SchemaTree: Maximum-Likelihood Property Recommendation for Wikidata

Lars C. Gleim[1](✉), Rafael Schimassek[1], Dominik Hüser[1], Maximilian Peters[1], Christoph Krämer[1], Michael Cochez[2,3], and Stefan Decker[1,3]

[1] Chair of Information Systems, RWTH Aachen University, Aachen, Germany
{gleim,decker}@dbis.rwth-aachen.de
[2] Department of Computer Science, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
m.cochez@vu.nl
[3] Fraunhofer Institute for Applied Information Technology FIT, Sankt Augustin, Germany

Abstract. Wikidata is a free and open knowledge base which can be read and edited by both humans and machines. It acts as a central storage for the structured data of several Wikimedia projects. To improve the process of manually inserting new facts, the Wikidata platform features an association rule-based tool to recommend additional suitable properties. In this work, we introduce a novel approach to provide such recommendations based on frequentist inference. We introduce a trie-based method that can efficiently learn and represent property set probabilities in RDF graphs. We extend the method by adding type information to improve recommendation precision and introduce backoff strategies which further increase the performance of the initial approach for entities with rare property combinations. We investigate how the captured structure can be employed for property recommendation, analogously to the Wikidata PropertySuggester. We evaluate our approach on the full Wikidata dataset and compare its performance to the state-of-the-art Wikidata PropertySuggester, outperforming it in all evaluated metrics. Notably we could reduce the average rank of the first relevant recommendation by 71%.

Keywords: Wikidata · Recommender systems · Statistical property recommendation · Frequent pattern mining · Knowledge graph editing

1 Introduction

Wikidata is a free and open knowledge base which acts as central storage for the structured data of several Wikimedia projects. It can be read and edited by both humans and machines. Related efforts are schema.org [15] and Linked Open Data[1] [7]. Manual editing of knowledge-bases is traditionally an error prone process [23] and requires

[1] We provide additional results for the LOD-a-lot dataset [12] together with our implementation in the supplementary material at https://github.com/lgleim/SchemaTreeRecommender.

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612.

© Springer Nature Switzerland AG 2020
A. Harth et al. (Eds.): ESWC 2020, LNCS 12123, pp. 179–195, 2020.
https://doi.org/10.1007/978-3-030-49461-2_11

intimate knowledge of the underlying information model. Even entities of semantically equal type regularly feature different property sets (also called the attributes or predicates of the entity in the context of RDF [8]), different property orderings, etc. [9].

For Wikidata, much care is taken to create useful properties, which have support from the community[2]. Nevertheless, due to the sheer number of available properties, users often struggle to find relevant and correct properties to add to specific entities of the knowledge base. In order to improve the process of manually incorporating new facts into the knowledge base, the Wikidata platform provides the PropertySuggester tool, which recommends suitable properties for a given subject using an association rule-based approach [25]. Similar recommendation approaches are also employed in more general RDF recommender systems and collaborative information systems [1,2,13,22,26].

The main contribution of this work is the *SchemaTree* in which we make use of frequentist inference to recommend properties; in particular using a compact trie-based representation of property and type co-occurences. We also detail how the approach qualitatively differs from the existing association rule-based approaches, investigate cases in which the baseline SchemaTree does not perform well, and present respective improvement strategies. The results in Sect. 4 show that the recommender performs well on the Wikidata dataset and significantly outperforms the current Wikidata PropertySuggester.

In the following, we first introduce relevant related work in property recommendation systems and frequent pattern learning and discuss potential limitations of these existing systems, before detailing the construction of the SchemaTree and its application to property recommendation. Afterwards, we present an extension of the baseline SchemaTree incorporating type information into the system in Sect. 3 and present backoff strategies to deal with specific cases in Sect. 3 to improve precision and recall further. Subsequently, we evaluate the performance of the proposed approach against a state-of-the-art approach and its applicability to scale to large RDF datasets, before summarizing our results and concluding our work.

2 Related Work

Several data-driven property recommendation systems have been introduced in recent years. In the context of databases and the Web, there are several examples of works which suggest schema elements to designers. As an example, Cafarelle et al. [10] propose the attribute correlation statistics database (AcsDB), which enables property suggestion based on their cooccurences in web tables, assisting database designers with choosing schema elements. Other examples include [5,18].

Also in the context of structured knowledge bases and the Semantic Web, several approaches have been proposed. Many of these are fundamentally based upon the idea of mining association rules [4] from sets of co-occurring properties. Motivated by human abstract association capabilities, association rule-based recommendation hinges on the following underlying rationale: If a number of properties co-occur frequently, the existence of a subset of those properties allows for the induction of the remaining properties with a certain confidence.

[2] See for example the discussion at https://www.wikidata.org/wiki/Wikidata:Requests_for_comment/Reforming_the_property_creation_process.

A first example of such work is by Abedjan et al. [1,2], whose RDF enrichment approach employs association rule mining for property suggestion. In their work, recommendations are ranked by the sum of the confidence values of all association rules that respectively entailed them. That work got extended into the Wikidata recommender, which is called *PropertySuggester*[3]. The difference with the basic approach is the introduction of so-called *classifying* properties, which are the properties `instanceOf` and `subclassOf` [25]. Subsequently, association rules are not only derived based on the co-occurrence of properties but also on which properties occur on which types of instances, providing additional information for the recommendation computation process.

The Snoopy approach [13,26] is another property recommendation system based on association rules, which distinguishes itself from previous systems by ranking recommendations based on the support – i.e., the number of occurrences of a given rule – across all training data items (in contrast to the sum of confidences used in the previous approaches). Zangerle et al. [25] proposed an extension of the Snoopy approach, inspired by previous work of Sigurbjörnsson et al. [22]. They rank properties by the number of distinct rules that respectively entail them and their total support as a proxy for including contextual information into the ranking process. Zangerle et al. [25] further conducted an empirical evaluation of several state-of-the-art property recommender systems for Wikidata and collaborative knowledge bases and concluded that the Wikidata recommender approach significantly outperforms all evaluated competing systems (including their own). As such, we consider the Wikidata PropertySuggester as state-of-the-art.

Unfortunately, the process of association rule mining can result in misleading rules. Especially due to the spuriousness of the underlying itemset generation, the inability to find negative association rules, and variations in property densities [3]. As such, important information about the context of the mined association rules is lost, leading to deviations between the true conditional probabilities of property occurrence and their association rule approximations. While the previously introduced approaches apply different heuristics in order to rank recommendations based on relevant association rules, they only loosely approximate an ordering based on true likelihoods of the property co-occurrences. In this work, we investigate how a frequentist approximation of this true likelihood can improve recommendations.

Recently, Balaraman et al. [6] developed Recoin, a statistical indicator for relative completeness of individual Wikidata entities. The system can also be repurposed to propose potentially missing properties based on the class information (only). However, this cannot take into account other properties of the entity. It will only suggest properties which a sufficient fraction of other instances of the same class also has. This system has not been shown to outperform the Wikidata PropertySuggester, which includes both class membership and property information.

Dessi and Atzori [11] presented an approach applying supervised, feature-based Machine Learning to Rank algorithms to the task of ranking RDF properties. The approach focuses on flexible personalization of properties' relevance according to user preferences for specific use cases in a supervised training approach. Given this context, a direct comparison between this and the other presented approaches is not feasible.

[3] http://gerrit.wikimedia.org/r/admin/projects/mediawiki/extensions/PropertySuggester.

HARE [19] is a generalized approach for ranking triples and entities in RDF graphs, capable of property recommendation, which is based on random walks in a bi-partite graph representation. Its scalability to large datasets has however also not been shown, nor compared to the state-of-the-art.

Razniewski et al. [20] further introduced an approach incorporating human interestingness ratings of properties into the property recommendation process, outperforming the state-of-the-art with respect to agreement with human interestingness annotations. The general applicability of the approach is however hindered by the limited availability of data on human preferences of individual properties and the fact that knowledge graphs are not necessarily created to maximize the interestingness of their contents for humans but often also for algorithmic and specific technical applications.

Next, we introduce our approach for property recommendation in the context of manual knowledge-base statement authoring, based on maximum-likelihood recommendation directly employing a frequent pattern tree (FP-tree) for efficient probability computations.

3 SchemaTree: Design and Construction

In this section, we introduce the design and construction of a data structure used for efficient pattern support lookup. A Knowledge Base (KB) generally consists of entities with associated properties and values. An entity can have the same property multiple times and entities in the KB can also have type information[4].

Preliminaries. The task of recommending properties is defined as proposing a relevant property for a given entity, which was previously not attributed to it. In this work, we limit ourselves to proposing properties with respect to their maximum-likelihood as determined from a set of training data. Hence, in the scope of this paper, we define the task of property recommendation as follows:

Definition 1 (Maximum-likelihood Property Recommendation). *Given an entity E with properties $S = \{s_1, \ldots, s_n\} \subseteq \mathcal{A}$ in a Knowledge Graph KG where $\mathcal{A}$ is the set of all properties of all entities, maximum-likelihood property recommendation is the task of finding the property $\hat{a} \in \mathcal{A} \setminus S$ such that*

$$\hat{a} = \operatorname*{argmax}_{a \in (\mathcal{A} \setminus S)} P(a | \{s_1, \ldots, s_n\}) = \operatorname*{argmax}_{a \in (\mathcal{A} \setminus S)} \frac{P(\{a, s_1, \ldots, s_n\})}{P(\{s_1, \ldots, s_n\})} \tag{1}$$

where $P(\{t_1, \ldots, t_m\})$ is the probability that a randomly selected entity has at least *the properties $t_1, \ldots, t_m$.*

Intuitively, we need to find the property a which is most often observed together with the properties which the entity already has ($\{s_1, \ldots, s_n\}$). This directly corresponds to a maximum-likelihood estimation over the true probability distribution P of property co-occurrences. To obtain k recommendations, this definition can be extended such that we obtain a list of the k properties which have the highest k maximum-likelihood probabilities, as sorted by that probability.

[4] These requirements are fulfilled by both Wikidata and RDF graphs in general.

Given a sufficiently large amount of training data, the true joint probabilities can be reasonably well approximated by their relative frequency of occurrence, using a frequentist probability interpretation. We borrow the common approach of grouping RDF triples by subject (i.e. entity in a KB KG) to derive the multiset $\mathfrak{P}$ of all per subject property sets [14,24,25], formally $\mathfrak{P} = \{Q|E \in KG, Q \text{ is the set of properties of } E\}$. Then, we can determine the absolute frequency (or support count) $\text{supp}(A)$ of a set of properties $A = \{a_1,\ldots,a_{|A|}\} \subseteq \mathcal{A}$ (i.e. a pattern) as the number of subject property sets that include it:

$$\text{supp}(A) = \text{supp}(a_1,\ldots,a_{|A|}) = |\{Q \in \mathfrak{P}|A \subseteq Q\}| \tag{2}$$

Subsequently, we can determine the most likely property recommendation by reformulating Eq. (1) via frequentist inference as:

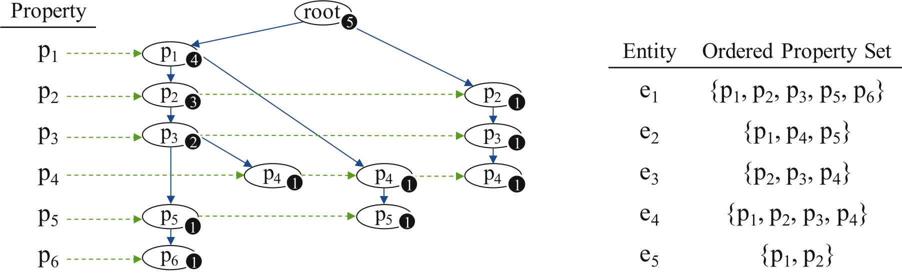
$$\hat{a} \simeq \underset{a \in (\mathcal{A} \setminus S)}{\text{argmax}} \frac{\text{supp}(a, s_1,\ldots,s_n)}{\text{supp}(s_1,\ldots,s_n)} \tag{3}$$

If we naively computed recommendations according to this definition, it would be impossible to produce these in a timely manner. This is because creating a recommendation will force us to scan trough the complete dataset, which for a realistically sized one like Wikidata will already take prohibitively long. Hence, to make the proposed technique usable, we need efficient lookup of these frequencies. However, given that the number of possible property combinations for n properties is 2^n, it is infeasible to precompute and store them in a simple lookup table. Hence, we introduce a suitable data structure which makes it possible to compute them in a short time in the next subsection.

Construction. To allow for efficient learning, storage, and retrieval of these patterns, we adapt the trie construction idea of the FP-tree, first introduced by Han et al. [17], in order to serve as a highly condensed representation of the property sets. We are not aware of prior work using this approach for frequentist inference. In contrast to common applications in association rule learning, we do not prune the tree based on minimum support but retain the full tree. While various optimized and specialized adaptations of the original FP-tree construction have been proposed in recent years (see, for example, the comparative study in [21]), we build upon the original 2-pass tree construction to enable a more transparent analysis of the tree's properties. Moreover, in order to ensure deterministic construction of the FP-tree, we do adopt the usage of a support descending property ordering together with a lexicographic order as proposed by [16]. As the tree is representing a higher level abstraction of the properties used in the KB, we call this tree the *SchemaTree*. Building the tree is done as follows:

1) For each property $a \in \mathcal{A}$, determine its support $\text{supp}(a)$ in one scan through the data and cache it. Additionally, create an empty lookup index to maintain a list of occurrences of each property within the tree, to later allow for efficient traversal of the tree.
2) Determine a fixed ordering of properties $p_1,\ldots,p_{|\mathcal{A}|}$, first by descending support and second by lexicographical ordering $\text{lex}(p_i)$.

3) Construct the prefix tree from all patterns, respectively sorted according to ordering $p_1, \ldots, p_{|\mathcal{A}|}$, by inserting the properties into the tree, starting from the root node (representing the empty set, contained in all patterns). Each node in the tree maintains a counter of its prefix-conditional support (the support for the set of properties between the root and the node in question) and a backlink to its parent. The root node thus counts the total number of patterns inserted. Whenever a new child node is created, it is additionally appended to the list of occurrences of the corresponding property.



Entity	Ordered Property Set
e_1	$\{p_1, p_2, p_3, p_5, p_6\}$
e_2	$\{p_1, p_4, p_5\}$
e_3	$\{p_2, p_3, p_4\}$
e_4	$\{p_1, p_2, p_3, p_4\}$
e_5	$\{p_1, p_2\}$

Fig. 1. The SchemaTree derived from the property sets depicted on the right.

Figure 1 illustrates the SchemaTree (left) derived from the example KB of five subjects with their respective property sets (right). Patterns are inserted starting from the root node at the top. The blue, solid arrows indicate the pattern tree hierarchy, the green, dashed arrows illustrate the links of the per-property occurrence index, depicted on the left side. The white numbers on black background denote the prefix-conditional support. Once this tree is constructed, it can be used to recommend properties.

Maximum-Likelihood Recommendation. The property recommendations for a given entity with non-empty property set $A \subseteq \mathcal{A}$ can be computed using the following procedure:

1) Make a candidate set $C = \mathcal{A} \setminus A$ of support counters for possible property recommendations and a support counter for A with respective initial support 0.
2) Sort A using property ordering $p_1, \ldots, p_{|\mathcal{A}|}$ by ascending support to get sorted properties $a_1, \ldots, a_{|A|}$, i.e. where a_1 is the least frequent property.
3) For each occurrence a'_1 of a_1 in the SchemaTree (directly retrievable via per-property occurrence index) with associated support s'_1:
   a) Check whether the remaining properties in A are contained in the prefix path (i.e. its ancestors).
   b) If yes, increment the support counter of all property candidates contained in the prefix but not already in A by s'_1, the support counter of A by s'_1 and the support counter of all property candidates that occur as part of the suffix of a'_1 (i.e. its children) by their respective occurrence support, as registered in the tree.

4) Sort the candidate set by descending support to receive the ranked list of property recommendations. The respective likelihood approximation of each recommendation can be obtained as its support divided by the support of A.

The reason all candidates occurring in the prefix of a'_1 are incremented by s'_1 in step 3 (b) and not by their respective individual occurrence support, is that they only occurred s'_1-many times together with the entire pattern A on this branch. Further, note that branches may be discarded early in step 3 (a) based on the known property ordering. More specifically, if the currently inspected prefix node has a lower sort order then the next expected node according to the sorted property set A, the expected property can no longer be encountered and the branch gets ignored immediately. Hereby, the strategy of checking prefix containment, starting with properties of minimal overall support, has a higher selectivity (i.e. specificity or true negative rate) then starting the search from the most likely properties at the root and is thus expected to lead to earlier search terminations.

Suppose we want to make property suggestions for an entity with properties $A = p_2, p_3$, based on p_i as in the SchemaTree depicted in Fig. 1. Ordering reveals p_3 to be the least frequent property. Inspection of the per-property occurrence index of p_3 reveals two occurrences in the tree, p_3^l (left) and p_3^r (right). Since the prefix of p_3^l does contain p_2, the support counters of p_1 (only candidate in the prefix) and A (i.e. the set support counter) are incremented by 2 (the support of p_3^l). The suffixes of p_3^l lead to the respective incrementation of support counters of p_4, p_5 and p_6 by their respective occurrence support of 1. Inspection of the prefix of p_3^r reveals that p_2 is also contained in its prefix, leading us to incremented A by 1 (the support of p_3^r). Since no other candidates are part of the prefix, we can directly continue with the suffix p_4, whose support counter is accordingly incremented by 1. Sorting of the candidate list and division by the support of A results in the final list of recommendations: p_1 and p_4 ($2/3 \simeq 66,67\%$ likelihood each) and p_5 and p_6 ($1/3 \simeq 33,33\%$ likelihood each). Note that we can further deduct that all other properties are unlikely to co-occur with the given set of properties. Depending on the application this knowledge may also have significant value by itself, e.g. in the context of data quality estimation. As such, the approach is also capable of capturing negative relationships, i.e. associations, between properties.

Employing Classifying Properties. The recommendation precision is expected to be limited by a lack of context information when only a small set of existing properties are provided as input to the recommender. This is especially true when these few properties are themselves rather common, since they occur together with a large number of other properties. To improve the recommender's precision in such cases, type information is integrated into the SchemaTree by employing the concept of classifying properties as implemented by the Wikidata PropertySuggester. [25] As such, any value of a classifying property can be considered a *type*. Correspondingly, any value of an instanceOf property (Property:P31) is a type in the sense of the Wikidata data model and can be extracted as such. Equivalently, it is possible to use e.g. the DBpedia type property or RDF type for generic RDF datasets.

To build the SchemaTree, we treat types as additional properties: In the first scan, we count the frequencies of properties as well as types. We create a strict totally ordered set including properties and types – again ordered first by descending support and second

by lexicographical order – and redefine the per subject property set as the ordered set of all corresponding properties and types. During the second pass, we insert all subjects' property sets (now including types) into the SchemaTree.

In the recommendation algorithm, we search for paths in the tree that contain both all properties and all types of the provided input set. When the list of recommendations is created, only properties (not types) are considered as possible candidates. Note that this makes it also possible to recommend properties for an entity that only has class information and that this approach could also be used to recommend suitable additional types for a provided input set.

Employing Backoff Strategies. Association rule-based approaches excel at generalizing to property sets not encountered in the training set, due to the typically small size of any given rule's precondition item set. The SchemaTree recommender, however, by default often fails to provide recommendations in this case, since the required lookup of the support of the provided input set and its super-sets will not return any results. To give an example, suppose that we want to compute recommendations for the input set $\{p_1, p_2, p_3, p_4\}$, given the SchemaTree depicted in Fig. 1, then p_4 is the property with the lowest support and therefore the starting point for the recommender. Only the leftmost p_4 node of the SchemaTree meets the condition that properties p_1, p_2, p_3 are on the path from p_4 to the root, so that this is the only node we regard. Unfortunately, there is no other property on that path, neither as predecessor nor successor. Therefore, the recommender does not recommend any new property to the set. Similarly, large input sets generally correlate to fewer corresponding examples in the training set and thus to tendentially less reliable recommendations of the SchemaTree recommender, while association rule-based approaches generally remain unaffected by this issue and rather suffer from the challenge of combining the tendentially many applicable association rules into a comprehensive property ranking.

In order to address these border cases, we designed two **backoff strategies**, which either reduce the set of employed input properties or split it into multiple input sets:

SplitPropertySet. Splits the input property set into two smaller input sets:

1) Sort incoming properties according to the global property support ordering $p_1, \ldots, p_{|\mathcal{A}|}$ of the SchemaTree.
2) Split the ordered property set P into 2 subsets P_1' and P_2' $(P_1' \cup P_2' = P)$. We consider two ways to perform splitting of ordered property set P:
   a) *Every Second Item.* The items are split in the sets such that each item in even position in the sorted set P comes in P_1, the others in P_2.
   b) *Two Frequency Ranges.* The first half of sorted set P is put in P_1, the last half in $P2$.
3) Perform recommendation on both subsets in parallel, obtaining two recommendations R_1' and R_2'.
4) Delete those properties from the recommendations which were in the other input property subset resulting in cleaned recommendations R_1 and R_2.
5) Merge recommendation R_1 and R_2 to for the recommendation R, which is finally returned as result of the backoff strategy. This we do, by either taking the *maximum* or the *average* of the two probabilities per individual recommended property.

DeleteLowFrequency. Reduces the size of the input property set by removing a varying number of properties with lowest support and computing recommendations for multiple such reduced input sets in parallel. In the end, one of these resulting sets of recommendations is selected. the procedure goes as follows:

1) Sort incoming properties according to the global property support ordering $p_1, \ldots, p_{|\mathcal{A}|}$ of the SchemaTree.
2) Create q subsets $P_i, i \in [q]$ by deleting the $d(i)$ least frequent items from the original input set P. Here, $d(i)$ determines the number of low frequent properties deleted form P in run i, we discuss options below.
3) Run the recommender on the subsets in parallel, obtaining recommendations sets R_i.
4) Choose the recommendation R_i with the least number of deleted properties which does no longer trigger a backoff condition.
5) Delete any recommendation already contained in the original input set P and return the remaining recommendations as the final result. We consider two possible ways to define the number of least frequent properties $d(i)$, which are deleted from P in run $i \in [p]$:
   a) *Linear Stepsize* $d_L(i) = i$, i.e. set P_i does not contain the least i properties. i.e. with every further parallel execution we remove one more item from the property set.
   b) *Proportional Stepsize* $d_P(i) = a * n * \frac{i}{q}, 0 \leq a \leq 1$. Here, n is the number of properties in P, a the largest fraction we want to remove, and q the number of runs. So, we remove up to a fraction a of the properties in q equally large steps.

The linear approach may result in many parallel executions of the recommender in cases where multiple properties have to be erased until no backoff condition are triggered anymore. In contrast, the proportional approach covers a wider range of input set reductions with fewer parallel executions at the cost of a less tight stepsize function, possibly deleting too many properties to find a condition satisfying recommendation, negatively impacting the recommender's precision.

We consider two **backoff conditions** to trigger the invocation of a backoff strategy:

a) *TooFewRecommendations.* A minimum threshold T_1 for the number of returned properties of the standard recommender.
b) *TooUnlikelyRecommendations.* A minimum threshold T_2 for the average probability of the top 10 recommendations returned by the standard recommender.

4 Evaluation

This section describes the conducted evaluation procedures and their respective results with respect to the performance and quality of the recommender. Furthermore, the effect of the proposed aggregation strategies and metrics will be demonstrated.

The described approach was implemented[5] using Golang for usage with arbitrary RDF datasets and evaluations were conducted on a machine with Intel Core i7 8700k

[5] https://github.com/lgleim/SchemaTreeRecommender.

processor ($6 \times 3, 7$ GHz, Hyper-threading enabled) and 64 GB of RAM. Note, however, that for the SchemaTree approach much less RAM would have been sufficient since the entire in-memory SchemaTree for the Wikidata dataset uses less then 1.3 GB of RAM. Further, the two-pass creation of the SchemaTree for this dataset takes about 20 min in total, whereas the runtime is largely dominated by disk IO and dataset decompression.

Dataset and Preparation. In order to evaluate the different variants of the SchemaTree recommender and compare its performance to the state-of-the-art Wikidata PropertySuggester, we employ the full Dumps of Wikidata as of July 29th, 2019[6]. We split the dataset into training set ($99.9\% = 58810044$ of the subjects in the dataset) and test set ($0.1\% = 58868$ of the subjects in the dataset) by splitting off every 1000th subject off into the test- and all others into the training set. The training set is then used to construct the SchemaTree, while the test set is used to measure performance. For technical reasons, the full Wikidata PropertySuggester association rules were generated from the full dataset, theoretically giving that system an unfair performance advantage, due to test data being part of its training process. However, as we will see later, even this additional advantage does not make it outperform the proposed approach. All recommenders are subsequently evaluated using the same test set.

Evaluation Procedure. To evaluate we use the procedure proposed by Zangerle et al. [25]. For each evaluated entity, we gather its full set of properties, order the properties by descending support in the training set, and split it into two subsets: the input set and the left-out set. Then, we call the recommender on the input set and evaluate how well it performs at recommending the very same properties that were initially left out. We start with an input set that contains all properties and repeatedly remove the least frequent non-type property in the input set, adding it to the left-out set. On each step, we run an evaluation with the current pair of input and left-out sets. This process is repeated as long as any non-type properties exist in the input set.

Recommender systems capable of employing type information will receive the entity types as additional context in their input set, while the other systems are evaluated without this additional information. Each evaluation run requires that both the input and left-out sets are non-empty.

The results are grouped by the amount of non-type properties in the input sets and left-out sets. This aggregation will guarantee that each entity will belong to the same group across evaluation runs with all models, whether the model uses the additional type properties or not. Ensuring that entities always belong to the same grouping, irrespective of the recommender system used, eases the direct comparison of the different model performances.

Metrics. In order to evaluate the quality of the computed recommendations we employ the following metrics, which are respectively computed for each group of entities:

[6] https://dumps.wikimedia.org/wikidatawiki/entities/20190729/.

- `∅Rank`: The average position of the first correct recommendation in the top-most 500 recommendations, respectively incurring 500 if not contained.
- `Stddev`: The standard deviation of the ranks.
- `Prec@L`: The average precision considering only the first L recommendations, i.e., the ratio of relevant properties found regarding only the first L recommendations to L, where L equals the number of left-outs in each individual run of the recommender.
- `TopX`: The percentage of all conducted recommendations, where the first correct result was contained in the top X recommendations, where `TopL` employs X equal to the number of left-outs L in each individual run of the recommender.
- `∅Latency`: The average time until the list of recommendations was received over all recommender calls in milliseconds.
- `Recall`: The average number of properties that could be found in the recommendations list, divided by the total number of left-out properties.
- Modified `F1`: The harmonic mean of `Prec@L` and `Recall`, with an optimal value of 1 (perfect precision and recall) and worst of 0.

Choosing a Backoff Strategy. The large number of possible configuration options, resulting from the different backoff conditions, strategies and parameters introduced in Sect. 3, motivates a preparatory empirical evaluation of different backoff configurations. The control variables include merger and splitter strategies inside the *SplitPropertySet* backoff strategy (c.f. Sect. 3), as well as several options to choose a stepsize function and the number of parallel executions inside the *DeleteLowFrequency* backoff strategy (c.f. Sect. 3). Additionally, it is necessary to set trigger thresholds for the backoff conditions, which can be combined arbitrarily with any backoff strategy above.

To find a good selection of parameters and a suiting combination of condition and backoff strategy, we perform a grid search in which we evaluate 96 different configurations, using the procedure described in Sect. 4 in conjunction with every 10th subject of the test set described in Sect. 4 and metrics computed over all conducted recommendations. We chose different parameters for each condition and backoff strategy by combining the different backoff strategies (depicted in the upper sub-table of Table 1) with the different combinations of the condition configurations (depicted in the lower sub-table of Table 1). We choose $a = 0.4$ as parameter for the linear *Delete Low Frequency Backoff Strategy*.

Table 1. Tested combinations of workflow configurations.

Backoff strategy	**Variable**	**Configuration variants**
Split property set	Splitter	Every second item, two frequency ranges
	Merger	avg, max
Delete low frequency	Stepsize	Linear, proportional
	Parallel runs	{1,..,6}
Backoff condition	**Variable**	**Configuration variants**
TooFewRecommendations	Threshold	{1, 2, 3}
TooUnlikelyRecommendations	Threshold	{0.033, 0.066, 0.1}

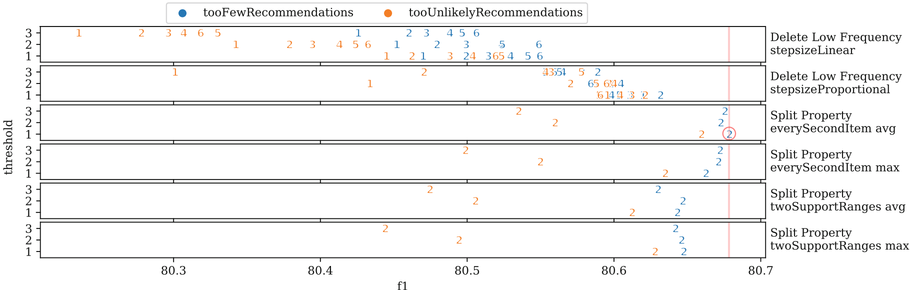


Fig. 2. Comparison of 96 different backoff configurations (c.f. Table 1) w.r.t. their modified `F1` score. Higher is better. The six subplots compare the six principal backoff *strategy* configurations outlined in Sect. 3. Sample color indicates the employed backoff *condition* and the position on the respective y-axis the associated backoff *threshold*. *TooUnlikelyRecommondations* thresholds are scaled by factor three for better visual comparability. The number markers indicated the respective number of parallel recommender runs. The best performing strategy is highlighted in red. The `F1` score for the system without any backoff startegy is 71.52%.

The evaluation results of all 96 configurations w.r.t. their modified `F1` score are illustrated in Fig. 2. First, we observe that any backoff strategy significantly improves the system as without any we obtained an `F1` score of 71.52% while all backoff strategies result in more than 80.2%. Comparing the two backoff conditions *TooFewRecommendations* and *TooUnlikelyRecommendations*, the superior performance of the *TooFewRecommendations* strategy is immediately obvious. Comparing the different backoff strategies, we see that the *DeleteLowFrequency* approach with a linear stepsize function performed clearly worst and only reaches comparably better results at the cost of multiple parallel executions. This is likely a direct result of removing an insufficient amount of properties from the initial property set to observe the desired backoff characteristic. In contrast, the *DeleteLowFrequency* strategy with proportional stepsize function achieves much better results, likely because a more optimal, larger amount of properties is left out of the input set compared to the linear stepsize function. In comparison to the *DeleteLowFrequency* approach, the *SplitPropertySet* backoff strategy generally appears to achieve higher recall, which intuitively makes sense, due to the fact that no properties providing context are deleted from the effective input to the recommender system. The respective average merging strategy appears to performs slightly better in most cases then taking the maximum per item probability across the splits.

Concluding, we choose the *SplitPropertySet* backoff approach in conjunction with *everySecondItem* splitter and average merging strategy, triggered by the *TooFewRecommendations* condition with threshold 1, which maximized the modified `F1` score over all evaluated strategies.

Evaluation Results. In order to compare the different variants of the SchemaTree recommender with the state-of-the-art Wikidata PropertySuggester (PS) system, we evaluated each system using the procedure described in Sect. 4. We first discuss the overall

evaluation results as summarized in Table 2, before examining selected metrics in more detail for different input property set sizes.

All three variants of the SchemaTree recommender (Standard, with type information and with both type information and backoff strategies enabled) clearly outperform the state-of-the-art in terms of `∅Rank` of the first correct recommendation. Additionally the `Stddev` of that rank is significantly lower, leading to more predictable recommendation results. When comparing only systems with or without usage of type information, the SchemaTree recommender consistently achieves higher `Prec@L`, `F1` and `TopX` scores, as well as lower average recommendation `∅Latency`. It thus outperforms the state-of-the-art Wikidata PropertySuggester in every evaluated metric, at the cost of about 1.3 GB of RAM for keeping the SchemaTree data structure in RAM.

With a relative reduction of 71% compared to the PropertySuggester baseline, the average rank of the first correct recommended property for the *Typed & Backoff* approach improves significantly, which directly results in an improved user experience. Note also that the *Typed & Backoff* method leads to relative improvement of 44.83% of the average rank of the first correct property recommendation over the simpler *Typed* approach and a 7.54% relative improvement of correct Top10 results, which in turn means that users will actually see relevant recommendations significantly more often.

Table 2. Benchmark results of the evaluated systems. At the top, we have the PropertySuggester, first without and second with type information for comparison. The three systems at the bottom are the variations of the SchemaTree recommender.

Recommender	∅Rank	Stddev	Prec@L	F1	Top1	Top5	Top10	TopL	∅Latency
PS wo/ Types	156.67	179.04	3.31%	6.26%	3.83%	10.15%	12.64%	10.97%	350.58 ms
Wikidata PS	13.05	70.84	64.57%	76.83%	74.34%	90.11%	93.28%	83.65%	29.18 ms
Standard	8.00	40.43	56.48%	71.87%	67.14%	83.38%	89.76%	77.64%	119.66 ms
Typed	6.73	46.25	67.90%	80.49%	78.97%	93.07%	96.02%	87.16%	**25.01 ms**
Typed & Backoff	**3.78**	**24.38**	**68.00%**	**80.76%**	**79.07%**	**93.30%**	**96.32%**	**87.40%**	25.73 ms

To provide a more detailed breakdown of the performance characteristics, we drill down into the results of the metrics `Top5`, `∅Rank` and modified `F1` score and inspect each measure in relation to their respective input set sizes to the recommender systems. Figure 3(a) illustrates the distribution of the respective input set sizes. Note that all following figures depict results for property set sizes of 2 to 55 non-type input parameters. Whereas the lower limit 2 directly results from the requirement to have non-empty input- and left-out sets for the evaluation, the upper limit 55 is selected because of the limited amount of subjects with corresponding larger set size in the test set and the resulting reduced reliability of the evaluation results.

When comparing the `Top5` results, depicted in Fig. 3(b), the PropertySuggester without provided type information (*PS wo/ Types*) only achieves a low sub-40% `Top5` score throughout the entire test set. In comparison, the *Standard* SchemaTree already results in a significant performance gain, reaching its peak of close to 90% at around 13 input properties, sustaining a score of about 80% as more properties are added to the input set.

The introduction of typing information favours both the PropertySuggester and the SchemaTree, as seen in the results obtained by *Wikidata PS* and *Typed*. As anticipated in Sect. 3, the typing information significantly boosts recommendation performance when only a limited amount of input properties is provided to the recommender. Effectively, the `Top5` score of the *Typed* SchemaTree recommender rises by up to more than 75% absolute compared to its untyped *Standard* counterpart. As more and more properties exist on an entity, type information plays a less important role as properties become more specialized and the existing input properties provide more context information. Notably, the SchemaTree (without type information) outperforms the PropertySuggester (with type information) on recalling left-out properties, especially after the 15 properties mark.

While the effect of introducing backoff strategies can be seen in Fig. 3(b) in the general slight performance improvement of the *Typed & Backoff* recommender w.r.t. to the *Typed* SchemaTree, its effect is more obvious when inspecting the `∅Rank` of the first correct recommendation in Fig. 3(c). While all characteristics of the different systems described w.r.t. the `Top5` score can also be observed for the `∅Rank`, it is clearly visible how the introduction of backoff improves the recommendation especially for larger input set sizes. Better recommendations are especially given for entities that are already rather complete. Due to the backoff, properties that co-occur with a subset of the given input can also be recommended, whereas without it only recommends properties that co-occurred with the complete input in the training data. As such, the backoff mechanism clearly fulfills its intended behaviour, as described in Sect. 3. As explained there, the performance degradation of the Wikidata PS likely stems from error accumulation when combining the confidence scores of the potentially many applicable association rules, compared to the frequentist inference approach of the SchemaTree recommender.

Finally, we examine the modified `F1` score (Fig. 3(d)) as a measure of the overall quality of the recommendations with varying degrees of left-out properties. Highlighting only the SchemaTree variants, we see a clear confirmation of our previous findings,

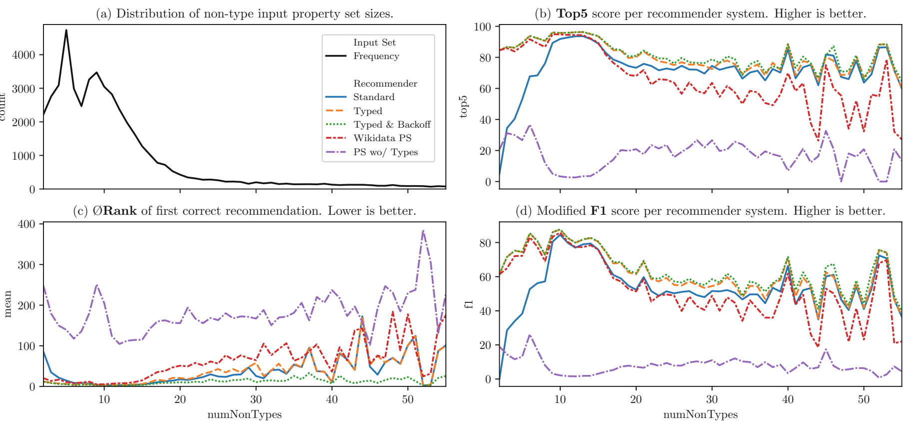


Fig. 3. Detailed results of the recommender system evaluation for different non-type input set sizes.

that type information improves the recommendation quality especially for low numbers of input properties. The incorporation of backoff strategies, on the other hand, only seems to have a slight positive impact with regards to this metric.

Overall, the proposed approach tends to recommend properties that are more contextually relevant (since it can take more context information into account). The Wikidata PropertySuggester, however, can only recommend contextually relevant properties as long as there are meaningful association rules.

5 Conclusion and Future Work

In this work, we introduced a trie-based data structure, capable of efficiently learning and representing property set coocurrence frequencies in RDF graphs. We refer to this data structure as the SchemaTree. We have shown how to use it to efficiently compute the support count of arbitrary property sets in the encoded graph and how it can be employed for maximum-likelihood property recommendation to assist in the manual creation of knowledge graphs, analogously to the Wikidata PropertySuggester. We showed how to improve recall and precision of the recommender system for entities with sparse property sets by incorporating type information from classifying properties into the recommender system. We then presented different novel backoff strategies to improve the capability of the recommender to generalize to unseen property combinations, further improving upon the state-of-the-art, and evaluated the approaches on the Wikidata dataset. We evaluated the performance of different backoff configurations and compared the resulting variations of the SchemaTree property recommender to the state-of-the-art Wikidata PropertySuggester system, demonstrating that our system clearly outperforms the state-of-the-art in all evaluated metrics. Finally, we provided qualitative reasoning as to the limitations of the popular association-rule based recommender systems and how our system overcomes them, as well as advantages and drawbacks of the approach.

One current limitation of this and other existing works is that qualifiers are not taken into account, nor predicted, providing additional directions for further investigations. Further, while we have shown that the presented backoff strategies already significantly improve the performance of the presented recommender, we want to investigate further backoff strategies in future work. Additional theoretical understanding of the current backoff approaches will likely lead to further improvements. To gain this understanding, one would also want to have experimental evidence on how the recommender works for rare properties in heterogeneous graphs.

Further aspects for future work include the inclusion of the values of properties into the property recommendations; one can assume that these also have additional information that can indicate relevance (e.g., typically only people born after 1900 have a personal homepage). Besides, one could also investigate the prediction of values for the properties.

However, due to the combinatorial explosion of options, these are not feasible with the current approach alone (when employing the same approach currently used for classifying properties). For value prediction, if only a small amount of values are possible for a given property, one could attempt to adapt the SchemaTree approach separately for

each specific property. For more involved cases, recommending values could be done in a second stage with a different algorithm. Note that value recommendation would also need to work for effectively infinite and/or continuous domains (e.g., floating point numbers), while the current approach only chooses from a finite set of discrete options.

Besides improving the quality of the recommendations themselves, we see also a need fur improving how they are presented to the user. For example, some recommended properties are closely related to each other and presenting them in some sort of clustered or hierarchical form might lead to a better user experience. Further, the conducted evaluation is an attempt to mimic the manual entity authoring process (analogously to evaluations in previous work), we envision a future user study to validate our findings in practice.

References

1. Abedjan, Z., Naumann, F.: Improving RDF data through association rule mining. Datenbank-Spektrum **13**(2), 111–120 (2013). https://doi.org/10.1007/s13222-013-0126-x
2. Abedjan, Z., Naumann, F.: Amending RDF entities with new facts. In: Presutti, V., Blomqvist, E., Troncy, R., Sack, H., Papadakis, I., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8798, pp. 131–143. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11955-7_11
3. Aggarwal, C.C., Philip, S.Y.: A new framework for itemset generation. In: Proceedings of the 17th Symposium on Principles of Database Systems, pp. 18–24 (1998)
4. Agrawal, R., Srikant, R., et al.: Fast algorithms for mining association rules. In: Proceedings of the 20th International Conference on Very Large Data Bases, VLDB, vol. 1215, pp. 487–499 (1994)
5. Alonso, O., Kumar, A.: System and method for search and recommendation based on usage mining, US Patent 7,092,936, 15 August 2006
6. Balaraman, V., Razniewski, S., Nutt, W.: Recoin: relative completeness in Wikidata. In: Companion Proceedings of the Web Conference, pp. 1787–1792 (2018)
7. Bauer, F., Kaltenböck, M.: Linked open data: the essentials. In: A Quick Start Guide for Decision Makers, January 2012
8. Berners-Lee, T., Hendler, J., Lassila, O., et al.: The semantic web. Sci. Am. **284**(5), 28–37 (2001)
9. Buneman, P.: Semistructured data. In: Proceedings of the 16th Symposium on Principles of Database Systems, pp. 117–121. ACM (1997)
10. Cafarella, M.J., Halevy, A., Wang, D.Z., Wu, E., Zhang, Y.: WebTables: exploring the power of tables on the web. Proc. VLDB Endowment **1**(1), 538–549 (2008)
11. Dessi, A., Atzori, M.: A machine-learning approach to ranking RDF properties. Future Gener. Comput. Syst. **54**, 366–377 (2016)
12. Fernández, J.D., Beek, W., Martínez-Prieto, M.A., Arias, M.: LOD-a-lot. In: d'Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10588, pp. 75–83. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_7
13. Gassler, W., Zangerle, E., Specht, G.: Guided curation of semistructured data in collaboratively-built knowledge bases. Future Gener Comput. Syst. **31**, 111–119 (2014)
14. Gleim, L.C., et al.: Schema extraction for privacy preserving processing of sensitive data. In: Joint Proceedings of the MEPDaW, SeWeBMeDA and SWeTI 2018, pp. 36–47. CEUR WS Proceedings, vol. 2112 (2018)
15. Guha, R.V., Brickley, D., Macbeth, S.: Schema.org: evolution of structured data on the web. Commun. ACM **59**(2), 44–51 (2016)

16. Gyorodi, C., Gyorodi, R., Cofeey, T., Holban, S.: Mining association rules using Dynamic FP-trees. In: Proceedings of the Irish Signals and Systems Conference, pp. 76–81 (2003)
17. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. ACM SIGMOD Rec. **29**, 1–12 (2000)
18. Lee, T., Wang, Z., Wang, H., Hwang, S.W.: Attribute extraction and scoring: a probabilistic approach. In: 29th International Conference on Data Engineering (ICDE), pp. 194–205. IEEE (2013)
19. Ngomo, N., Hoffmann, M., Usbeck, R., Jha, K., et al.: Holistic and scalable ranking of RDF data. In: International Conference on Big Data, pp. 746–755. IEEE (2017)
20. Razniewski, S., Balaraman, V., Nutt, W.: Doctoral advisor or medical condition: towards entity-specific rankings of knowledge base properties. In: Cong, G., Peng, W.-C., Zhang, W.E., Li, C., Sun, A. (eds.) ADMA 2017. LNCS (LNAI), vol. 10604, pp. 526–540. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69179-4_37
21. Said, A.M., Dominic, P., Abdullah, A.B.: A comparative study of FP-growth variations. Int. J. Comput. Sci. Netw. Secur. **9**(5), 266–272 (2009)
22. Sigurbjörnsson, B., Van Zwol, R.: Flickr tag recommendation based on collective knowledge. In: Proceedings of the 17th International Conference on World Wide Web, pp. 327–336. ACM (2008)
23. Suen, C.Y., Shinghal, R.: Operational Expert System Applications in Canada. Elsevier, Amsterdam (2014)
24. Völker, J., Niepert, M.: Statistical schema induction. In: Antoniou, G., et al. (eds.) ESWC 2011. LNCS, vol. 6643, pp. 124–138. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21034-1_9
25. Zangerle, E., Gassler, W., Pichl, M., Steinhauser, S., Specht, G.: An empirical evaluation of property recommender systems for Wikidata and collaborative knowledge bases. In: Proceedings of the 12th International Symposium on Open Collaboration, p. 18. ACM (2016)
26. Zangerle, E., Gassler, W., Specht, G.: Recommending structure in collaborative semistructured information systems. In: Proceedings of the 4th Conference on Recommender Systems, pp. 261–264. ACM (2010)