Extending Cross-Domain Knowledge Bases with Long Tail Entities using Web Table Data

Yaser Oulabi, Christian Bizer
Data and Web Science Group, University Mannheim
B6 26, 68159 Mannheim, Germany
{yaser,chris}@informatik.uni-mannheim.de

ABSTRACT

Cross-domain knowledge bases such as YAGO, DBpedia, or the Google Knowledge Graph are being used as background knowledge within an increasing range of applications including web search, data integration, natural language understanding, and question answering. The usefulness of a knowledge base for these applications depends on its completeness. Relational HTML tables from the Web cover a wide range of topics and describe very specific long tail entities, such as small villages, less-known football players, or obscure songs.

This systems and applications paper explores the potential of web table data for the task of completing cross-domain knowledge bases with descriptions of formerly unknown entities. We present the first system that handles all steps that are necessary for this task: schema matching, row clustering, entity creation, and new detection. The evaluation of the system using a manually labeled gold standard shows that it can construct formerly unknown instances and their descriptions from table data with an average F1 score of 0.80. In a second experiment, we apply the system to a large corpus of web tables extracted from the Common Crawl. This experiment allows us to get an overall impression of the potential of web tables for augmenting knowledge bases with long tail entities. The experiment shows that we can augment the DBpedia knowledge base with descriptions of 14 thousand new football players as well as 187 thousand new songs. The accuracy of the facts describing these instances is 0.90.

1 INTRODUCTION

Cross-domain knowledge bases like YAGO [18], DBpedia [20], Wikidata [30], or the Google Knowledge Graph are being employed for an increasing range of applications, including natural language processing, web search, and question answering.

The YAGO, DBpedia, and Wikidata knowledge bases all rely on data that has been extracted from Wikipedia and as a result cover mostly head instances that fulfill the Wikipedia notability criteria. Their coverage of less well known instances from the long tail is rather low [11]. As the usefulness of a knowledge base often increases with its completeness, adding long tail instances to an existing knowledge base is an important task.

Web tables [8], which are relational HTML tables extracted from the Web, contain large amounts of structured information, covering a wide range of topics, and describe very specific long tail instances. Web tables are thus a promising source of information for the task of augmenting cross-domain knowledge bases.

Augmenting knowledge bases with descriptions of long tail instances requires, on the one hand, identifying instances of a specific class that are not yet part of a knowledge base, and, on the other hand, compiling descriptions of the new instances according to the schema of the knowledge base. Two areas of related work are relevant for this task: Existing work on slot filling [11, 22, 23, 27, 29] focuses on adding missing facts describing existing instances to a knowledge base. The methods do not attempt to discover new instances. In contrast, existing research on set expansion [24, 31, 32] focuses on determining the names of new instances and is not concerned with compiling structured descriptions of those instances according to a schema of a knowledge base. As most set expansion methods disambiguate new instances solely based on names, they miss the potential of exploiting additional features for disambiguation.

As a result, no viable methods that are able to automatically augment a knowledge base with new instances and their descriptions exist. In this work, we close this gap by introducing and evaluating the first system that is able to generate descriptions of formerly unknown long-tail entities given a corpus of relational web tables. The system exploits the synergies between the task of identifying new instances and compiling descriptions of these instances using an iterative approach. The contributions of the paper are as follows:

- We introduce the first system that is able to generate descriptions of new instances given the set of all instances of a class from a knowledge base and a corpus of relational web tables.
- We evaluate our system using a manually built gold standard of annotated web tables and report the lessons learned from this experiment.
- We run our system over a large corpus of web tables which allows us to profile the general potential of web table data for augmenting knowledge bases with descriptions of long tail instances.

Figure 1 gives an overview of the overall process performed by our system. The process consists of four main steps which are executed in two iterations. We first apply schema matching methods to match web tables and attribute columns of those tables to classes and properties in the knowledge base. Second, a row clustering method identifies rows that describe the same entity. From these row clusters, the entity creation component creates entity descriptions according to the schema of the knowledge base. Finally, the new detection component determines whether an entity already exists in the knowledge base. We iterate over the pipeline a second time using the row clusters and entity-to-instance correspondences from the first run in order to refine the schema mapping. After the second run of the pipeline, entities identified as new are added to the knowledge base.

The paper is structured as follows: First, we describe the profile of the knowledge base, the web table corpus, and the gold standard that are used for the experiments throughout the paper. Section 3 describes and evaluates the individual steps of the overall process. Section 4 discusses the overall performance of the pipeline on the gold standard, while in Section 5 we run our

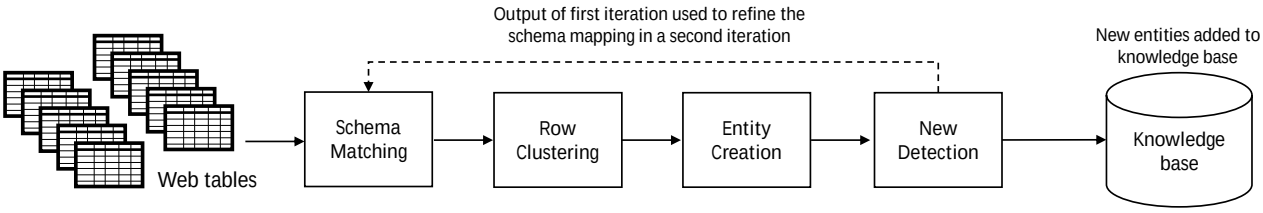

Figure 1: Overview of the overall pipeline.

Table 1: Number of instances and facts for selected DBpedia classes.

Class	Instances	Facts
GF-Player	20,751	137,319
Song	52,533	315,414
Settlement	468,986	1,444,316

system on the large corpus of web tables in order to profile the overall potential of web tables for the task at hand. Section 6 compares our system to the related work.

2 EXPERIMENTAL SETUP

This section describes the datasets that we use to evaluate our system. We will first describe the knowledge base and the selection of classes we aim to extend. We then describe the web table corpus in which we hope to find new instances. We finally describe the gold standard that we use throughout this work.

2.1 Knowledge Base and Classes

We employ DBpedia [20] as the target knowledge base to be extended. It is extracted from Wikipedia and especially Wikipedia infoboxes. As a result, the covered instances are limited to those identified as notable by the Wikipedia community.

From DBpedia we selected three classes on which we focus throughout this work. To ensure that information covered by the classes is diverse, we selected each from a different first-level class, i.e. Agent, Work, and Place. To ensure that the classes are not too broad, we preferred classes further down in the hierarchy. As a result we chose the following three classes: GridironFootballPlayer (GF-Player), Song and Settlement. The class Song includes all instances of the class Single. Tables 1 and 2 provide an overview of the number of instances and facts, and the property densities of those classes. We only consider properties that have an initial density of at least 30 %. We use the 2014 release of DBpedia, as this release has been used in related work [22, 23, 26, 27], and its release date is also closer to the extraction of the web table corpus used in this work.

Table 1 shows that DBpedia already covers tens of thousands of instances for the profiled classes. This could indicate that most of the well-known instances are already covered, so that we are especially interested in finding instances from the long tail.

Table 2 also reveals that the density differs significantly from property to property. Only the properties of class Song have consistently high densities larger than 60 %. The football player class has many properties, but half of them have a density below 50 %. The class Settlement suffers from both, a small number of properties, and low densities for some of them.

Table 2: Number of facts and property densities for selected DBpedia properties.

Class	Property	Facts	Density
GF-Player	**birthDate**	20,218	97.43 %
GF-Player	**college**	19,281	92.92 %
GF-Player	**birthPlace**	17,912	86.32 %
GF-Player	**team**	13,349	64.33 %
GF-Player	**number**	11,430	55.08 %
GF-Player	**position**	11,240	54.17 %
GF-Player	**height**	10,059	48.47 %
GF-Player	**weight**	10,027	48.32 %
GF-Player	**draftYear**	7,947	38.30 %
GF-Player	**draftRound**	7,932	38.22 %
GF-Player	**draftPick**	7,924	38.19 %
Song	**genre**	47,040	89.54 %
Song	**musicalArtist**	45,097	85.85 %
Song	**recordLabel**	43,053	81.95 %
Song	**runtime**	42,035	80.02 %
Song	**album**	40,666	77.41 %
Song	**writer**	33,942	64.61 %
Song	**releaseDate**	31,696	60.34 %
Settlement	**country**	433,838	92.51 %
Settlement	**isPartOf**	416,454	88.80 %
Settlement	**populationTotal**	292,831	62.44 %
Settlement	**postalCode**	154,575	32.96 %
Settlement	**elevation**	146,618	31.26 %

2.2 Web Table Corpus

In this work, we utilize the english-language relational tables set of the Web Data Commons 2012 Web Table Corpus.[1] The set consists of 91.8 million tables. Table 3 gives an overview of the general characteristics of tables in the corpus. We can see that the majority of tables are rather short, with an average of 10.4 rows and a median of 2, whereas the average and median number of columns are 3.5 and 3. As a result, a table on average describes 10 instances with 30 values, which likely is a sufficient size and potentially useful for finding new instances and their descriptions. In [27] we have profiled the potential of the same corpus for the task of slot filling, meaning to find missing values for existing DBpedia instances.

For every table we assume that there is one attribute that contains the labels of the instances described by the rows. The remaining columns contain values, which potentially can be used to generate descriptions according to the knowledge base schema.

[1] http://webdatacommons.org/webtables/#toc3

Table 3: Characteristics of the web table corpus.

	Average	Median	Min	Max
Rows	10.37	2	1	35,640
Columns	3.48	3	2	713

Table 4: Number of tables and value correspondences for selected classes.

Class	Tables	V_{Matched}	$V_{\text{Unmatched}}$
GF-Player	10,432	206,847	35,968
Song	58,594	1,315,381	443,194
Settlement	11,757	82,816	13,735

For the three evaluated classes, Table 4 shows the result of matching the table corpus to existing instances and properties in DBpedia, using a method from previous work [26, 27]. The first column shows the number of matched tables that have at least one matched attribute column. Rows of those tables were matched directly to existing instances of DBpedia. From the second and third columns we see how many values were matched to existing instances and how many values remained unmatched. While more values were matched, the number of unmatched values is still large, especially for the song class.

2.3 Gold Standard

For the purpose of this work we built a publicly available gold standard of annotated web tables. We first annotated clusters of rows that describe the same instance. Additionally, we annotated if these clusters describe new instances and for clusters that overlap with existing instances in DBpedia, we also annotated the clusters with the correspondence to the existing instance. We annotated attribute-to-property correspondences, where table columns are mapped to properties in DBpedia. Finally, we annotate facts for all cluster and property combinations for which a candidate value exist in the annotated web tables.

The gold standard contains tables with instances of varying degree of popularity, so that they describe head and long tail instances. We also prioritized tables with rows that are unlikely to have a match in DBpedia and ensured that for some labels, we select at least five rows to be able to form large enough clusters.

Table 5 provides an overview of the annotations per class. In the first three columns we see the number of table, attribute and row annotations. On average, we have 1.85 attribute annotations per table, not counting the label attribute. The two following columns show the number of annotated clusters, followed by the number of values within those clusters that match a knowledge base property. The second to last column shows the number of overall value groups, i.e. the number of cluster and property combinations for which at least one candidate value exists. For all groups we included facts, i.e. the correct value given the group's cluster and property. The last column shows for how many of the groups, the correct value is contained among the candidate values. We annotated 271 clusters, of which 39 % are new. On average, each cluster has approximately 3.42 rows, 7.69 values, 3.17 value groups and their facts, and 2.88 groups where the correct value is present in the web tables.

We use the gold standard for learning and testing. For this, we split the data into three folds and performed cross-validation. We ensured that we evenly split new clusters and homonym groups, which are groups of clusters with highly similar labels. All clusters of a homonym group were always placed in one fold.

The gold standard, along with the code and other data, is publicly available.[2] The results of this work are therefore replicable.

3 METHODOLOGY

In this section we present our system and evaluate alternative approaches for the individual components of the pipeline. As shown in Figure 1, the pipeline begins with the web tables and ends with entities being added to the knowledge base as new instances. In between, the pipeline consists of four components: schema matching, row clustering, entity creation and new detection.

We iterate over the pipeline twice. During the second run, we utilize the output of the row clustering and the new detection to generate a refined schema mapping. The attribute-to-property correspondences derived by the schema matching are important, because they allow us to extract for a row a set of values which correspond to the schema of the knowledge base. These values are utilized by the row clustering and new detection components with a positive impact on performance. More importantly, these values are required to create descriptions for new instances.

During the schema matching phase, we also match each table to a class in the knowledge base. Afterwards we run the remainder of the pipeline for each class separately.

3.1 Schema Matching

The first step in the pipeline is to create a mapping between the schemata of the individual web tables and the schema of the knowledge base. As the web tables have heterogeneous schemata, this task is non-trivial. Overall there are four steps necessary: (1) data type detection, (2) label attribute detection, (3) table-to-class matching and (4) attribute-to-property matching.

Data Type Detection

Throughout our pipeline we utilize a number of data types to type individual values, facts, attribute columns or knowledge base properties. Each type has a corresponding similarity function, and an equivalence threshold, which is used to determine if the compared values are equal. We employ overall six data types:

- **Text:** string, where two strings do not have to be exactly equal to be similar, e.g. label of an instance.
- **Nominal String:** string, where two strings are either completely equal or unequal, e.g. ISO code of a country.
- **Instance Reference:** reference to an instance, e.g. team of an athlete or musical artist of a song.
- **Date:** date with two possible granularities: year or specific day, e.g. release date of song, or birth date of a person.
- **Quantity:** numeric quantity, where numeric closeness has a semantic relevance, e.g. population of a settlement.
- **Nominal Integer:** integer, where numbers close to each other are not semantically related. This include e.g. numbers or draft rounds of athletes. Typing nominal integers in addition to nominal strings allows some components, especially the attribute-to-property matcher, to use methods tailored for this type.

We run a data-type detection algorithm [26] that assigns to each table attribute one of the following types: text, date and quantity. Detecting the other three types requires an understanding of the

[2] http://data.dws.informatik.uni-mannheim.de/expansion/LTEE/

Table 5: Overview of the gold standard.

Class	Tables	Attributes	Rows	Existing Clusters	New Clusters	Matched Values	Value Groups	Correct Value Present
GF-Player	192	572	358	81	19	1,207	475	444
Song	152	248	193	34	63	425	231	212
Settlement	188	162	376	49	25	451	152	124

actual semantics of an attribute, so that they are assigned by the attribute-to-property matcher, after an attribute has successfully been matched to a knowledge base property.

The data type detection is performed using manually defined regular expressions. We decide the data type of an attribute based on the majority data type among its values.

Label Attribute Detection

For each table we assign one column as the label attribute, which contains natural language labels for the entities described in the table rows [26]. For this we find the column with the data type text and the highest number of unique values. In case there is a tie between multiple columns, we choose the column that is furthest to the left [26].

Table-to-Class Matching

We utilize an approach that performs both row-to-instance and attribute-to-property matching to find the class of a table.

We first extract from the label attribute a label for each row, and use the label to find candidate instances from the knowledge base. A class, for which many rows of a table have a candidate instance, is chosen as a possible candidate class of that table. We assign the number of rows with a match as a score to that class.

Given these candidate classes, we then evaluate how well their properties match. We compare the values in the rows with facts of their candidate instance in the knowledge base, to find if they match a certain property of the candidate class. We block comparisons based on data type as detected above. Using the matched values we are able to perform duplicate-based attribute-to-property matching [5], where we chose the property with the highest number of matched cells as the property of the attribute, and assign the number as the score of the correspondence.

Per candidate class, we aggregate all scores to compute a ranked list of candidate classes. We choose the class with the highest score as the class of the table. This approach was proposed and evaluated by Ritze et al., where authors find that it can achieve an F1 score of 0.97 on a web table corpus [26].

Attribute-to-Property Matching

Our attribute-to-property matching approach consists of three steps. We first select candidate properties from the knowledge base schema based on data types. For text attributes, we choose all properties with types instance reference, nominal string and text, for quantity attributes we choose properties with types quantity and nominal integer and finally for date attributes, we choose properties with types date, quantity and nominal integer as candidates. After matching, the data type of the attribute is changed to the data type of the matched property and the values are accordingly normalized.

Secondly, we use various matchers, described further below, to compute matching scores. Given a candidate knowledge base property, a matcher finds a score from 0.0 to 1.0 that measures the likelihood that the attribute matches the property. Scores of multiple matchers are then aggregated based on a weighted average, where weights are learned for each class individually.

Table 6: Attribute-to-property matching performance by iteration.

Iteration	P	R	F1
First	0.929	0.608	0.735
Second	0.924	0.916	0.920
Third	0.929	0.916	0.922

We then utilize thresholds on the aggregated scores to determine if a certain candidate property matches an attribute. The thresholds are learned per property of the knowledge base schema. An attribute is matched to a property if it is both, a property that achieves a score above the property-specific threshold, and the property with the highest aggregated score.

Overall we implement five matchers, three of which exploit the knowledge base. **`KB-Overlap`** computes the proportion of values in the attribute that generally fit the candidate property in the knowledge base. **`KB-Label`** compares the label in the attribute header row to the labels of the candidate property in the knowledge base. **`KB-Duplicate`** computes the proportion of values in the attribute that is equal to the fact of the candidate property in the knowledge base, based on the instance correspondences generated by the new detection component.

We further implement two matchers that exploit the large web table corpus. For this, we first match attributes using the above described matchers for a preliminary mapping. We then rerun the matching using two additional matchers that exploit the preliminary mapping and the web table corpus. **`WT-Label`** utilizes the column headers of columns matched in the preliminary run, to derive label-to-property scores, where the score represents the likelihood that an attribute with a certain header row label corresponds to a certain candidate property of the knowledge base. **`WT-Duplicate`** knows through a previous row clustering run which rows in the tables describe the same instances. Using the preliminary mappings, we can find values in the corpus that are matched to same instance and property. This matcher measures and returns the proportion of values in an attribute, for which an equal value matched to same instance exists.

Table 6 shows by iteration the performance of an attribute-to-property matching method that aggregates all matchers. The duplicate-based methods are not included in the first iteration, as they require output from the other pipeline components. We evaluated the methods on the attribute annotations in the gold standard. We split the annotations first into a learning and a testing set, where the learning set contains two third of annotations.

From the table we can see that a second iteration and the utilization of the output of the row clustering and the new detection components have a large positive effect on schema matching

performance. The table also shows that a third iteration has only a marginal positive effect, so that two iterations suffice.

To determine the usefulness of each individual matcher, we evaluate the weights assigned in the aggregated method of the second iteration. As we learn weights per class, the following weights are averages. The duplicate-based matchers have a combined weight of 0.43, where the `KB-Duplicate` matcher with a weight of 0.25 is more important. The label-based matchers achieve a higher combined weight of 0.46 where the `WT-Label` with a weight of 0.25 is very effective. Finally the `KB-Overlap` method is the least important method with a weight of 0.10. Additionally, the distribution of weights for the individual classes were similar to the here mentioned averages.

From the weights we can first of all see, that the attribute label is quite an effective method for schema matching. More importantly, we can conclude, that the most effective approach is one that combines various matchers and thereby exploits the highest number of individual signals for schema matching.

3.2 Row Clustering

After matching tables to classes and table attributes to properties of the knowledge base, we cluster rows that describe the same instance together. This step is especially important, as it reveals the overall number of unique instances described in the tables.

Our row clustering methods consist of a row similarity metric, which measures the likelihood that two rows describe the same instance, and a clustering algorithm, that utilizes the similarity metric to create clusters of rows.

In earlier work [25–27], we determined which rows describe the same instance by matching them to existing instances in the knowledge base. As a result, our earlier methods were unable to cluster rows of new instances, unlike in this work, where we perform clustering independently from existing instances.

Clustering Algorithm

In the context of this work, a required feature of the clustering algorithm is its ability to determine the number of clusters. In case of a perfect clustering, this number would correspond to the exact number of instances described by the rows of all tables.

Correlation clustering approaches [1, 2, 6, 10] fulfill this requirement. Clustering here is viewed as an optimization problem that aims to find the optimal partitioning of a set of vertices by maximizing a fitness function that aggregates similarities within a partition and dissimilarities between partitions.

Due to the large number of rows that need to be clustered (see Table 11), we need clustering methods that scale. As correlation clustering approaches try to find the globally optimum solution, they do not scale for the task at hand. We therefore utilize a greedy correlation clustering algorithm [15, 16], which solves the optimization problem locally for each decision made by the clusterer. The algorithm employs a row similarity function with a normalized output from -1.0 to 1.0 and sequentially tries to assign each row to the optimum cluster by summing the similarity scores of the current row with all individual rows in already created clusters in order to compute aggregated row-to-cluster scores. If there are scores larger than zero, the row is assigned to the cluster with the highest score. If no score is larger than zero, a new cluster is created, and the row is assigned to it. While every row assignment or cluster creation would maximize the fitness function locally, this does not ensure global optimization.

To achieve further scalability, we perform the row assignment in parallel instead of sequentially. While this is much faster, it can results in errors during clustering. We therefore run the Kernighan-Lin with joins (KLj) [19] clustering algorithm as a secondary step. The algorithm improves an existing preliminary clustering, in our case the output of the parallelized greedy clustering, by comparing cluster pairs and attempting to move individual rows between those clusters or merging the clusters fully. Similarly, each cluster is compared with an empty set to find whether splitting a cluster increases the fitness function locally. The operations are repeated until no further operation is able to increase the local fitness function.

As a result, we are able to quickly build a preliminary clustering with complete parallelization, while with a second step, we ensure that clustering quality is still high. Scalability is further ensured by the blocking approach described below.

Row Similarity Metrics

A row similarity metric compares two rows and returns a score that measures the likelihood that the two rows describe the same instance. Depending on the exact implementation, other input, e.g. from the knowledge base or the web table corpus, might be utilized. We implement six different similarity metrics:

- **`LABEL`:** We use the label attribute of a table to derive labels for all rows to then derive a similarity score by comparing labels using the Monge-Elkan similarity with Levenshtein as the inner similarity function.
- **`BOW`:** For each row we create a bag-of-words binary term vector that contains the terms that occur in all cells of a row. For this, cell values are cleaned, normalized and tokenized. To compare two rows, we compute the cosine similarity of their vectors.
- **`PHI`:** This approach allows us to compare two rows by comparing their tables. It derives a similarity between two tables using the PHI correlation of row labels, which we first compute using the following formula:

$$PHI(x,y) = \frac{n \times n_{xy} - n_x \times n_y}{\sqrt{n_x \times n_y \times (n-n_x) \times (n-n_y)}},$$

  where n = total number of unique labels,
  n_{ab} = occurrence of labels a and b in same table,
  n_a = occurrence of label a in a table.

  For each label we therefore have a vector that measures its correlation with all other labels in the corpus. We then create such a vector for each table, by averaging the vectors of the table's row labels. With this we attempt to derive a vector that captures semantic information about all rows described in the table. When comparing two rows, we return the cosine similarity of the vectors of their tables.
- **`ATTRIBUTE`:** Using the attribute-to-property correspondences we can derive for each row values matched to the knowledge base schema. This allows us to perform value normalization and apply data-type-specific similarity functions to compare row values. If the two rows being compared have overlapping value pairs, so that both values are matched to same property, we use the data type similarity function to determine if those values are equal, assigning them a score of either 1.0 or 0.0. As there are possibly more than one overlapping value pair, the similarity returned equals to the average similarity scores of all pairs. Additionally, a confidence score is attached that

equals the number of pairs compared. This confidence score is used by the aggregation methods described below.

- **IMPLICIT_ATT:** Many tables have rows that describe instances that are similar, e.g. cities in Germany or athletes drafted in 2010. This information is not stated explicitly in any of the row cells. Using the following approach, we attempt to derive for a table implicit property-value combinations that apply to all instances described by the table. We can then use these implicit property-value combinations to compare rows with each other.
  We first use the row labels to find candidate instances for all rows, and then for each row all property-value combinations that exist for at least one candidate in the knowledge base. For each property-value combination we then derive a score for the whole table, which equals the proportion of rows that have this combination. We keep only combinations with a score above a certain threshold. The remaining combinations are the implicit attributes of a table and their score is assigned as a confidence score.
  Given two rows we compare the implicit attributes of one row with overlapping implicit attributes and column attributes of the other row and vice versa. We return the average of the similarities of all compared pairs and the sum of the implicit attribute scores as the confidence.
- **SAME_TABLE:** This metric builds on the observation that rows in a single table usually describe different entities. The metric assigns two rows of the same table a similarity of 0.0, otherwise 1.0.

Similarity Score Aggregation

We implement two approaches to aggregate the row similarity scores. We first utilize a weighted average, where the weights assigned to each metric are learned. In this case, attached confidence scores are not considered. We also learn a threshold, where scores above the threshold indicate that the rows describe the same instance. This threshold is used to normalize the similarity metric to -1.0 and 1.0. To learn the weights, we model the data in the learning set as row-pairs that either match or not, i.e. with scores of either 1.0 or 0.0. When learning weights we utilize a genetic algorithm that attempts to maximize the matching performance on the learning set.

As a second, alternative aggregation approach, we use random forest regression tree [7], where as features we include both similarity and confidence scores. We again model the data as row-pairs, where non-matching row-pairs are assigned a score of -1.0, while matching pairs a score of 1.0. To learn the random forest regression tree we utilize the WEKA library. We learn the hyperparameters of the algorithm by using the out-of-bag error with different out-of-bag rates on the learning set.

As a third aggregation approach, we combine both aggregation methods using a weighted average, where the weights are also learned as described above. In all cases we upsample to balance the number of matching and non-matching row pairs.

Blocking

To ensure the scalability of the row clustering for large web table corpora, we implement a blocking algorithm. We block comparisons by first limiting the number of clusters a row is compared to during the parallel greedy clustering, and, secondly, limiting the cluster pairs that are compared with each other during the KLj clustering.

Table 7: Average clustering performance and metric importance scores for alternative row clustering methods.

Run	PCP	AR	F1	MI
LABEL	0.71	0.83	0.76	0.33
+ **BOW**	0.73	0.84	0.78	0.18
+ **PHI**	0.74	0.84	0.78	0.05
+ **ATTRIBUTE**	0.75	0.85	0.80	0.21
+ **IMPLICIT_ATT**	0.78	0.87	0.82	0.17
+ **SAME_TABLE**	0.79	0.87	0.83	0.07

We utilize the row labels for the blocking mechanism. We first normalize the labels of all rows and use them to build a Lucene index. Each label in the index forms a block, which includes all rows with that exact label. For each row we use the index to retrieve a number of labels similar to the row's label, and assign their blocks to the row.

During the parallelized greedy clustering, we compare a row only to clusters with which the row shares a block. The blocks of a cluster are the union of the blocks of all rows in that cluster. Similarly, during the KLj clustering, two clusters are only compared when they share a block.

Evaluation

To evaluate the performance of the row clustering, we employ the evaluation approach proposed by Hassanzadeh et al. [17]. We use the set of clusters annotated in the gold standard, denoted as G, and the set of clusters returned by our method, denoted as C, to first compute a one-to-one mapping between the clusters in G and the clusters in C. We map a cluster in C to a cluster in G, if it contains the highest fraction of rows that are from that cluster in G. In case two clusters in C have the same proportion, we take the cluster with the highest absolute number of overlapping rows. We denote the mapping as M.

Using this mapping we compute average recall, penalized clustering precision and their F1 score. Average recall is the average of the individual recalls of the clusters in G. The recall of a cluster in G is equal to the ratio of rows in the mapped cluster from C that are also in G to the number of total rows in G. If no cluster from C was mapped to a cluster in G, the recall of that cluster is zero.

To compute the clustering precision we compute the precision of all pairs of rows that are part of the same cluster in C. A pair is determined to be correct if both rows are part of the same cluster in G. Unlike the average recall, this does not measure how well we find cluster, but how well we place rows in the same cluster.

As finding the correct number of unique instances described in the web tables is important, finding the correct number of clusters is important. We therefore penalize the clustering precision, as is also suggested by [17], if the number of returned clusters deviates from the correct number of clusters. We penalize by multiplying the clustering precision by a penalizing factor. This factor is computed by finding the sizes of C, G or M and dividing lowest by the highest size. We take this penalized clustering precision as the main precision-based score to evaluate our clustering.

Results and Lessons Learned

We will first evaluate the effectiveness of the individual row similarity metrics, and afterwards take a look at the effect of different aggregation methods as well as the blocking.

Table 7 shows the average performance of various row clustering methods using the third aggregation method, which combines random forest and weighted average. The first row contains the results when using only the LABEL metric. For every following row we aggregate one additional similarity metric. The last column of the table shows the metric importance, which is the average of the relative importance of the metric inside the learned random forest regression tree and the weights in the learned weighted average function. The importance scores shown are derived for the method that aggregates all metrics, i.e. the one that corresponds to the last row of the table.

The table shows that the similarity of row labels is the best indicator if two rows describe the same instance, as it has the highest average metric importance of 0.33 and with it we are able to achieve a moderate F1 of 0.76. At the same time, it alone is not enough and all other similarity metrics positively impact the row clustering performance when aggregated. This applies especially to the metrics BOW, ATTRIBUTE and IMPLICIT_ATT, which increase the F1 score by 2 percentage points each.

Both PHI and SAME_TABLE have smaller effects, as both only increase precision by 1 percentage point without an effect on recall. PHI likely achieves a low impact because it does not measure the similarity of two rows directly, but rather compares their tables. On the other hand, the same applies to IMPLICIT_ATT and it has a significantly higher impact. This shows the likely benefit of utilizing the knowledge base as background knowledge.

From the overall results, we can conclude that the best approach is to aggregate multiple metrics, thereby combining the different signals exploited by the individual metrics. The LABEL method utilizes the output of the label attribute detection, while the ATTRIBUTE method utilizes the knowledge base as background knowledge to semantically understand the attributes of the table. The IMPLICIT_ATT also exploits a knowledge base, but to assign semantic property-value combinations to its rows. Finally, the BOW method includes all information of a row, potentially covering information that couldn't be mapped to a schema.

The last row in Table 7 shows the clustering method that aggregates all metrics using a combination of random forest and weighted average. Applying both aggregation methods separately would have achieved an F1 score of 0.81 for weighted average and 0.82 for random forest.

Finally, the blocking yields no decrease in F1, which shows that it is an effective approach with minimal loss in recall.

3.3 Entity Creation

The entity creation component receives clusters of rows and transforms each cluster into an entity. First, an entity consists of one or more labels, which we extract from the label attribute of the entity's rows. More importantly, an entity contains a set of values mapped to the properties of the knowledge base. Given that at the row-level, each row can have multiple values matched to certain knowledge base properties, and that we have multiple rows in a cluster, there are likely to be multiple candidate values for one property when creating an entity. We therefore apply the following four-step method to fuse candidate values:

(1) **Scoring:** We score candidate values using one of the three alternative approaches described below.
(2) **Grouping:** We group equal values together. This is done using the data type specific similarity functions.
(3) **Selection:** We then select the group with the highest sum of individual candidate value scores.
(4) **Fusion:** We fuse a group into a fused value by using data type specific fusers. For text and instance reference types we utilize the majority value in a group, whereas for quantity and date types we use a weighted median approach. For nominal string and nominal integer, no fusion is necessary, as all values in a group will be equal.

We test three different scoring approaches. VOTING assigns all candidate values equal scores of 1.0. In the KBT [13] approach we measure for a certain table attribute the correctness of its overlapping values, i.e. those matched to an existing fact in the knowledge base, to estimate the trustworthiness of the whole attribute. Finally, the MATCHING approach utilizes the scores attached to a column by the attribute-to-property component. We measure the effect of the scoring approach using the output at the end of the pipeline. The results are discussed in Section 4.2.

The methods presented here are similar to those in our earlier works [22, 23, 27]. In this work, we additionally apply scores derived from the schema matching component for fusion.

3.4 New Detection

After creating entities from row clusters, we now determine whether a created entity describes a new instance, not yet present in the knowledge base. This is done by attempting to match the entities to existing instances by exploiting various features through entity-to-instance similarity metrics. If there are no instances found or the distance between an entity and an instance is large enough, the entity is determined to be new.

Additionally, for entities not classified as new, we attempt to match them to an existing instance in the knowledge base. These correspondences to existing instances are fed back into a second iteration of the pipeline to refine the schema mapping.

Our new detection approach consists of three steps:

Candidate Selection: We find a list of candidate instances from the knowledge base using a Lucene index built from the labels of knowledge base instances. To search candidates for an entity we utilize the labels attached to the entity in the entity creation component. Additionally candidates found must be of the class of the created entity or share one parent class.

Similarity Score Computation: We compute a score to measure the similarity between the created entity and a candidate instance. Multiple entity-to-instance similarity metrics are implemented and tested below.

Classification: If the highest similarity for any candidate instance is lower than a learned threshold, we classify the entity as new. Otherwise we find the candidate instance with the highest similarity score, and in case its score is higher than another threshold, the entity is classified as existing and a correspondence from the entity to that instance is generated.

In earlier work we presented methods that are solely concerned with matching rows to existing instances of a knowledge base, whereas with the methods presented here, we also determine whether rows have a match at all. Additionally, we do not match rows directly, but first create entities from row clusters, which allows us to exploit more information for matching.

Entity-To-Instance Similarity Metrics

As described above, the following metrics compute a similarity score between a created entity, and a one candidate instance of the knowledge base. We implement overall six different metrics:

- **LABEL:** We compute the similarity between the labels of the created entity and the labels of the candidate instance

using Monge-Elkan with Levenshtein as the inner similarity function.

- **TYPE:** In DBpedia every class is part of a hierarchy with a certain number of parent classes. We compute the overlap of the classes of the candidate instance with the class of the entity and its parent classes.
- **BOW:** We create a bag-of-words binary term vector for the entity by combining the vectors of all its rows, which themselves are created as described in the row clustering approach. We then create a vector for the candidate instance in the knowledge base, using its labels, abstract and facts. We return the cosine similarity of both vectors.
- **ATTRIBUTE:** For each property, where a fact exists in both the created entity and the candidate instance in the knowledge base, we determine if the fused fact is equal to the fact in the knowledge base. As there could be multiple overlapping properties, we return an average similarity and a corresponding confidence score, which equals the number of overlapping properties.
- **IMPLICIT_ATT:** We utilize the implicit attributes derived for tables, as described in Section 3.2, to derive implicit attributes for a created entity. We sum up the confidence scores of equal implicit attributes for the tables of all rows in the entity and divide by the total number of rows to compute an entity-level confidence score. We then compare these property-value combinations at the entity level with overlapping facts of a candidate instance.
- **POPULARITY:** We use a dataset of Wikipedia page links to rank all candidate instances of an entity by their number of incoming page links. A similarity score is assigned to each candidate based on its rank. If an entity has only one candidate instance, we assign it a score of 1.0.

Similarity Score Aggregation
We aggregate various similarity scores using the same aggregation approaches utilized for row clustering.

Evaluation
We evaluate the new detection component using the clusters annotated in the gold standard. Before we run new detection on those clusters, we create entities from them as outlined in Section 3.3 above. From the gold standard we know whether a certain created entity describes a new instance or not. In case it describes an existing instance, we additionally know from the gold standard the exact instance it describes.

When running the new detection component on those entities, we receive a set of entities classified as new, and another set classified as existing with additional correspondences to existing instances in the knowledge base. We can now use the gold standard to determine the accuracy of those classifications, which equals the fraction of correctly classified entities. Existing entities must additionally be matched to the correct instance in the knowledge base to be counted as correctly classified.

As the accuracy measures the classification performance for both the new and existing instances, we additionally evaluate both separately using F1. The precision of the new entities equals the fraction of entities returned with a new classification that were correctly classified as new, whereas recall is equal to the fraction of total new entities in the gold standard that were correctly classified as new. The same applies to the existing entities, with a second condition that the entity must be matched to the correct instance in the knowledge base as well.

Table 8: Average performance and metric importance scores for alternative new detection methods.

Run	ACC	$F1_{\text{Existing}}$	$F1_{\text{New}}$	MI
LABEL	0.69	0.66	0.67	0.20
+ TYPE	0.79	0.75	0.82	0.26
+ BOW	0.85	0.84	0.83	0.17
+ ATTRIBUTE	0.85	0.86	0.84	0.20
+ IMPLICIT_ATT	0.88	0.87	0.89	0.11
+ POPULARITY	0.89	0.88	0.88	0.06

Results and Lessons Learned
Table 8 shows the performance of various new detection methods. All numbers in table are averages of all classes and folds. The first row shows a method that only utilizes the label. For each following row we aggregate an additional similarity metric into the method using the combined aggregation approach. The metric importance shown in the last column reflects a score derived from the random forest and the weights in a method that aggregates all metrics, i.e., the method described in the last row.

From the table we can see that with an accuracy of 0.89, the final aggregated method performs quite well. It achieves a performance considerably better than the LABEL method, which only has an accuracy of 0.69. Additionally we can see that all similarity metrics are important and contribute positively to the overall performance. This shows that the combined approach is able to leverage the different features exploited by the individual metrics. The LABEL metric does however have a high importance score, even though the candidate selection already only returns candidates with similar labels.

The later a metric is aggregated, the more difficult it is to yield a large absolute increase in performance. As such, the metrics TYPE and BOW increase accuracy by 10 and 6 percentage points respectively, which is much larger than for metrics added later. At the same time we see, that ATTRIBUTE, which has a higher importance score than BOW, does not increase accuracy at all. It does however increase the individual F1 scores.

Noticeable is also the increase achieved by the IMPLICIT_ATT metric. It is able to improve the accuracy by further 3 percentage points, even though it is the second to last metric to be aggregated. This means, that we are successfully able to leverage the knowledge base as background knowledge to derive semantically relevant property-value combinations per table and entity.

The POPULARITY metric only impacts performance for the Settlement class. This means that, given just the name of a settlement, it is safe to assume that the most well-known settlement is meant. This makes sense, as this assumption is typically made when speaking about cities in general.

The last row shows the performance of a new detection method that aggregates all metrics using a combined approach of weighted average and random forest. When using the aggregation methods separately, we achieve an accuracy of 0.85 and 0.86 respectively. The combined approach is therefore able to exploit both aggregation methods to achieve a higher performance.

4 OVERALL RESULTS ON THE GOLD STANDARD

In this section, we choose the best methods for clustering and new detection, and evaluate the output of a complete run of our system using our gold standard. We will first evaluate how

Table 9: Results of new instances found evaluation.

Class	Clust.	New Det.	P	R	F1
GF-Player	**GS**	**ALL**	0.89	0.95	0.91
GF-Player	**ALL**	**ALL**	0.82	0.95	0.87
Song	**GS**	**ALL**	0.92	0.88	0.90
Song	**ALL**	**ALL**	0.72	0.72	0.72
Settlement	**GS**	**ALL**	0.84	0.90	0.87
Settlement	**ALL**	**ALL**	0.74	0.87	0.80
Average	**ALL**	**ALL**	0.76	0.85	0.80

Table 10: Results of the facts found evaluation.

Class	Clust.	New Det.	F1 VOTING	F1 KBT	F1 MATCHING
GF-Player	**GS**	**GS**	0.82	0.82	0.82
GF-Player	**GS**	**ALL**	0.81	0.81	0.81
GF-Player	**ALL**	**ALL**	0.81	0.81	0.81
Song	**GS**	**GS**	0.80	0.81	0.81
Song	**GS**	**ALL**	0.74	0.73	0.74
Song	**ALL**	**ALL**	0.67	0.69	0.68
Settlement	**GS**	**GS**	0.98	0.98	0.98
Settlement	**GS**	**ALL**	0.93	0.93	0.93
Settlement	**ALL**	**ALL**	0.91	0.91	0.91
Average	**ALL**	**ALL**	0.80	0.80	0.80

many of the new instances were correctly found, while in the second part, we will evaluate how many correct facts were found. Throughout this evaluation we utilize three fold cross-validation.

4.1 New Instances Found Evaluation

To evaluate how well new instances were found, we utilize precision and recall. First, we determine the number of new instances annotated in the testing sets, for which an entity was correctly returned by the system. For this, three conditions must be met. First, a majority of the rows of an entity must correspond to the same new instance in the gold standard, while at the same time the entity must also contain the majority of the rows that actually describe that instance. Lastly, the entity must be classified as new by the new detection component. Based on this, recall is defined as the fraction of new instances in the gold standard for which a correct entity was returned. Precision, on the other hand, is the fraction of entities returned by the system as new, that correctly match an instance in the gold standard.

Table 9 shows the performance of our system for the three classes separately. To evaluate the individual impact of row clustering and new detection, we once evaluate using the clustering from the gold standard, denoted GS, and once with the aggregated clustering method containing all similarity metrics, denoted ALL. For new detection we run in both cases the aggregated method containing all similarity metrics, also denoted ALL.

Generally, we achieve good performance for football players and settlements, while for songs the performance is less convincing. This is likely, because for songs, the homonym problem is much larger. It is much more likely that there exist songs of the same name by various artists. Sometimes these homonyms even represent cover versions, so that they are highly similar in their descriptions, e.g. in runtime or writer.

When investigating how much performance is lost by the individual components, we find that for football players and settlements the new detection component is causing a larger decrease than the row clustering. More specifically, the drop is mainly caused by classifying entities as new even though they should be matched to existing instances. This is confirmed by the recall being higher than precision. For songs, the clustering causes a much larger decrease in performance, showing that the clustering task is more difficult for songs, and that we require more sophisticated clustering methods.

4.2 Facts Found Evaluation

In this section we evaluate how well we can generate facts from web tables for new entities. Again, we need a mapping between entities returned and instances in the gold standard, for which we utilize the approach described above. For wrongly created entities, or entities incorrectly determined to be new, i.e. they could not be mapped to a new cluster in the gold standard, their facts are counted as wrong and therefore reduce precision. To determine if returned facts for matched entities are correct, they are compared to the facts in the gold standard using data type specific similarity functions and a learned tolerance range. To measure recall, we utilize the number of annotated facts for which a correct value is present in the web tables, as seen in Table 5.

Table 10 shows fusion performance per class. In order to measure the individual impact of the new detection and the row clustering on the overall performance, we again perform multiple runs. For the first run, we utilize for both components the correct annotations from the gold standard. In the following run we use our new detection methods, while for the third run we additionally use our clustering methods, in both cases using the methods that aggregate all similarity metrics. Additionally we test performance for the different fusion scoring methods, `VOTING`, `KBT` and `MATCHING`, described in Section 3.3.

From the table we can see that for football players and songs we lose 18 and 19 percentage points of F1 score even when row clustering and new detection are perfect. We looked at a sample of errors and were able to identify two main causes. The largest amount of errors is due to the attribute-to-property matching component, where the proportion of errors caused by wrong or missing column matches makes up 43 % of all errors. This is followed by 35 % of errors that occurred as a result of wrong or outdated data in the tables.

In addition, we evaluate various fusion scoring approaches. We find that all performances are very close, so that the choice of scoring approach is of low relevance.

Finally, we can deduce from the table the individual impacts of the pipeline components on the performance. The largest negative impact is due to errors in finding facts as described above, of which the attribute-to-property matching is a large contributor. Not as large are the individual impacts of the row clustering and new detection components, but they are still significant ranging from 1 to 13 percentage points. Overall, the way errors compound throughout the pipeline shows the difficult nature of the task at hand. Every individual component has to perform very well for there to be good performance at the end of the pipeline.

5 LARGE-SCALE PROFILING

In this section we try to estimate the potential of web tables for extending knowledge bases with new instances, by running our

system on not only the tables of the gold standard, but on all tables of a specific class within the whole corpus (see Table 4). We are especially interested in how many new instances we can correctly add per class and how that compares to the number of existing instances in the knowledge base. We additionally are interested in the descriptions of these new instances, the number and accuracy of facts, and property densities of the new instances.

Evaluation

From the entities returned as new by the system, we pull a stratified sample of 50 entities for each class. We group the returned entities by the number of facts generated for each entity. We then pull from each group a number of entities proportional to the size of the group in relation to the total number of new entities.

The accuracy of new entities equals the fraction of entities that were correctly identified as new when compared to the 2014 release of DBpedia, while the accuracy of facts equals the proportion of facts within those new entities that are correct.

Results and Lessons Learned

Table 11 shows the results of the large-scale profiling per class. The second column lists the number of rows of all tables matched to the given class. The three columns afterwards describe existing entities found, to how many unique instances in the knowledge base they were matched, and the ratio of the two numbers. The remaining columns contain the number of new entities, their facts, the relative increases when compared to Table 1 and the accuracies of new entities and facts of the extracted sample.

First of all, we find that the ratio of existing entities to matched instances in the knowledge base differs by class. While for players and settlements the number is good, it is less so for songs. Song was the class with the worst performance at row clustering, i.e. identifying the exact number of unique instances. This shows, that we need to implement more sophisticated row clustering methods or, alternatively, perform deduplication after clustering.

For the class Song we have a very large number of new entities and facts, even if we would correct the number of new entities by the ratio in the fifth column. For Settlement, there are in comparison very few new entities. When considering that only 26 % of them are correct, we would actually achieve a relative increase in knowledge base instances of 0.3 %. The difference can be explained by understanding the notability rules of Wikipedia.

There are a very large number of obscure songs. It is very common, even for well-known artists, to release for example only a few songs from an album as singles, which are the only ones that become popular. And here the notability rules compound the issue, as songs only receive their own Wikipedia article if they are notable, e.g. because they were independently released.

For Settlement, almost the opposite is true. While there are many small villages, they are never irrelevant, as there are always enough people living in them, who might contribute to a Wikipedia article. More importantly, Wikipedia deems any place notable if it has legal recognition. As a result, Wikipedia covers a lot of settlements, and it is difficult to find new ones.

Football players are in the middle of both classes. There are not as many obscure football players, as, theoretically speaking, the number of teams is limited, but there are many that are obscure enough, not to be covered in a Wikipedia article. And while the absolute number of newly added players is not high, compared to the number of existing instances in the knowledge base, we achieve an increase of 67 % for instances and 32 % for facts.

Table 12 shows the property densities for new entities. As one would expect, the properties are not as dense as in Table 2. More importantly, the distribution of densities differs significantly. For football players, personal properties like birthDate and birthPlace have a very low density for new instances, but high for the knowledge base. This might be, because in Wikipedia one is interested in describing a person, whereas in web tables, the games, teams and drafts are more in focus. For those tables, a property like position might be more relevant, which explains why its density is even higher than in the knowledge base.

For songs, the properties writer, genre, and record label have very low densities compared to the knowledge base. It is likely that for genre, this is a column matching issue, as song genres are not always objectively defined. For writer and recordLabel there could be two causes. First, they might be uninteresting properties, and secondly, there are often multiple correct facts. The record label might even differ by country. This makes these properties difficult to match, and, more importantly, unlikely to be included in a table. We can confirm the latter, as these properties occurred very rarely in the tables we annotated for the gold standard.

When looking at the accuracies of new entities we also find differences per class. We achieve a moderate accuracy for songs, a sub-par accuracy for players and a low accuracy for settlements.

The primary reason for the low accuracy for settlements are conflicting values in an entity of an existing instance and the instance in the knowledge base. This includes outdated population numbers, but also isPartOf values, where the values in the entity and in the knowledge base are both correct, but different, preventing the entity from matching. This problem makes up 36 % of all errors. 25 % of errors are because the new entity does not describe a settlement, but a different place, like a region or a mountain. This error is caused by incorrect table-to-class matching. These problems are magnified because there are so few new entities to begin with, so that these corner cases make up a huge proportion of the new entities returned.

For GF-Player, the sources of errors include bad column-to-attribute matching, entities not being football players due to bad table-to-class matching, and incomplete information in DBpedia. The latter happened primarily when a football athlete was not assigned the correct class in DBpedia. The accuracy for entities with a higher number of values is however much higher. If we do not consider entities with one value, the accuracy of new entities rises to 0.72. If we further do not consider entities with two values, we achieve an accuracy of 0.85. This would mean excluding 6,360 entities, but also that with an accuracy of 0.85 we can add 7,623 entities with 34,922 facts to the knowledge base, an increase of 37 % for instances, and 25 % for facts.

For songs, the sources of errors are versatile. The main contributors are bad new detection, incorrect table-to-class matching, and bad clustering. The latter meaning that an entity was incorrectly detected as new, as a result of being created from rows that describe different instances.

We generally notice that the performance does not correlate with the performance on the gold standard. This might indicate that the gold standard either does not completely reflect the nature of the task, or the gold standard is not large enough. On the other hand, we achieve a consistently high accuracy for new facts, similar to the high performance on the gold standard, as seen in Section 4.2. This means that when it comes to finding descriptions, our performance is quite good, even if the density is lower when compared to the knowledge base.

Table 11: Results and evaluation of a system run on all tables matched to a class.

Class	Total Rows	Existing entities	Matched KB instances	Matching Ratio	New Entities	New Facts	N. Entities Accuracy	N. Facts Accuracy
GF-Player	648,741	30,074	24,889	1.21	13,983 (+67%)	43,800 (+32%)	0.60	0.95
Song	2,173,536	40,455	29,140	1.39	186,943 (+356%)	393,711 (+125%)	0.70	0.85
Settlement	1,472,865	28,628	27,365	1.05	5,764 (+1%)	7,043 (+0%)	0.26	0.94

Table 12: Property densities for new entities returned by the full run.

Class	Property	Facts	Density
GF-Player	**position**	9,204	65.82%
GF-Player	**team**	7,637	54.62%
GF-Player	**college**	6,849	48.98%
GF-Player	**weight**	5,915	42.30%
GF-Player	**height**	4,253	30.42%
GF-Player	**number**	2,951	21.10%
GF-Player	**birthDate**	2,537	18.14%
GF-Player	**draftPick**	2,404	17.19%
GF-Player	**draftRound**	1,538	11.00%
GF-Player	**draftYear**	386	2.76%
GF-Player	**birthPlace**	126	0.90%
Song	**musicalArtist**	143,656	76.84%
Song	**runtime**	115,652	61.86%
Song	**album**	52,664	28.17%
Song	**releaseDate**	47,377	25.34%
Song	**genre**	23,814	12.74%
Song	**recordLabel**	10,278	5.50%
Song	**writer**	270	0.14%
Settlement	**isPartOf**	2,889	50.12%
Settlement	**postalCode**	1,605	27.85%
Settlement	**country**	1,232	21.37%
Settlement	**populationTotal**	1,214	21.06%
Settlement	**elevation**	103	1.79%

Overall we find that for some classes there is high potential for finding new instances using web tables. Additionally, we also find that the performance of our system for these classes is generally good. Yet, it is clear that more sophisticated approaches are necessary for row clustering, new detection and table-to-class matching.

6 RELATED WORK

This section compares our method to the related work. As we investigate a new problem, we compare to research on similar tasks as well as on specific subtasks, including slot filling, set expansion, schema matching and identity resolution.

Slot Filling

Many works that exploit web table data for knowledge base augmentation [11, 25–29] focus on the task of slot filling, i.e. adding missing facts for existing instances. One prominent state of the art work by Dong et al. [11] introduces a probabilistic approach that exploits background knowledge, in their case Freebase, to construct a large knowledge base using web data, including web tables. The extracted facts however only describe instances that already exist in Freebase [11, 12].

While slot filling is a different task, we can still generally compare the numbers of generated facts. In a previous work, where we use web tables to perform slot filling for DBpedia [27], we are able to find 378, 892 facts, 64, 237 of which are new facts for existing instances. We reach an F1 score of 0.71. Compared to that work, we are able to achieve better numbers and accuracy. In the work by Dong et al., the authors are able to find 271 M facts for instances in Freebase with an expected correctness higher than 90 %. Of those facts, 90 M are new facts for existing instances. While the amount of facts that we discover are much smaller, we are only dealing with three classes and looking at facts only for new instances. Additionally we only use web tables to extract new facts, while Dong et al. also use free text, HTML DOM trees and schema.org annotations. Our average accuracy of 0.91 for facts of new entities is comparable.

Set Expansion

Set expansion is a task, where new instances are retrieved to complete a set [24, 31, 32]. Set expansion methods, however, only focus on finding the labels of new instances. The methods rely on seeds from that set to perform set expansion. Both, the complete sets and the number of seeds, are often small. In contrast, we focus on a scenario in which large sets of entities, already contained in the knowledge base, are extended with potentially large sets of new instances. Finally, most set expansion methods make use of ranked evaluation, where the precision of the top k instances is measured. In contrast, our evaluation not only focuses on precision, but also considers recall.

One set expansion method exploits a corpus of relational tables to augment an incomplete relational table with new instances and their descriptions [33]. The methodology differs significantly from ours. To find more instances, the method uses sets of 1 to 5 seed instances to first search for candidates in the tables. For this, it exploits the labels of the seeds and the caption of the seed table. Candidates are then ranked based on how often they co-occur with the seeds and how similar their tables' captions are. Similarly, the method searches for candidate columns in the corpus and ranks them based on how well they fit the seed table. The method then returns a fixed number of entities, where the authors use 256 as their cut-off. While this approach does generate descriptions, it does not resolve any of the following problems of set expansion. Their approach still ranks candidate entities based on their similarity to the seeds, and, more importantly, always returns a fixed number of instances. Especially the latter makes this approach not applicable to our task, as we are interested in generating as many new instances as possible.

To compare our work with works of set expansion, we need to utilize ranked evaluation and therefore need to implement a ranking algorithm for new entities. We rank based on the similarity scores returned by the new detection. These scores measure the distance between one or more existing instance in the knowledge base, and an entity generated from the web tables. We rank new

entities higher, the higher their lowest distance to the closest existing instance is. Using this, we achieve a MAP, with a cut-off at 256, of 0.88, while related works achieve 0.63 [33], 0.95 [32] and 0.78 [31]. For precision at 5 and 20 we achieve 0.84 and 0.78 respectively, while a related work achieves 0.94 and 0.91 [33]. We find that our performance is comparable, even though the task we are solving is more difficult.

Schema Matching and Identity Resolution
Throughout the components of the pipeline, we apply approaches for which a large corpus of related work exists. This includes schema matching methods, which are surveyed in [3]. For row clustering and new detection we essentially exploit identity resolution methods, which are extensively surveyed in [9, 14].

For the specific use case of matching web table attributes to DBpedia properties, authors from our research group were able to achieve an F1 score of 0.81 [25]. While our performance of 0.92 is higher, we consider a smaller number of classes and properties.

There exists a large corpus of research on the task of matching web table rows to existing knowledge base instances. While this task is not a primary objective of this paper, we are able to evaluate our pipeline by comparing how well we are able to perform this task. We achieve an average F1 score of 0.83, compared to 0.80 [25] and 0.87 [34] in the related work, and we achieve an accuracy of 0.78, compared to 0.83 [21] and 0.93 [4] in the related work. While for F1, our performance is comparable to the related work, it is lower when it comes to accuracy.

7 CONCLUSION

This paper explored the potential of web table data for extending a cross-domain knowledge base with new long tail entities and their descriptions according to the schema of the knowledge base. To the best of our knowledge, this specific task has to this date not been handled in related research. For this task, we present and evaluate a complete system. It consists of a pipeline with multiple components, including schema matching, row clustering, entity creation and new detection.

We evaluated our pipeline using a manually annotated gold standard of web tables. We find that the task is non-trivial, as it requires good performance in all steps of the process. For all components of the pipeline we implemented and evaluated multiple alternative methods. We find that aggregating the similarity scores of multiple metrics that exploit different features yields the best results. We also find that metrics that make use of label similarity, while highly important, are not sufficient to yield a good performance. Additionally, we are able to show that metrics that use the knowledge base as background knowledge, e.g. to semantically understand cell values or to derive semantic information about web tables, have a positive impact on performance. Finally, we are able to utilize the output of the pipeline in a second iteration to achieve a large improvement in schema matching performance, while any further iteration has negligible impact.

We are successfully able to utilize our pipeline and our proposed implementations of the pipeline's components to find new instances and their descriptions from the web tables. At the same time, there remains room to further improve the quality of the generated data.

Finally we run the method on the complete web table corpus to profile the overall potential of web tables in augmenting a knowledge base class with new instances. We find that this potential differs per class, but at the same time, we find that for some classes a large number of instances and facts with a high accuracy can successfully be added to the knowledge base.

REFERENCES

[1] Nir Ailon, Moses Charikar, and Alantha Newman. 2008. Aggregating Inconsistent Information: Ranking and Clustering. *JACM* 55, 5 (2008), 23:1–23:27.
[2] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. 2004. Correlation Clustering. *Machine Learning* 56, 1 (2004), 89–113.
[3] Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. 2011. Generic Schema Matching, Ten Years Later. *VLDB* 4, 11 (2011), 695–701.
[4] Chandra Sekhar Bhagavatula, Thanapon Noraset, and Doug Downey. 2015. TabEL: Entity Linking in Web Tables. In *ISWC '15*.
[5] Alexander Bilke and Felix Naumann. 2005. Schema Matching using Duplicates. In *ICDE '05*. 69–80.
[6] Francesco Bonchi, David Garcia-Soriano, and Edo Liberty. 2014. Correlation Clustering: From Theory to Practice. In *KDD '14*.
[7] Leo Breiman. 2001. Random Forests. *Machine Learning* 45 (2001), 5–32.
[8] Michael J. Cafarella, Alon Y. Halevy, Yang Zhang, Daisy Zhe Wang, and Eugene Wu. 2008. Uncovering the Relational Web. In *WebDB '08*.
[9] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. 2015. *Entity Resolution in the Web of Data*. Morgan & Claypool Publishers.
[10] Erik D. Demaine, Dotan Emanuel, Amos Fiat, and Nicole Immorlica. 2006. Correlation clustering in general weighted graphs. *Theoretical Computer Science* 361, 2 (2006), 172 – 187.
[11] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. 2014. Knowledge Vault: A Web-scale Approach to Probabilistic Knowledge Fusion. In *KDD '14*.
[12] Xin Luna Dong. 2016. How Far Are We from Collecting the Knowledge in the World. In *ICWE'16 Keynote*.
[13] Xin Luna Dong, Evgeniy Gabrilovich, Kevin Murphy, Van Dang, Wilko Horn, Camillo Lugaresi, Shaohua Sun, and Wei Zhang. 2015. Knowledge-based Trust: Estimating the Trustworthiness of Web Sources. *VLDB* 8, 9 (2015), 938–949.
[14] Xin Luna Dong and Divesh Srivastava. 2015. Record Linkage. In *Big Data Integration*. Morgan & Claypool Publishers.
[15] Micha Elsner and Eugene Charniak. 2008. You Talking to Me? A Corpus and Algorithm for Conversation Disentanglement. In *ACL-08: HLT*.
[16] Micha Elsner and Warren Schudy. 2009. Bounding and Comparing Methods for Correlation Clustering Beyond ILP. In *ILP '09*.
[17] Oktie Hassanzadeh, Fei Chiang, Hyun Chul Lee, and Renée J. Miller. 2009. Framework for Evaluating Clustering Algorithms in Duplicate Detection. *VLDB* 2, 1 (2009), 1282–1293.
[18] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. 2013. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence* 194 (2013), 28–61.
[19] Margret Keuper, Evgeny Levinkov, Nicolas Bonneel, Guillaume Lavoue, Thomas Brox, and Bjorn Andres. 2015. Efficient Decomposition of Image and Mesh Graphs by Lifted Multicuts. In *ICCV '15*.
[20] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. 2015. DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.
[21] Girija Limaye, Sunita Sarawagi, and Soumen Chakrabarti. 2010. Annotating and Searching Web Tables Using Entities, Types and Relationships. *VLDB* 3, 1-2 (2010), 1338–1347.
[22] Yaser Oulabi and Christian Bizer. 2017. Estimating missing temporal meta-information using Knowledge-Based-Trust. In *KDWeb '17*.
[23] Yaser Oulabi, Robert Meusel, and Christian Bizer. 2016. Fusing Time-dependent Web Table Data. In *WebDB '16*.
[24] Patrick Pantel, Eric Crestan, Arkady Borkovsky, Ana-Maria Popescu, and Vishnu Vyas. 2009. Web-scale Distributional Similarity and Entity Set Expansion. In *EMNLP '09*.
[25] Dominique Ritze and Christian Bizer. 2017. Matching Web Tables To DBpedia - A Feature Utility Study. In *EDBT '17*.
[26] Dominique Ritze, Oliver Lehmberg, and Christian Bizer. 2015. Matching HTML Tables to DBpedia. In *WIMS '15*.
[27] Dominique Ritze, Oliver Lehmberg, Yaser Oulabi, and Christian Bizer. 2016. Profiling the Potential of Web Tables for Augmenting Cross-domain Knowledge Bases. In *WWW '16*.
[28] Yoones A. Sekhavat, Francesco Di Paolo, Denilson Barbosa, and Paolo Merialdo. 2014. Knowledge Base Augmentation using Tabular Data. In *LDOW '14*.
[29] Mihai Surdeanu and Heng Ji. 2014. Overview of the English Slot Filling Track at the TAC 2014 Knowledge Base Population Evaluation. In *TAC '14*.
[30] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: A Free Collaborative Knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85.
[31] Chi Wang, Kaushik Chakrabarti, Yeye He, Kris Ganjam, Zhimin Chen, and Philip Bernstein. 2015. Concept Expansion Using Web Tables. In *WWW '15*.
[32] Richard C. Wang and William W. Cohen. 2007. Language-Independent Set Expansion of Named Entities Using the Web. In *ICDM '07*.
[33] Shuo Zhang and Krisztian Balog. 2017. EntiTables: Smart Assistance for Entity-Focused Tables. In *SIGIR '17*.
[34] Ziqi Zhang. 2017. Effective and Efficient Semantic Table Interpretation using TableMiner(+). *Semantic Web* 8, 6 (2017), 921–957.