*Further with Knowledge Graphs. M. Alam et al. (Eds.)*
*AKA Verlag and IOS Press, 2021*

*doi:10.3233/SSW210033*

# NEST: Neural Soft Type Constraints to Improve Entity Linking in Tables

Vincenzo CUTRONA[a,1], Gianluca PULERI[a], Federico BIANCHI[b], Matteo PALMONARI[a]
[a] *University of Milano - Bicocca, Milan, Italy*
[b] *Bocconi University, Milan, Italy*

**Abstract.** Matching tables against Knowledge Graphs is a crucial task in many applications. A widely adopted solution to improve the precision of matching algorithms is to refine the set of candidate entities by their type in the Knowledge Graph. However, it is not rare that a type is missing for a given entity. In this paper, we propose a methodology to improve the refinement phase of matching algorithms based on type prediction and soft constraints. We apply our methodology to state-of-the-art algorithms, showing a performance boost on different datasets.

**Keywords.** Table to KG matching, Type constraints, Soft constraints

## 1. Introduction

Linking values occurring in a table to entities in a Knowledge Graph (KG) is a critical task for Semantic Table Interpretation (STI). The task is crucial for several downstream applications, including the transformation of tabular data into a KG, question answering over web tables, and KG completion, and it has collected a lot of attention in last years [1,2]. In STI, entity linking is referred to as Cell Entity Annotation (CEA), where other tasks are performed at the same time to annotate columns with entity *types* (CTA) and *properties* (CPA) in order to interpret the table schema into the graph-based model of the reference KG. The CEA, CTA, and CPA tasks are interlinked and can mutually inform each other; for example, entity-level annotations can suggest or provide evidence for type and property-level annotations, while type-level annotations may help disambiguation for entity-level annotations. Approaches addressing all the STI tasks usually implement complex pipelines to collect and propagate the evidence across tasks. A CEA algorithm is expected to disambiguate the textual values in a cell, referred to as *labels* from now on, after retrieving a set of candidate entities from the KG. As a result, the algorithm can decide whether to establish a link for the label, *i.e.*, to annotate the cell with an entity from the KG, or leave it not linked. Different benchmarks have shown that algorithms perform

[1]Corresponding Author. E-mail: vincenzo.cutrona@unimib.it

linking effectively when tables are small and labels are characterized by no or low ambiguity. However, the performance drops dramatically as soon as labels are ambiguous and the tables dimension increases, thus novel datasets have been developed with the objective of challenging algorithms to properly deal with both ambiguity and large tables [2,3]. Algorithms must tackle these challenges and improve their performance in settings covering relevant application scenarios.

In this paper we focus on how the entity type information can be better exploited to support CEA pipelines. Indeed, entity type plays a key role in CEA, with column-wise coherence of the entity types being a characterizing feature of table semantics; however, type information explicitly stored in KGs is known to be imperfect and incomplete [4], but this aspect is often overlooked by CEA algorithms, which assume KGs to be complete [5].

We found two patterns used by CEA algorithms available in the literature that can be improved by better handling entity types: (i) *Filtering by type*, where types associated to a column are used as hard constraints to filter out candidate entities having different types; (ii) *Ranking by distributed entity representations similarity*, where distributed representations of entities (*i.e.*, entity embeddings) are used to compute the similarity between candidates for different labels in order to support the disambiguation. These patterns are, for example, core mechanisms used by a state-of-the-art algorithms such as FactBase, EmbeddingsOnGraph, and their hybrid combinations [6], T2K [7], TableMiner+ [8], but also in more recent approaches tailored on STI challenge data [5,9,10].

We propose to use neural models for type prediction and type representation to improve the above-mentioned mechanisms, by enriching the type information used in existing pipelines in a modular fashion. A first approach, *Type enrichment for filtering by type*, enriches the types of candidate entities with types predicted by a neural network. A second approach, *Type enrichment for ranking by distributed entity representations similarity*, extends entity embedding with distributed representations of types, making similarity more aware of entity types. The two approaches can be combined and we propose to enrich entity embeddings with the type predicted by a neural model. Both approaches capture a similar principle in orthogonal ways, that is, to implement soft type-based constraints to improve entity disambiguation, from which the name NEST was given to the proposed methodology, as further explained in Section 4. As for type prediction, we test two different models using respectively textual descriptions and entity embeddings trained over the KG as input for the prediction.

To test our methodology we conduct experiments on datasets that have been used in previous work, or have been published to make disambiguation more challenging. In our experimental settings, we apply the methodology to a selected pool of state-of-the-art algorithms [6] that we chose because of their performance and because they are more prone to handle large tables, general enough to be applied to different settings and not based on specific assumptions tailored to a specific challenge. However, our novel methodology potentially applies to almost every algorithm that uses a filtering or ranking strategy based on the entity typing. Results suggest that the soft constraint principle significantly improves approaches based on similarity computed using entity embeddings and improves approaches based on filtering by type when the types of the correct entities are not completely coherent (by the construction of the dataset).

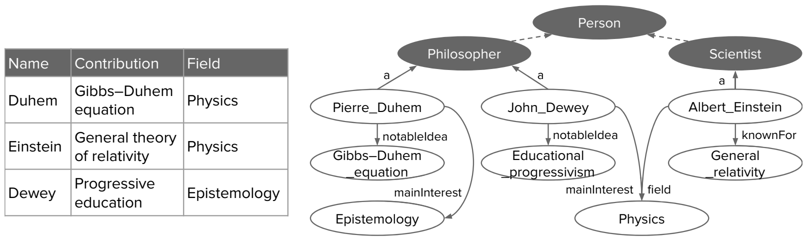
The rest of the paper is organized as follows: Section 3 describes the need for soft type-based constraints for entity disambiguation also introducing the key terminology. Section 4 describes NEST, the two type enrichment strategies and two neural models for type prediction. In Section 5 we evaluate NEST while in Section 2 we discuss related work. In Section 6 we report conclusive remarks.

## 2. Related Work

The growing interest of the Semantic Web community is witnessed by the creation in 2019 of the SemTab challenge [1], aimed at standardizing the evaluation of table annotation algorithms. Among the participant systems, we mention MTab [5], which applied a majority voting strategy to select the best candidate in a pool, Tabularisi [11], which created a feature space from the lookup service results using TF-IDF, and DAGOBAH [9], which is based on entity embeddings and K-means clustering. However, these systems are tailored to the challenge specifications, lacking generality. Indeed, the performance of these approaches are computed without considering the knowledge gap because the list of cells to annotate is given as input within SemTab, disregarding the decision making phase. Moreover, algorithms like MTab relied on a brute force strategy which raises the computational cost, hindering its adoption in real-world scenarios. We remark that FactBase and EmbeddingsOnGraph do not have these shortcomings.

Other unsupervised entity matching approaches adopt an iterative method that combines schema and entity matching. T2K [7] brought outstanding improvements in the state-of-the-art and inspired different systems [8,10]. The disambiguation component in the aforementioned approaches is dependant on the type-based constraints, which are assumed as hard constraints. None of the existing approaches explicitly addresses the problem of relaxing hard constraints, highlighting the novelty of NEST. Moreover, to the best of our knowledge, NEST is the first methodology that focuses on improving the use of types for entity disambiguation using neural models to predict the type of candidate entities. ColNet [12] is a supervised system based on convolutional neural networks to predict column types and it is limited to predicting only 17 types.

The work in [13] used a graphical model and a collective classification technique to optimize a global coherence score for a set of entities in a table. The approach requires collecting tables to train the underlying model. Similarly to our work, the authors tried to remove hard type constraints from the matching process, but they did not exploit the type assertion axioms in the KG, preferring to encode the type into features based on entity co-occurrence statistics. A very recent work demonstrated how language models can be exploited to solve many table understanding tasks [14]. Nonetheless, the authors did not exploit the information contained in the KG and required the use tables during pre-training.



| Name | Contribution | Field |
|---|---|---|
| Duhem | Gibbs–Duhem equation | Physics |
| Einstein | General theory of relativity | Physics |
| Dewey | Progressive education | Epistemology |

**Figure 1.** A table containing data about philosophers of science (1900–1930) from Wikipedia, and a subgraph of the DBpedia KG. Dark nodes are types from the DBpedia Ontology and dashed lines represent subsumption axioms.

## 3. Entity Linking in Tables and Entity Types

The CEA task is defined as in Section 1.

Most algorithms exploit context information to support the disambiguation, looking at values occurring on the same column or row of the cell to annotate. The cell $i, j$ refers to cell in the $i$th row and $j$th column, respectively indicated also as $R_i$ and $C_j$. In STI it is assumed that some columns contain entity mentions while others do not. Some approaches and datasets even use a stronger *one-entity-per-row* assumption, *i.e.*, that only one column contains cells to annotate, which means that each row describes only one entity [7], but this assumption is not realistic in several application scenarios and is discarded in recent datasets [1]. When we are trying to annotate a cell $i, j$ we refer to the column $C_j$ as to the entity column and to all other columns as context columns, we refer to its label as to $label_{i,j}$ (while for sake of clarity we will use *value* to refer to the content of cells in the context columns).

We assume that the reader is familiar with KGs, for which we only introduce the terminology used in this paper. A KG describes entities with a set of (entity-related) facts that we express in a simple predicate logic notation. If $e$ is an entity identifier, $P(e, v)$ or $P(v, e)$ are facts relating $e$ to $v$ with a property $P$, which represents a relation; for $P(e, v)$, $v$ can be an entity identifier, a type identifier (or class identifier) or any literal. The set of *direct types* of an entity is defined by facts stored in the KG. An entity can have zero, one or more direct types. Types are connected by a subclass relation, which supports type inheritance.

To discuss the role that types play in CEA, we use the example depicted in Figure 1. First, we remark the ambiguity of the labels: *e.g.*, if we search for Dewey in DBpedia we retrieve two persons, John Dewey, the philosopher, and a librarian, plus several other entities of different types. Einstein also returns entities of different types.

Second, we can expect type-wise coherence in each column, in this case, philosophers, but we cannot expect that the classification of the table perfectly matches the classification in the KG (in some cases the list of direct types may be simply incomplete). The DBpedia entity `Albert Einstein` is assigned only with the type `Scientist`, even if his thoughts have been appreciated also from a philosophical point of view.[2] Filtering out candidates by using the column

[2]https://en.wikipedia.org/wiki/List_of_philosophers_of_science

type `Philosopher` as a hard constraint will lead to discard the entity `Albert Einstein`. Some algorithms resort to use a more generic type, which is superclass of the types associated with all or some entities (in this case, it would be `Person`), but as a drawback, they have to deal with a much larger set of candidates; for example, 34 entities of type `Person` in DBpedia contains the term "Einstein" in their label, 2 of which are of type `Scientist`.

CEA algorithms usually combine three main operations into complex pipelines:

1. **Candidates retrieval**, where some value in the table, usually the label of the cell to annotate, is matched against the facts of the KG; this operation returns a - possibly empty - set of *candidate entities.*
2. **Ranking**, where candidate entities are ranked according to some criterion, which may combine matching scores (*e.g.*, retrieval function), other scores, filters, and more sophisticated mechanisms.
3. **Decision making**, where the collected evidence drives the decision whether to link or not, and, in the first case, which entity consider as the annotation.

The combination of ranking and decision making are the core of a disambiguation algorithm. Filters over candidates and scoring used in ranking can contribute to decision making: if after filtering the ranked list of candidates is not empty, the top candidate can be selected for the annotation, otherwise the cell is not annotated. Other decision making strategies can use thresholds; however, it is difficult to apply thresholds over scores that are not bound, which is typical for scores returned by matching functions powered by search engines and available *lookup* services; search engines, on the other side, offer very efficient search over the vast amount of information stored in KGs. As a results, *lookup services* usually combine string similarity, document frequencies (*e.g.*, Lucene-based scores), and even other aspects like popularity (the DBpedia Lookup Service[3] exploits entity popularity measures, *i.e.*, inlinks pointing at the candidate). These considerations make filters and scoring particularly relevant. In this paper, we focus on type-based filters, and on the entity similarity used for scoring the candidates. As anticipated in Section 1, we believe that algorithms proposed by [6] provide state-of-the-art solutions in terms of trade-off among generality, performance, and scalability (see Section 2 for a more detailed discussion of related work).

## *3.1. Linking pipelines in FactBase and EmbeddingsOnGraph*

Before explaining how the proposed methodology can support the selected CEA pipelines [6], we prefer to discuss these approaches with sufficient amount of details in such a way to ease the understanding of our methodology and its evaluation, as well as to make this work self-contained, favoring the replicability.

[3]https://wiki.dbpedia.org/lookup

*FactBase* The algorithm works column-wise, *i.e.*, it examines all the cells of an entity column, it exploits direct entity types for filtering on candidates (filter by type) as well as representative language tokens (filter by token), and uses the values in the other columns to match facts in the KG and expand the set of candidates. It implements a pipeline that consists of a preliminary step and three entity annotation steps:

- *Candidates retrieval and schema-level annotation.* All the labels in the entity column are looked up in the index returning, for each label, a list of candidates ordered by the lookup service score. For each label, the algorithm looks at its top ranked candidate to extract (from the KG) its direct types and its textual description, which is processed (*e.g.*, stop words are discarded) to return a set of tokens. Given all the entity types and description tokens extracted for all the labels in the column, the k-most frequent types (in the original work, k = 5) and the most frequent token in the information extracted for all the labels are associated to the column, thus returning a set of *column types* and one *column token*. The algorithm then uses unambiguous labels, *i.e.*, labels for which one unique candidate is found, to understand which columns in the table describe *facts* about the entities that are also present in the KG. Given the entity column $C_j$ under evaluation, it tries to annotate a context column $C_k$ with a KG property $P$ that describes the relation between the labels in the entity column and the values in the context columns. When a context column $C_k$ is annotated with a property $P$ also used in the KG, a fact $P(label_{i,j}, value_{i,k})$ can be extracted from the $i$th row of the table and used to look up more candidates, and, in particular, all those entities $x$ for which the fact $P(x, value_{i,k})$ is part of the KG, thus expanding the set of candidates for a given label. To choose the properties to annotate some of the context columns, the algorithm picks each unambiguous label in the $C_j$ column and, for each context column $C_k$, matches the pair $(label_{i,j}, value_{i,k})$ against all the indexed facts. A property $P$ is chosen to annotate a context column if it matches facts extracted from at least $n$ different rows (the original work heuristically set $n = 5$) that have unambiguous labels. If more properties satisfy this constraint, the most frequent property is selected. We refer to this property as to the *context column property.* As a consequence of this step, the entity column under attention is annotated with a set of its (five) column types and its most frequent token, while some context columns are annotated with a property.
- *Annotation by lookup - for unambiguous labels.* Unambiguous labels found in the previous step are annotated with their unique candidate entity.
- *Annotation by strict lookup for ambiguous labels*: this step refines the candidates list of ambiguous labels for which more than one candidate was found, by filtering out candidates that have types that differ from the entity column types *and* a description that does not contain the most frequent token. The label is annotated with the candidate with the highest score.
- *Loose lookup for labels without candidates*: this step looks for new candidates for the labels which annotation is still missing; given the context columns annotated in the first step with a property, this step retrieves as

candidates all the entities that match some of the indexed facts of the KG, as explained in the preliminary step. The new set of candidates is then ranked based on the edit distance between the entity name and the label in the entity column. The first candidate is used to annotate the label in current cell.

*EmbeddingsOnGraph* The algorithm can work column-wise, row-wise or table-wise. It does not apply any filter by type mechanism and is based on the construction of a disambiguation graph like several approaches also applied to named entity linking [15]. We describe the column-wise approach that was tested in the original algorithm,[4] and we use in our experiments. A set of candidates is retrieved for each label in the entity column by selecting the best $m$ matches using a char-level trigram similarity with a threshold $\sigma$ (where $m = 8$ and $\sigma = 0.82$ in the original paper). All the candidate entities for each label represent the nodes of the disambiguation graph; each node has i) a *prior* probability, which is based on the degree (in links + out links) of the corresponding DBpedia page (see the original paper for details), and ii) an embedding that represent the entity. Each pair of candidates from different sets is connected by an edge, which is weighted by the cosine similarity of the respective embeddings. Finally, the priors are updated by executing PageRank on the constructed graph; the candidate with the highest score in each set is chosen for the annotation.

*HybridI/HybridII* The hybrid models sequentially combine the baselines; HybridI executes FactBase first, then annotates cells without an annotation with using EmbeddingsOnGraph; HybridII works in the other way around.

## 4. NEST: Candidate Selection with Soft Type Constraints

NEST (NEural Soft Type Constraints) is a methodology to replace hard constraints based on entity types adopted in CEA pipelines. The methodology relies on distributed representations of entities, *i.e.*, entity embeddings, which can be computed with different approaches [16]. We consider two strategies to include soft type constraints, addressing two patterns used in previous work and, especially, in the unsupervised state-of-the-art algorithms described in Section 3:

- the first strategy, *type enrichment for filtering by type*, combines direct types with types predicted by a neural model to refine type-based filtering. The neural model relies on distributed representations of entities for type prediction. Given an input vector representing an entity, a neural network returns a probability distribution over the possible entity types. The distribution can be used to select a list of most probable types according to

[4]The approach used in the original work was not explicitly stated. The table-wise approach combines richer information but at the price of scalability for large tables with thousands of rows; the row-wise approach demands for embedding that maximizes the relatedness between entities, like the ones used in the original work, while the column-wise approach should exploit embeddings which maximize the type coherence between similar entities. These aspects are not discussed in [6], but we suppose the original algorithm works column-wise because it has been tested only on datasets featuring tables with at most one entity column.

the network, which can enrich (or refine) the set of direct types, *e.g.*, by predicting the type `Philosopher` for `Albert Einstein`. This strategy will be demonstrated by applying it to the FactBase algorithm.

- the second strategy, *type enrichment for ranking by distributed entity representations similarity*, starts from the consideration that entity embeddings are particularly useful to evaluate entity similarity, and evaluating the similarity between candidate entities in a same column helps entity disambiguation like in the EmbeddingsOnGraph algorithm. However, the type-level characterization is not explicitly featured in popular entity embeddings, *e.g.*, RDF2Vec. To feature type-level information more explicitly in the embeddings, we rely on type embeddings [17], *i.e.*, distributed representations of entity types: given the vector of an entity **e** and the vector of its type **t**, the two vectors are concatenated generating a final representation, a *typed entity embedding*, in a vector space where entities that share the same types are closer [17]. In this way, the type-parts of the concatenated vectors induce a sort of *soft* (type) constraint over the selected annotations. For example, if the vector `Philosopher` is concatenated to the vector of `Albert Einstein`, this candidate entity will be more similar to other entities of type `Philosopher`; otherwise, since `Philosopher` and `Scientist` are similar in the type space, even if we concatenate the vector `Scientist`, the similarity between `Albert Einstein` and other philosophers will not be penalized too much.

A clear advantage of NEST is its modularity. The two methods can be used jointly and also integrated into different pipelines with near to zero engineering disruption. Different entity and type embeddings and different type prediction models can be adopted for NEST. For type embeddings, we use Type2vec [17], a model inspired by distributional semantics that does not require expressive and rich ontologies. Type2Vec embeddings are obtained by annotating a text with entities, replacing the entities with their direct type and then applying Word2Vec [18] to generate the embeddings. In Section 4.1 we describe two neural prediction models working with different source of information, while in Section 4.2 we explain in detail how we featured these strategies into algorithms by [6] to demonstrate their effectiveness.

## *4.1. Type Prediction Models*

The two Deep Neural Networks (DNNs) introduced for type prediction are depicted in Figure 2 and take as input embeddings generated with different models. The first one uses embeddings generated with RDF2Vec [19], which creates a virtual document containing random walks over a KG and then applies Word2Vec on this document. The second one uses entity embeddings generated from textual descriptions using Bidirectional Encoder Representations from Transformers (BERT) [20], which has shown strong performance over several downstream tasks in different languages [21]. Thus, for generating BERT Entity Embeddings (BERT EE), we collect DBpedia abstracts, and for each entity, we extract token

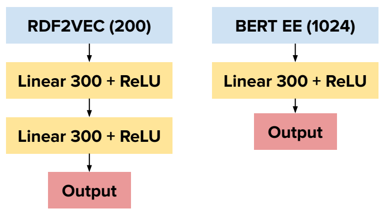


**Figure 2.** The two architectures used in this paper, numbers describe layer size.

embeddings from its abstract with BERT,[5] and then we represent the entity with the average vector of the token embeddings of its abstract.

The architectures have been selected with respect to the performances obtained at validation time and we use early stopping to prevent overfitting. Both DNNs are trained to reduce the categorical cross-entropy loss. To predict the type of an entity, NEST implements a classifier modeled as a straightforward DNN, which learns to map entities with similar embeddings to the same type. As an example, we obtain that the entity `Albert Einstein`, whose embedding is similar to the embeddings of other scientists and philosophers, has an high probability of being of type `Scientist` and of type `Philosopher`. While architecturally simple, the DNNs in NEST can be trained quickly with already available data and it does not require sophisticated hardware to be trained. This increases the applicability of the methods to different KGs.

### *4.2. NEST-enriched algorithms*

*FactBaseST* Since FactBase uses the column types and the column token to filter out the retrieved candidates, we enhance this algorithm by exploiting the neural type prediction models, trained and executed over DBpedia (see details below), instead of relying only on the direct types from the KG. The usage of predicted types in FactBaseST aims to capture two intertwined intuitions: (i) Predicted types for all entities in a column can provide additional evidence to determine the correct column types thus reducing the set of column types used as filters subsequently; (ii) By enriching the set of types associated with a candidate entity (*e.g.*, by adding the type `Philosopher` to `Albert Einstein`), the filters applied to individual candidates are softened and become less sensitive to missing type information or mismatches between the intended conceptualization in the table column and the classification in the KG. More precisely, these intuitions are captured as follows:

- The set of column types extracted from the KG is refined by retaining only types predicted also by the neural type prediction model (if any, otherwise, direct types are preserved). The set of predicted types used in this refinement consists of the $h$ most frequent types among all the most likely types predicted for each candidate. We chose $h = 5$ as for the column types.

[5]BERT has a limit on the number of input tokens, if the sentence we pass to it contains more tokens, the rest is ignored.

- The set of types associated with each individual candidate is extended by considering also the two most likely types predicted for the candidate. We consider only the first two predicted types because we found that this reflects the average number of different direct types in DBpedia, for the entities that have more than one direct type.

By considering the two neural type prediction models described in Section 4, we implemented two versions of FactBaseST:

- FactBaseST-R2V uses the DNN based on RDF2vec to predict types. The network has been trained using 200-dimensional vectors of ~200k DBpedia entities as input [19] (1000 samples per type); we removed those types with less than 1000 instances to reduce variability (*e.g.*, `BowlingLeague` and `MouseGene`), resulting in 236 different predictable types. This network does not exploit the textual description of entities, thus it can be applied to almost every KGs. This DNN has shown remarkable performance, surpassing 0.90 in accuracy score both on the training and on the validation set (that is run 20% of the data, we use early stopping on the validation loss with a patience[6] equal to 4) on the type prediction task.
- FactBaseST-A2V, uses the DNN based on BERT EE to predict types, which is suitable for a KG that includes a textual description of its entities. We trained the DNN in FactBaseST-A2V with embeddings generated by feeding a pre-trained large BERT model (1024-dimensional vectors) with abstracts of DBpedia entities. Like above, we sampled 1000 samples per type removing those types with less than 1000 instances, but we also had to remove from the training set the entities without an abstract, resulting in 228 different predictable types. This DNN scores slightly above 0.90 in accuracy score on the type prediction task (same conditions as above).

*EmbeddingsOnGraphST* This algorithm is an extension of EmbeddingsOnGraph and differs only in the - small but conceptually relevant - difference that *typed entity embeddings* are used to computed the similarity instead of plain entity embeddings. We remind that this mechanism makes entities with similar predicted types have a higher cosine similarity; as a consequence, when the PageRank is computed, the weight of edges between entities of the same type increases while the weight of edges between entities of different types decreases further, thus implementing the soft constraint over the similarity that we aimed to capture.

*HybridIST/HybridIIST* The algorithms jointly apply FactBaseST and EmbeddingsOnGraphST (as for their original implementations), generating the variants HybridIST-R2V, HybridIST-A2V, HybridIIST-R2V, HybridIIST-A2V.

## 5. Experiments and Results

Our experiments can be replicated using the documented code we release.[7]

[6]Maximum number of epochs with no improvement before stopping the training.
[7]Source code available at https://github.com/vcutrona/nest.

**Table 1.** Profiles of benchmark datasets.

| Dataset | Cols (avg) | Rows (avg) | Matches | Entities | Cols with matches (avg) | Tables |
|---|---|---|---|---|---|---|
| T2D | 1,153 (4.95) | 28,333 (121.60) | 26,124 | 13,785 | 233 (1.00) | 233 |
| ST19-R4 | 3,564 (4.36) | 51,249 (62.73) | 107,352 | 53,007 | 1,732 (2.12) | 817 |
| 2T | 802 (4.46) | 194,438 (1080.21) | 663,656 | 15,997 | 540 (3.00) | 180 |

We recreated the SemTab 2019 environment for the CEA task, thus we used DBpedia 2016-10 as the target KG and we scored the algorithms using the *macro* Precision (P), Recall (R), and F1-score (F1) metrics [1].

*Datasets* In our experiments we considered three datasets (Table 1):

- T2D [7] is a small dataset with only limited contents, but still represent a reference dataset in the literature; we also used it to better observe the impact of the adaptation we made while re-implementing the original algorithms. We did not use the other datasets tested with the original algorithms (Limaye [13] and Wikipedia [6]) because Limaye has a profile similar to the T2D one, but with smaller tables and less columns to annotate; Wikipedia features only very small tables (23 cells in average to annotate for each column), and anyway it has been partially included in the SemTab dataset.
- ST19-R4 is part of SemTab 2019 [1]. This dataset is the only one in SemTab that contains only non-trivial cases.[8] More importantly, ST19-R4 has been built using a generator, which constructs tables by querying DBpedia. Each table has one class as the main topic, and the other columns are filled with values of a predefined pool of properties of each instance. The generator ensures that the type of the object property matches the expected range in the ontology [1]. Thus, the problem we are addressing in this paper has been artificially removed from ST19-R4. As an example, in a table about `Film`, the actor `Arnold Schwarzenegger`, typed as `OfficeHolder` in DBpedia, will be filtered out from the results of the property `starring`, which has range `Actor`. For this reason, NEST is not expected to improve the results on ST19-R4, making it a good resource to study the possible negative impact of applying NEST to algorithms.
- Tough Tables (2T) [3] features ambiguous and noisy tables that resemble real-world cases. 2T has been included in the last round of SemTab 2020, showing that its high ambiguity makes it harder than any other dataset.[9]

*Algorithms* To demonstrate the generality of NEST, we looked for algorithms in literature to use in our experiments. However, just a few CEA algorithms are open-source, with some limitations (*e.g.*, MantisTable [10] has scalability issues;

[8]The performance obtained in SemTab 2019 for this dataset were high, also thanks to hard-coded workarounds adopted by the participants [22] that we did not implement.

[9]Performance with the Wikidata version of 2T is dramatically reduced for all the systems that participated in the SemTab 2020 challenge [2].

the repository of CSV2KG [22] is incomplete. Authors of the best performing systems in SemTab 2019 (MTab [5], Tabularisi [11], and CSV2KG [22]) reported that their systems were not ready to be released.

Thus, in our experiments we applied NEST to the algorithms described in Section 4 because they employ the filtering and ranking strategies we want to test with NEST, and can scale reasonably (*i.e.*, annotating selected datasets takes a few hours). The source code of these algorithms is not available, but they are partially reproducible since they have been explained in detail [6]. We tested our re-implemented versions on T2D - used in the original work - observing low performance compared with the original results.[10] In our opinion, the performance decrease is due to the following factors:

- In the original work, the private *FactBase* index was used; the index included documents from Wikipedia, WikiData and DBpedia (not provided). Thus, we generated a new ElasticSearch index containing entities from DBpedia, their labels,[11] and their anchor texts from Wikipedia. We preferred to not include Wikidata in our index to reduce the amount of total memory needed to store it.
- Within FactBase, the *description* field of WikiData entities is crucial for the candidate disambiguation phase. The corresponding property of this field in DBpedia is `dct:description`, which however is missing for many entities (*e.g.*, `Milan`). We resembled it by analyzing the DBpedia short abstracts, which leads to different descriptions.
- The queries to the index are not publicly available, thus it is not possible to either reuse the same strategies (*e.g.*, fuzzy match) or apply the same parameters (*e.g.*, max edit distance in fuzzy search).
- We used the RDF2Vec vectors (uniform model from [19]) in EmbeddingsOnGraph, which differ from the embeddings used in the original work.

Given the above limitations, we managed to replicate the disambiguation pipelines of the original algorithms, while the lookup search is suboptimal.

The original algorithms annotate rows, based on the one entity per cell assumption. We provided a generalization of the algorithms by exploiting their column-wise nature: we can annotate tables with multiple entity columns by considering one entity column at a time, and setting the other as context columns, thus annotating individual cells instead of entire rows.

For our experiments, we modified the original EmbeddingsOnGraph algorithm to avoid scalability issues; in fact, running EmbeddingsOnGraph on a table with 5000 rows will lead to the creation of a disambiguation graphs with 40k nodes in the worst case (if all the top-8 candidates for each label are distinct) and ~800M edges. The disambiguation graph is a k-partite graph; thus the maximum number of edges is $\frac{n^2(k-1)}{2k}$ in the worst case. We thus split big tables in chunks of 500 rows each to execute the algorithm on large tables.

[10] A proper comparison is not possible because we computed the macro P, R, and F1 (as in SemTab), while the original work reported their micro versions. We report the original scores to help the reader in quantifying the gap: FactBase (P: 0.88, R: 0.78, F1: 0.83); EmbeddingsOnGraph (P: 0.86, R: 0.77, F1: 0.81).

[11] We also include the labels from the DBpedia Lexicalization datasets.

**Table 2.** Results for benchmark datasets. ♣ identifies our algorithms. Highlighted, best result for each dataset; bold text, best results for each algorithm.

| Method | T2D | | | ST19-R4 | | | 2T | | |
|---|---|---|---|---|---|---|---|---|---|
| | **P** | **R** | **F1** | **P** | **R** | **F1** | **P** | **R** | **F1** |
| EmbeddingsOnGraph | 0.782 | 0.723 | 0.751 | 0.483 | 0.470 | 0.477 | 0.293 | 0.245 | 0.267 |
| EmbeddingsOnGraphST ♣ | **0.811** | **0.751** | **0.780** | **0.540** | **0.526** | **0.533** | **0.378** | **0.316** | **0.344** |
| FactBase | **0.791** | 0.635 | 0.704 | **0.745** | **0.465** | **0.573** | 0.365 | 0.185 | 0.246 |
| FactBaseST-R2V ♣ | 0.789 | **0.638** | **0.706** | 0.731 | 0.454 | 0.560 | **0.434** | **0.241** | **0.309** |
| FactBaseST-A2V ♣ | 0.783 | **0.638** | 0.703 | 0.735 | 0.458 | 0.565 | 0.374 | 0.216 | 0.274 |
| HybridI | 0.756 | 0.740 | 0.748 | 0.530 | 0.526 | 0.528 | 0.275 | 0.231 | 0.251 |
| HybridIST-R2V ♣ | **0.766** | **0.751** | **0.759** | 0.549 | 0.544 | 0.546 | **0.355** | **0.299** | **0.324** |
| HybridIST-A2V ♣ | 0.762 | 0.746 | 0.754 | **0.551** | 0.547 | **0.549** | 0.317 | 0.266 | 0.289 |
| HybridII | 0.758 | 0.742 | 0.750 | 0.488 | 0.484 | 0.486 | 0.295 | 0.248 | 0.270 |
| HybridIIST-R2V ♣ | **0.784** | **0.768** | **0.776** | **0.544** | **0.540** | **0.542** | **0.380** | **0.319** | **0.347** |
| HybridIIST-A2V ♣ | **0.784** | **0.768** | **0.776** | **0.544** | **0.540** | **0.542** | **0.380** | **0.319** | **0.347** |

We remark here that the gap in the algorithms performance does not impact on our analysis, which is fair over the different models.

## 5.1. Results

Table 2 confirms that the use of NEST can improve state-of-the-art matching pipelines. As expected, applying NEST to EmbeddingsOnGraph increases the F1 score in all our tests (T2D: +2.9%; ST19-R4: +5.6%; 2T: +7.7%) because using typed entity embedding strengthens the similarity between entities of the same type; as a result, since EmbeddingsOnGraph is a column-wise approach, the typed entity embeddings can guarantee a higher column type coherence, which was completely disregarded in the original work. Also considering the improvement brought by the application of NEST, results on 2T are still poor. The main reasons are that i) the candidate retrieval phase is based only on the trigram similarity and uses a high threshold, returning a small set of candidates, and ii) the *priors* used to initialize PageRank are based on the entity popularity, which rewards popular entities in almost the cases, but 2T contains tables with many homonyms, which often do not link to the most popular mentioned entity.

FactBase does not benefit a lot from NEST when annotating T2D and ST19-R4, while its contribution is more valuable on 2T. We observe similar results for FactBaseST-R2V and FactBaseST-A2V, showing that the type information can be predicted alternatively from textual and factual descriptions, when both are available. However, we did not have enough evidence to prefer one source over the other for similar KGs. The performance on T2D are in line with the one achieved by the original algorithm (< 1%): the recall slightly increases, while the precision drops a bit; this is the expected behaviour, since types in T2D are homogeneous in each column; furthermore, tables in T2D are mainly collected from Wikipedia, the same source used to create DBpedia, so there is an overlap between the information in Wikipedia and the entity representations in the KG. If there is a mismatch, like the example of `Albert Einstein` in Section 3, NEST helps increasing the performance, but in all the other cases, considering the secondary type of an entity may add noise into the matching pipeline. This effect is amplified in ST19-R4, where we knew in advance that there was a perfect match between the content of the tables and the type information in DBpedia. Interestingly, the performance drop is limited ($-1\%$ in P and R), proving that NEST does not critically affect the original performance; moreover, the small loss on this specific dataset is balanced by the consistent gain in all the other settings where the artificial removal of this problem does not occur. We observe that the standard FactBase pipeline underperforms on 2T, mainly because the candidate retrieval step is not able to deal with the values in 2T tables, which are ambiguous and perturbed with typos. However, using NEST to relax the type-based filtering leads to a valuable performance increase (+6.3% and +2.8% for FactBaseST-R2V and FactBaseST-A2V respectively), helping the algorithm to disambiguate the higher number of candidates. Hybrid methods improve the recall of their main method at cost of precision. Results of HybridII-based algorithms are higher[12] thanks to the better performance of their main EmbeddingsOnGraphST method.

## 6. Conclusions and Future Work

In this paper we presented NEST, a novel methodology to include soft type-based constraints into an entity matching pipeline. NEST is modular and can be applied with nearly zero effort. Our experimental campaign shows how state-of-the-art algorithms can benefit from the use of NEST, testing different NEST-improved algorithms on benchmark datasets with different profiles. In our experiments, we used DBpedia as target KG, and we expect NEST achieving similar results on KGs with a similar type hierarchy granularity (which is a typical size in entity linking in tables). As future work, we plan to investigate the application of soft constraints to matching algorithms that target KGs with a more fine-grained type hierarchy (also by many order of magnitude, like YAGO); in this scenario, training a type prediction model could be challenging. Also, we are interested in finding more insights about possible error patterns, studying if the prediction models are biased towards specific types. Finally, we want to investigate the effects of replacing the prediction models with more complex models [4], and relaxing filters for entities linked on the same row, so that to increase their relatedness.

## References

[1] Jiménez-Ruiz E, Hassanzadeh O, Efthymiou V, Chen J, Srinivas K. SemTab 2019: Resources to Benchmark Tabular Data to Knowledge Graph Matching Systems. In: ESWC. vol. 12123 of Lecture Notes in Computer Science; 2020. p. 514-30.

[2] Jiménez-Ruiz E, Hassanzadeh O, Efthymiou V, Chen J, Srinivas K, Cutrona V. Results of SemTab 2020. In: SemTab 2020@ISWC. vol. 2775 of CEUR Workshop Proceedings; 2020. p. 1-8.

[12] Results in Table 2 show equal performance, but it is an effect due to the rounding.

[3] Cutrona V, Bianchi F, Jiménez-Ruiz E, Palmonari M. Tough Tables: Carefully Evaluating Entity Linking for Tabular Data. In: ISWC. vol. 12507 of Lecture Notes in Computer Science; 2020. p. 328-43.

[4] Melo A, Völker J, Paulheim H. Type Prediction in Noisy RDF Knowledge Bases Using Hierarchical Multilabel Classification with Graph and Latent Features. Int J Artif Intell Tools. 2017;26(2).

[5] Nguyen P, Kertkeidkachorn N, Ichise R, Takeda H. MTab: Matching Tabular Data to Knowledge Graph using Probability Models. In: SemTab@ISWC. vol. 2553 of CEUR Workshop Proceedings; 2019. p. 7-14.

[6] Efthymiou V, Hassanzadeh O, Rodriguez-Muro M, Christophides V. Matching Web Tables with Knowledge Base Entities: From Entity Lookups to Entity Embeddings. In: ISWC. vol. 10587 of Lecture Notes in Computer Science; 2017. p. 260-77.

[7] Ritze D, Lehmberg O, Bizer C. Matching HTML Tables to DBpedia. In: WIMS; 2015. p. 10:1-10:6.

[8] Zhang Z. Effective and efficient Semantic Table Interpretation using TableMiner$^+$. Semantic Web. 2017;8(6).

[9] Chabot Y, Labbé T, Liu J, Troncy R. DAGOBAH: An End-to-End Context-Free Tabular Data Semantic Annotation System. In: SemTab@ISWC. vol. 2553 of CEUR Workshop Proceedings; 2019. p. 41-8.

[10] Cremaschi M, Paoli FD, Rula A, Spahiu B. A fully automated approach to a complete Semantic Table Interpretation. Future Gener Comput Syst. 2020;112.

[11] Thawani A, Hu M, Hu E, Zafar H, Divvala NT, Singh A, et al. Entity Linking to Knowledge Graphs to Infer Column Types and Properties. In: SemTab@ISWC. vol. 2553 of CEUR Workshop Proceedings; 2019. p. 25-32.

[12] Chen J, Jiménez-Ruiz E, Horrocks I, Sutton C. Colnet: Embedding the semantics of web tables for column type prediction. In: AAAI; 2019. p. 29-36.

[13] Bhagavatula CS, Noraset T, Downey D. TabEL: Entity Linking in Web Tables. In: ISWC. vol. 9366 of Lecture Notes in Computer Science; 2015. p. 425-41.

[14] Deng X, Sun H, Lees A, Wu Y, Yu C. TURL: Table Understanding through Representation Learning. Proc VLDB Endow. 2020;14(3):307-19.

[15] Moro A, Raganato A, Navigli R. Entity Linking meets Word Sense Disambiguation: a Unified Approach. TACL. 2014;2.

[16] Ji S, Pan S, Cambria E, Marttinen P, Yu PS. A survey on knowledge graphs: Representation, acquisition and applications. arXiv:200200388 preprint. 2020.

[17] Bianchi F, Palmonari M, Nozza D. Towards encoding time in text-based entity embeddings. In: ISWC. vol. 11136 of Lecture Notes in Computer Science; 2018. p. 56-71.

[18] Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J. Distributed representations of words and phrases and their compositionality. NIPS. 2013;26.

[19] Ristoski P, Rosati J, Di Noia T, De Leone R, Paulheim H. RDF2Vec: RDF Graph Embeddings and Their Applications. Semantic Web Journal. 2018.

[20] Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: NAACL-HLT; 2019. p. 4171-86.

[21] Nozza D, Bianchi F, Hovy D. What the [mask]? making sense of language-specific BERT models. arXiv:200302912 preprint. 2020.

[22] Vandewiele G, Steenwinckel B, Turck FD, Ongenae F. CVS2KG: Transforming Tabular Data into Semantic Knowledge. In: SemTab@ISWC. vol. 2553 of CEUR Workshop Proceedings; 2019. p. 33-40.