTab 2020 can be described as:

- CEA (Cell Entity Annotation): to link each entity mention string t_{ij} in table T to its referent entity in $\mathcal{E}$.
- CTA (Column Type Annotation): to associate a table column c_j with an entity type $t \in \mathcal{T}$. A column may be described by multiple types and the most specific one is usually preferred.
- CPA (Columns Property Annotation): to associate a pair of columns, c_s and c_t with a property $p \in \mathcal{P}$.

2 Approach

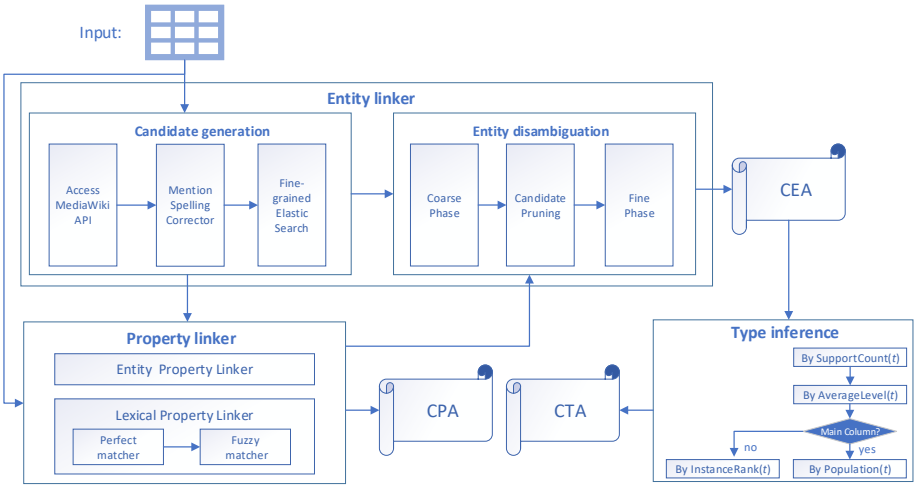

Fig. 1. Overall framework of LinkingPark.

Fig. 1 illustrates the overall framework of LinkingPark. The input table is passed to the *entity linker* and the *property linker*. The entity linker generates candidate entities through a cascaded pipeline that becomes the input for both the entity disambiguation module and the property linker. The entity disambiguation module adopts a coarse-to-fine algorithm to output the annotation of entities. We also integrate the property features from the property linker into

[5] http://wikidata.org/

the entity disambiguation module to characterise the relatedness among different rows. Finally, we design a heuristic multi-pass sieve method for type inference based on the linked entities. Next, we describe each component in detail.

2.1 Entity linker

The entity linker is implemented with a typical approach that consists of two sub-modules: *candidate generation* and *entity disambiguation*.

Candidate generation Given an entity mention t_{ij}, we generate its candidate entities $E_{ij} = (e_{ij1}, \ldots, e_{ijk})$ through a cascaded pipeline which includes three core steps:

- **Accessing Wikidata MediaWiki API:** we start by accessing Wikidata MediaWiki API[6]. We set the largest number of candidates returned from this API to be 50.
- **Correcting the spelling errors**: The MediaWiki API does not handle spelling errors. Following the design principles of a typical spelling corrector[7], we implement a tailored mention spelling corrector for better candidate retrieval. Specifically, the corrector checks all strings within one edit distance to the original mention string, then retains the strings among the set of Wikidata entity titles as candidates. This step is not intended for mentions with multiple spelling errors due to the exponential complexity in the length of edit distance.
- **Searching using fine-grained Elastic Search:** In addition, we build a fine-grained Elastic Search index using all entity titles of Wikidata. The Elastic Search uses a weighted combination of word-based BM25 score and trigram-based BM25 score to do fuzzy matching. This step can improve the recall of candidate generation, but may also return more false positive candidates compared with the first two steps.

Entity disambiguation Given an entity mention t_{ij} along with its candidate list $E_{ij} = (e_{ij1}, \ldots, e_{ijk})$, the entity disambiguation stage aims to select the correct entity $\hat{e}_{ij} \in E_{ij}$ from its candidate list based on their contextual information.

Formally, given a table $T = \{\{t_{11}, \ldots, t_{1n}\}, \ldots, \{t_{m1}, \ldots, t_{mn}\}\}$, the objective of entity disambiguation is to find the most compatible entity assignment for each cell t_{ij}:

$$\underset{e_{11}, e_{12}, \ldots, e_{mn} \in E_{11} \times E_{12} \times E_{mn}}{\operatorname{argmax}} g(e_{11}, e_{12}, \ldots, e_{mn}|T). \tag{1}$$

where $g(e_{11}, e_{12}, \ldots, e_{mn}|T)$ is the function measuring the compatibility score of entity assignments in table T.

[6] https://www.wikidata.org/w/api.php?action=help&modules=wbsearchentities
[7] https://norvig.com/spell-correct.html

Algorithm 1 coarse-to-fine disambiguation algorithm

Input: Table T with candidate lists $\{E_{ij}\}$ and parameters $\{\alpha, \beta, \gamma\}$
Output: Entity assignments $\{\hat{e}_{ij}\}$

1: Initialize $e_{ij}^0 = \text{argmax}_{e \in E_{ij}} \gamma \cdot \text{edit\_dist\_sim}(e, t_{ij}) + (1 - \alpha - \beta - \gamma) \cdot ps(e|t_{ij})$
2: **while** t < max_iter and any of the entity assignments have changed **do**
3: $\quad \text{s}_{\text{col}} = \frac{1}{m-1} \sum_{k=1;k\neq i}^{m} \text{coarse\_ent\_sim}(e, e_{kj}^{t-1})$
4: $\quad \text{s}_{\text{row}} = \begin{cases} \frac{1}{n-1} \sum_{k=1;k\neq j}^{n} \max(\text{f}_{\text{lexical}}(e, t_{ik}), \text{f}_{\text{entity}}(\{e\}, E_{ik})) & \text{if j=0} \\ \text{f}_{\text{entity}}(E_{i0}, \{e\}) & \text{else} \end{cases}$
5: $\quad s_{ij}(e) = \alpha \cdot \text{s}_{\text{col}} + \beta \cdot \text{s}_{\text{row}} + \gamma \cdot \text{edit\_dist\_sim}(e, t_{ij}) + (1 - \alpha - \beta - \gamma) \cdot ps(e|t_{ij})$
6: **end while**
7: Prune candidates based on $s_{ij}(e)$
8: **while** t < max_iter and any of the entity assignments have changed **do**
9: $\quad \text{s}_{\text{col}} = \frac{1}{m-1} \sum_{k=1;k\neq i}^{m} \text{fine\_ent\_sim}(e, e_{kj}^{t-1})$
10: $\quad \text{s}_{\text{row}} = \begin{cases} \frac{1}{n-1} \sum_{k=1;k\neq j}^{n} \max(\text{f}_{\text{lexical}}(e, t_{ik}), \text{f}_{\text{entity}}(\{e\}, \hat{E}_{ik})) & \text{if j=0} \\ \text{f}_{\text{entity}}(\{e_{i0}^{t-1}\}, \{e\}) & \text{else} \end{cases}$
11: $\quad s_{ij}(e) = \alpha \cdot \text{s}_{\text{col}} + \beta \cdot \text{s}_{\text{row}} + \gamma \cdot \text{edit\_dist\_sim}(e, t_{ij}) + (1 - \alpha - \beta - \gamma) \cdot ps(e|t_{ij})$
12: **end while**
13: $\hat{e}_{ij} = e_{ij}^t$
14: **return** $\{\hat{e}_{ij}\}$

Since the exact inference of the above objective is NP-hard, we adopt the framework of an Iterative Classification Algorithm (ICA) [1] for approximate inference. ICA is an iterative local search method which greedily re-assigns each cell to the entity that maximises the probability conditioned on the current entity assignments of other cells. The main assumption behind the design of the disambiguation model is to characterise: (1) *type consistency* along each column of entities, and (2) *property relatedness* within each row of attribute values. In other words, entities mentioned in the same column should have compatible types, while entities or values mentioned in the same row (henceforth describing the same entity) should be related via relational facts and satisfy lexical constraints. Specifically, our model includes a coarse-grained phase which tries to filter out type-incompatible candidates and a fine-grained phase which selects the best candidate by considering more fine-grained property values. The pseudo-code of the disambiguation procedure is shown in Algorithm 1, which can be described as the following four steps:

1. **Initialization** (line 1): Let e_{ij}^t be the cell t_{ij}'s entity assignment at iteration t. Initially, the entity assignments for all cells are independently set by maximising local scores for each specific cell (line 1). The score is a weighted combination of the string similarity between the cell text and the title of the candidate entity ($\text{edit\_dist\_sim}(e, t_{ij})$[8]) and a prior score $ps(e|t_{ij})$. The prior score $ps(e|t_{ij})$ is calculated as $ps(e|t_{ij}) = \frac{1}{rank_e}$, where $rank_e$ is the ranking index (starting at 1) of the entity e in its candidate list E_{ij}.

[8] Implemented using the `Levenshtein.ratio` function in Python

2. **Coarse-grained phase** (lines 2-6): During the coarse phase, the candidate entity's score $s_{ij}(e)$ is a weighted combination of column support score $\mathrm{s_{col}}$, row support score $\mathrm{s_{row}}$, string similarity $\mathrm{edit\_dist\_sim}(e, t_{ij})$ and prior score $ps(e|t_{ij})$. The column score $\mathrm{s_{col}}$ is calculated by averaging the entity similarity between the current candidate entity and each of the remaining cells' entity assignments in the same column of the previous iteration (e_{kj}^{t-1}). Specifically, we represent each entity as a sparse feature vector where each property and the value of `instance of` (P31) / `subclass of` (P279) properties serve as one feature dimension. Our basic assumption is that the properties of an entity are also a proxy of its type besides the explicit types annotation in the KB. The $\mathrm{coarse\_ent\_sim}(\cdot, \cdot)$ function is implemented by calculating the cosine similarity of the above sparse feature vectors. Obviously, the features are not equally important. We adopt a dynamic method to generate feature weights by considering how the feature is shared along the column and how discriminative it is for disambiguating the current cell. We use something similar to TF-IDF weighing: the *term fraction* of a feature f in a column j denoted by $\mathrm{TF}_j(f)$ is defined as

$$\mathrm{TF}_j(f) = \frac{|\{e_{ij}^{t-1} | f \in e_{ij}^{t-1}, 1 \leq i \leq m\}|}{m}, \tag{2}$$

which is the fraction of entities in the column of last time step consisting of this feature. To avoid the noise of irrelevant features, we set $\mathrm{TF}_j(f) = 0$ if it is lower than 0.5. The Inverse Document Frequency (IDF) of a feature f over one cell T_{ij} is defined as

$$\mathrm{IDF}_{ij}(f) = \log\Big(\frac{|E_{ij}| + 1}{|\{e | f \in e, e \in E_{ij}\}| + 1}\Big) + 1, \tag{3}$$

essentially treating each candidate as a document and measures the IDF over it. Here we adopt a smoothed version of IDF to avoid zero-divisions and zero weights. Finally, a feature over a cell T_{ij} denoted by f_{ij} is defined as

$$f_{ij} = \mathrm{TF}_j(f) \cdot \mathrm{IDF}_{ij}(f). \tag{4}$$

Similar TF-IDF formulations have been used successfully in previous SemTab participants (e.g., the Tabularisi system [7] at SemTab 2019 calculating the ranking score). We adapt this formulation for the ICA framework to calculate pairwise entity similarities by implementing a smoothed version of IDF and prune features with low support to mitigate the noise.
The row score $\mathrm{s_{row}}$ is calculated by extracting the property features at both lexical and entity level. This feature characterises the property relatedness between current candidate entity and the remaining cells in the same row. Specifically, for each cell if it lies in the main column of the table, we will calculate the support score from each remaining cell in the same row. Otherwise, we only consider the support score from the cell in the main column. Given the property distribution from the property linker, the support score

($f_{entity}(\cdot,\cdot)$ or $f_{lexical}(\cdot,\cdot)$) is calculated by first retrieving the possible properties between the current candidate entity and the remaining cells followed by getting the largest confidence in the corresponding property distribution.

3. **Pruning** (line 7): We reduce the search space at the current stage before more fine-grained processing. For each entity we look at the candidates sorted by their final scores. If the difference between the final scores of the top-2 entities is above a threshold min_diff, then we only keep the top-1 candidate. Otherwise, we only keep the top-K candidates plus candidates whose final score is above a certain threshold (min_abs).
4. **Fine-grained phase** (lines 8-12): For some highly ambiguous cases, we need to compare the specific values of a certain property instead of looking at only the appearance of the property fields. For example, for a column of Canadian cities such as ["Kingston", "Montreal"], the system could know that these are cities after the coarse-grained step, but there exist multiple cities named "Kingston". We still have to make a choice between Kingston in Jamaica and Kingston in Canada. In such cases, we have to further consider the specific values of certain key properties, such as `Country = Canada`. In this fine-grained phase we extend the sparse features for calculating entity similarity from all properties to all property values.

2.2 Property linker

For the property linking algorithm, we use the approach presented in the technical report [3]. For every relational column, we start from the strings in the cells and try to generate candidates as described in the previous section for the coarse-grained phase. When the search does not return satisfactory results (for example, none of the strings in the column can be matched to an entity), we usually encounter *numerical properties* which contain numbers or dates and we treat them as special columns.

For columns where we can identify KB entities, we try to find direct matches or matches within a given edit distance with the property values of the entities in the main column. For numerical properties, we try to find direct matches within unit conversion. Once we have a set of matches, each row votes to find a first most-likely property. If we do not reach a certain threshold, or the difference between the top choices is too small, we use a second refinement phase that is more computationally expensive. For numerical properties we have pre-computed a set of characteristic statistics per type (for example, human heights have a certain *range*, *mean* and *standard deviation*). For each given type that can suitably describe the main column, we check which of the pre-computed statistics are best matches for the numerical column that we could not identify. For the SemTab dataset we found that just looking at ranges suffices.

A common issue we encounter for Wikidata is that the entities do not have complete information, i.e. some properties could be missing. For columns where we can identify KB entities, we extend the ranking score by considering the properties of similar entities. If several rows voted for a given property and a given row does not have that property, we want to know if that property is

missing or not applicable. We extend binary scoring, a given property present for a given entity, to a new score in $(0, 1)$ that takes into account how many similar entities do or do not contain the relevant property. We define the most similar entities of a given entity as the set of nearest neighbours (in cosine distance) that share the same type with the given entity in the BigGraph space [4].

2.3 Type inference

Our type inference algorithm is a heuristic multi-pass sieve method that is fully dependent on the entity linking results. To predict the type of column j, we first acquire the entity linking results $E_j = \{\hat{e}_{ij} | 1 \leq i \leq m\}$ from the entity linker. Then we retrieve the entity types $\mathcal{T}(e)$ for each entity $e \in E_j$, where we define $\mathcal{T}(e)$ as the set of all types satisfying the SPARQL expression `?entity wdt:P31/wdt:P279?/wdt:P279? ?types.`, treating the values of `instance of` (P31) and `subclass of` (P279) as the *types* for each entity. Then the goal is to find the most common types shared by most of the entities. To do so, we define the first criterion named SupportCount(t):

$$\text{SupportCount}(t) = |\{e | e \in E_j, t \in \mathcal{T}(e)\}|. \tag{5}$$

We select the type with maximum SupportCount(t), but multiple types may have the same count. In that case, we want to prioritise the most specific one. We design a second criterion named AverageLevel(t) based on the type ontology to characterise the specificity of a type t:

$$\text{AverageLevel}(t) = \text{AVG}(\{h | e \text{ is instance of } t \text{ via a } h \text{ length path}, e \in E_j\}) \tag{6}$$

Since lower distance with respect to the entity nodes indicates a more specific type, we select the type with the minimum AverageLevel(t) to break the above ties. However, this method does not guarantee uniqueness. In practice, we found the following design works well on the SemTab data for tie-breaking. For the main column, we select the type with minimum Population(t) on Wikidata, where

$$\text{Population}(t) = |\{e | t \in \mathcal{T}(e), e \in \mathcal{E}\}|. \tag{7}$$

For relation columns, we select the type with the minimum InstanceRank.

$$\text{InstanceRank}(t) = \text{AVG}(\{r | e \text{ is instance of } t \text{ at } r \textit{ rank}, e \in E_j\}), \tag{8}$$

where *rank* means the position of the type t among the statement group of the `instance of` property.

3 Setup and Results

Accessing the online SPARQL endpoint is very slow given the large amount of data, so we use an offline Wikidata dump (20200525). Our experimental pipeline

starts by calling the MediaWiki API which usually takes 2-3 days for each Round. After we generate the entity candidates, we cache the results and extract the relevant subset of the Wikidata dump. Our multi-threaded Python pipeline takes at most 20-30 minutes for each Round on a Intel(R) Xeon(R) CPU E7-4860 v2 (4 processors) machine. As we do not train the hyper-parameters, we empirically set α to be 0.20, β to be 0.50, γ to be 0.1, min_diff to be 0.30, min_abs to be 0.50 and K to be 2.

3.1 Results

Table 1 shows the performance of our approach on SemTab 2020. For Round 1-3, our system is consistently among the top-3 ranking systems in the leaderboard[9]. Specifically, our approach is among the top-2 for CEA, top-1 for CTA and top-3 for CPA. For Round 4, the evaluation dataset additionally introduced a Tough Tables (2T) subset with an average number of approximately 1,080 rows in each table. This additional complexity makes it almost infeasible to use the original candidate generation scheme that starts from accessing WikiMedia API, with the number of queries significantly increased. We instead try to match the mention strings with the entity titles, and query the online MediaWiki API only for those remaining unmatched mentions. Without the ranking information of entity candidates returned from MediaWiki API which induces our entity prior scoring (Sec 2.1), the performance of our system drops on Round 4, especially on the 2T subset. The above results are indicative of the effectiveness of our approach.

Table 1. Experiment results on SemTab 2020 dataset

	CEA			CTA			CPA		
Dataset	F1	Precision	Rank	AF1	APrecision	Rank	F1	Precision	Rank
Round1	0.987	0.988	1	0.926	0.926	1	0.967	0.978	2
Round2	0.993	0.993	2	0.984	0.985	1	0.993	0.994	2
Round3	0.986	0.986	2	0.978	0.978	1	0.985	0.988	3
Round4	0.985	0.985	2	0.953	0.953	4	0.985	0.988	5
Round4 - 2T	0.810	0.811	3	0.686	0.687	3	-	-	-

4 Discussion

4.1 Synthetic data vs. Real data

The evaluation datasets for the SemTab challenge use synthetic data that is automatically generated from the knowledge base. Although in the generation process various refinement strategies have been adopted to simulate real data, we argue that there is still a significant gap.

[9] https://www.cs.ox.ac.uk/isg/challenges/sem-tab/2020/results.html

– In real data a table expresses the intent of its creator, while synthetic data is generated through a random combination of type compatible entities,
– The spelling errors introduced are not necessarily representative of the errors that a table creator might produce,
– Real data might contain much more entities, types, and relations outside the specified knowledge base, making them more challenging than data synthesised from the knowledge base.

The currently available datasets curated from real-world data are either in small scale [5, 6] or with huge noise as the data is automatically extracted from Wikipedia [2]. In order to make progress in this field, better datasets need to be curated and carefully annotated to compliment the synthetic SemTab data produced in the current way.

4.2 Type ontology in Wikidata

The type ontology in Wikidata is noisy, as we can see from the example in Fig 2. Under an ontology with such complex sub-structures, it is hard to determine the specificity of a certain type. In order to define the CTA task more clearly and more fairly on the Wikidata ontology, further cleaning is required (either manually or automatically) to reach a more reliable structure such as the one curated for DBpedia.

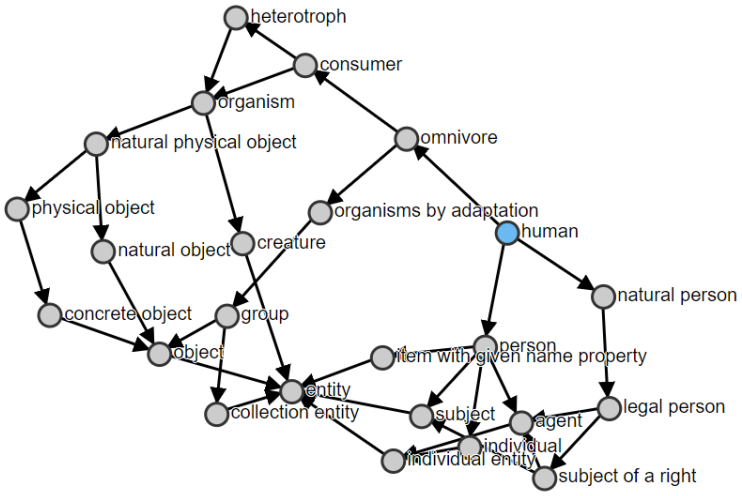

Fig. 2. The graph of subclass of relations for the item Q5 labelled as human and its respective ancestors. Source: https://angryloki.github.io/wikidata-graph-builder/?property=P279&item=Q5.

4.3 Challenge design

Finally, we would like to suggest to split the dataset into a development set and a test set. The test set should be used for final evaluation, while the development set should be released for model design and tuning. This way participants can try to improve their systems without having to make multiple submissions.

5 Conclusion

In this paper, we present LinkingPark, our system for SemTab 2020. Our pipeline with multiple components is an integrated approach for semantic table interpretation. Results on SemTab 2020 demonstrate the effectiveness of our approach for all three tasks. We hope that some parts of our solutions as well as the observations and insights we gathered during the challenge will be beneficial for future research efforts towards better understanding of tabular data.

References

1. Bhagavatula, C.S., Noraset, T., Downey, D.: Tabel: entity linking in web tables. In: International Semantic Web Conference. pp. 425–441. Springer (2015)
2. Efthymiou, V., Hassanzadeh, O., Rodriguez-Muro, M., Christophides, V.: Matching web tables with knowledge base entities: from entity lookups to entity embeddings. In: International Semantic Web Conference. pp. 260–277. Springer (2017)
3. Karaoglu, A., Negreanu, C., Chen, S., Williams, J., Fabian, D., Gordon, A., Lin, C.Y.: Wiki2row - the in's and out's or row suggestion with a large scale knowledge base. Tech. Rep. MSR-TR-2020-37, Microsoft (October 2020), https://www.microsoft.com/en-us/research/publication/wiki2row-the-ins-and-outs-or-row-suggestion-with-a-large-scale-knowledge-base/
4. Lerer, A., Wu, L., Shen, J., Lacroix, T., Wehrstedt, L., Bose, A., Peysakhovich, A.: PyTorch-BigGraph: A Large-scale Graph Embedding System. In: Proceedings of the 2nd SysML Conference. Palo Alto, CA, USA (2019)
5. Limaye, G., Sarawagi, S., Chakrabarti, S.: Annotating and searching web tables using entities, types and relationships. Proceedings of the VLDB Endowment **3**(1-2), 1338–1347 (2010)
6. Ritze, D., Lehmberg, O., Bizer, C.: Matching html tables to dbpedia. In: Proceedings of the 5th International Conference on Web Intelligence, Mining and Semantics. pp. 1–6 (2015)
7. Thawani, A., Hu, M., Hu, E., Zafar, H., Divvala, N.T., Singh, A., Qasemi, E., Szekely, P.A., Pujara, J.: Entity linking to knowledge graphs to infer column types and properties. In: SemTab@ISWC (2019)