SchemaTree: Maximum-likelihood Property Recommendation for Wikidata and Collaborative Knowledge Bases

Lars Gleim
RWTH University
Aachen, Germany

Dominik Hüser
RWTH University
Aachen, Germany
dominik.hueser@rwth-aachen.de

Rafael Schimassek
RWTH University
Aachen, Germany
rafael.schimassek@rwth-aachen.de

Maximilian Peters
RWTH University
Aachen, Germany
maximilian.peters1@rwth-aachen.de

Christoph Krämer
RWTH University
Aachen, Germany
christoph.kraemer@rwth-aachen.de

ABSTRACT

Wikidata is a free and open knowledge base which can be read and edited by both humans and bots. 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 an alternative approach to provide such recommendations. We introduce a trie-based method that can efficiently learn and represent property set probabilities in RDF graphs. We furthermore improve the method by adding type information to improve recommendation precision. Additionally we introduce backoff strategies which later increase the recall of the initial approach for large entities. Then, we investigate how the captured structure can be employed for property recommendation, analogously to the Wikidata property recommender. We evaluate our approach on the full Wikidata dataset.

CCS CONCEPTS

• **Information systems → Similarity measures**.

KEYWORDS

Wikidata, Recommender Systems, Statistical Property Recommendation, Frequent Pattern Mining, Knowledge Graph Editing

ACM Reference Format:
Lars Gleim, Dominik Hüser, Rafael Schimassek, Maximilian Peters, and Christoph Krämer. 2019. SchemaTree: Maximum-likelihood Property Recommendation for Wikidata and Collaborative Knowledge Bases. In *KGlab 2019*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/1122445.1122456

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 bots. However, manual editing of knowledge-bases is traditionally an error prone process [17] and requires intimate knowledge of the underlying information model.

In the context of the Semantic Web [6] and the Linked Open Data Cloud [7], knowledge bases typically fall into the category of semi-structured data. In contrast to classical structured data – often associated with relational databases – where the data follows a fixed and regular structure or data model, semi-structured data do not conform to a fixed structure, but provide self-describing structural primitives. If separate schemata exist, they only place loose constraints on the data. As such, entities of semantically equal type regularly feature different attribute sets (also called the properties or predicates of the entity in the context of RDF), different attribute orderings, etc. [8]

For Wikidata, much care is taken to create useful properties, which have support from the community[1]. 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 [20]. Similar recommendation approaches are also employed in more general RDF recommender systems and collaborative information systems [1, 2, 10, 16, 21]. The main contributions of this work are the use of frequentist inference for recommendation of properties.

Furthermore we detect cases in which the baseline SchemaTree does not perform optimally, so that we immediately present improvements in this work too. The results in section 5 show that the recommender performs well on the Wikidata dataset, with quality lacks in entites with low number of properties and already quite a lot properties. We see that the introduced backoff strategies as well as the type information compensate those quality lacks well.

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

Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

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

property recommendation. Afterwards, we present a the type extension of the baseline SchemaTree in section 3 present strategies to deal with border cases in section 4 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 large scale RDF datasets, before summarizing our results.

2 RELATED WORK

Several data-driven attribute 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. [9] propose the attribute correlation statistics database (AcsDB), which enables attribute suggestion based on their cooccurences in web tables, to help database designers to choose schema elements. Other examples include [5, 14].

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. Fundamentally motivated by human abstract association capabilities, association rule-based recommendation is based on the rationale that 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.

A first example of such work is by Abedjan et al. [1, 2], whose RDF enrichment approach employs association rule mining for predicate 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[2], which is called *PropertySuggester*. The difference with the basic approach is the introduction of so-called *classifying* properties, which are the properties `instanceOf` and `subclassOf` [20]. 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 [10, 21] is another property recommendation system based on association rules, which distinguishes itself from previous systems by ranking recommendations based on the number of occurrences of a given rule across all training data items (in contrast to the sum of confidences the previous approaches). Zangerle et al. [20] proposed an extension of the Snoopy approach, inspired by previous work of Sigurbjörnsson et al. [16], ranking 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. [20] further conducted an empirical evaluation of several state-of-the-art property recommender systems for Wikidata and collaborative knowledge bases, concluding the Wikidata recommender approach to significantly outperform all evaluated competing systems.

Other interesting applications of association rule learning in the context of knowledge bases include schema induction [19], aiming to derive ontological schema directly from RDF data, and the mining of frequently occurring subgraph patterns within data graphs [18].

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 attribute 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 whether a frequentist approximation of this true likelihood would suffice to produce good recommendations. 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 for efficient probability computations.

2.1 Preliminaries

We are creating a recommendation system for a Knowledge Base (KB), which consists of entities with properties (which are also often called attributes or predicates). An entity can have the same property multiple times and entities in the KB can also have type information.[3] Recommending properties is proposing a relevant property for a given entity, which it did not have before. In this work, we limit ourselves to proposing properties with respect to their maximum-likelihood. Hence, in the scope of this paper we define the task as follows:

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

$$\begin{aligned}\hat{a} &= \underset{a \in (\mathcal{A} \setminus S)}{\operatorname{argmax}} P(a | \{s_1, \ldots, s_n\}) \\ &= \underset{a \in (\mathcal{A} \setminus S)}{\operatorname{argmax}} \frac{P(\{a, s_1, \ldots, s_n\})}{P(\{s_1, \ldots, s_n\})} \\ &= \underset{a \in (\mathcal{A} \setminus S)}{\operatorname{argmax}} P(\{a, s_1, \ldots, s_n\})\end{aligned}$$

where $P(\{t_1, \ldots, t_m\})$ is the probability of finding an entity with at least the properties $t_1, \ldots, t_m$.

Intuitively, we need to find the property which is most often observed together with the properties which the entity already has. This directly corresponds to a maximum-likelihood estimation over the true probability distribution P of attribute co-occurrences.

To obtain k recommendations, this definition gets extended such that we obtain a list of the k attributes which have the highest k maximum-likelihood probabilities, sorted by that probability.

Given a sufficiently large amount of training data, the true joint probabilities can be reasonably well approximated by their relative

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

[3]These rather minimal requirements are fulfilled by Wikidata and RDF data.

frequency of occurrence, using a frequentist probability interpretation. We borrow the common approach of grouping RDF triples by subject (i.e. entity) to derive the multiset $\mathfrak{P}$ of all per subject predicate sets [11, 19, 20], formally $\mathfrak{P} = \{A | E \in KG, A \text{ is the set of attributes of } E\}$. Then, we can determine the absolute frequency (or support) $supp(A)$ of a set of attributes $A = \{a_1, \ldots, a_{|A|}\} \subseteq \mathcal{A}$ (i.e. a pattern) as the number of subject property sets that include it:

$$supp(A) = supp(a_1, \ldots, a_{|A|}) = \sum_{\{Q \in \mathfrak{P} | A \subseteq Q\}} 1$$

Subsequently we can determine the most likely property recommendation via frequentist inference as:

$$\begin{aligned} \hat{a} &\simeq \operatorname*{argmax}_{a \in (\mathcal{A} \setminus S)} \frac{supp(a, s_1, \ldots, s_n)}{supp(s_1, \ldots, s_n)} \\ &= \operatorname*{argmax}_{a \in (\mathcal{A} \setminus S)} supp(a, s_1, \ldots, s_n) \end{aligned}$$

If we would naively compute recommendations according to this definition, it is 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 the number of possible property combinations, it is infeasible to precompute them all. Hence, we introduce a suitable data structure which makes it possible to compute them in a short time in the next subsection.

2.2 Construction

In this subsection, we detail the design and construction of a data structure used for efficient pattern support lookup. To allow for efficient learning, storage, and retrieval of these patterns, we adapt the trie construction idea of the Frequent-Pattern tree (FP-tree), first introduced by Han et al. [13], in order to serve as a highly condensed representation of the property sets. 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 [15]), we build upon the original 2-pass tree construction to enable a more transparent analysis of the tree's properties and because we noticed that the simpler approach was sufficient for dealing with even the LOD-a-lot dataset as an input. However, 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 [12]. As the three is representing a higher level abstraction of the properties used in the knowledge base, we call this tree the *SchemaTree*. Building the tree is done as follows:

(1) For each attribute $a \in \mathcal{A}$, determine its support $supp(a)$ 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 attributes $p_1, \ldots, p_{|\mathcal{A}|}$, first by descending support and second by lexicographical ordering $lex(p_i)$, i.e., such that:

$$\forall i : supp(p_i) \geq supp(p_{i+1}) \wedge \big(supp(p_i) = supp(p_{i+1}) \rightarrow lex(p_i) > lex(p_{i+1})\big)$$

(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. Each node in the tree maintains a counter of its prefix-conditional support and a backlink to its parent. Whenever a new child node is created, it is additionally inserted into the list of occurrences of the corresponding property.

Table 1: A set of entities and respectively associated property sets.

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	$\{p_1, p_2\}$

Figure 1 illustrates the SchemaTree derived from the example knowledge base of five subjects with their property sets depicted in table 1. Patterns are inserted starting from the root node at the top. The blue, continuous 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.

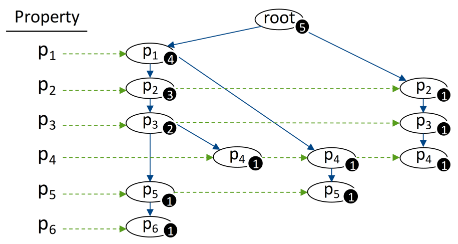


Figure 1: The SchemaTree derived from the entities and their property sets in table 1.

Once this tree is constructed, it can be used to recommend properties.

2.3 Maximum-likelihood Property Recommendation

The property recommendations for a given non-empty property set $A \subseteq \mathcal{A}$ can now 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 (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), and
   (b) if yes, increment the support counter of all property candidates contained in the prefix but not already in A by s', the support counter of A by s' 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, i.e. if the currently inspected prefix node has a lower sort order then the next expected node according to the sorted attribute set A, the expected attribute 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 higher selectivity (i.e. specificity or true negative rate) then starting the search from the most likely attributes 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 $A = p_2, p_3$ (with p_i as in table 1), based on 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).

It may be worth noting, that we can further deduct that all other properties are unlikely to co-occur with the given set of attributes. 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.

3 ADD TYPES TO THE SCHEMATREE

A weakness when recommending properties with our approach are cases in which we want to recommend properties to an entity that has only a few already existing properties. In these cases, the schema tree has too many paths that are matching the already existing properties. To improve the recommender's precision in such cases, we are integrating type information into the schema tree. Therefore we are using the concept of classifying properties ("types") that is already used by the PropertySuggester [20]. The PropertySuggester uses types in an association rule-based approach. However, we adopt this concept into our tree-based approach.

With type information, entities can be classified as belonging to a specific group of entities (the "type"). Wikidata uses the classifying properties "instanceOf" and "subclassOf". The "instanceOf" property identifies an entity as a concrete instance of a class. The "subclassOf" property is used to define hierarchical class structures. For example, "cat-subclassOf-pet" means that every instance of "cat" is also an instance of "pet". Purely property based recommender systems would not recognize the type. However, It is not as meaningful to know that an item is an instance of some type as it is to know of which type the item is an instance.

We collect the property and type data by reading wikidata in the RDF format. The identifier of the predicate represents a property. In contrast, the identifier of the object related to a classifying property represents a type. For example, the statement "Paris-instanceOf-capital" gives us the property "instanceOf" and as "instanceOf" is a classifying property we further get the type "capital".

To build the schema tree, we treat the types equally to the properties. In the first scan, we count the frequencies of properties as well as types. Then we create a strict totally ordered set, including properties and types.

We order by descending support because we want the nodes related to frequent items to be on the top of the tree structure. As second we order by the lexicographical order of the item names, to ensure that we have a fixed order between items with the same support. As wikidata items have a unique name, the lexicographical order is sufficient for this purpose.

We define a subject as the ordered set of all properties and types corresponding to one entity. During the second pass, we insert all subjects into the tree structure.

In the recommendation algorithm, we search for paths in the tree that contain all properties and types of the requested entity. At this point, the types improve the accuracy of our algorithm. By including types in the tree, the algorithm considers only paths that contain the corresponding types. Accordingly, paths of entities of other types, but with similar properties, are excluded.

When creating the candidate set, we have to distinguish between types and properties. In fact, we only consider properties and no types as possible candidates, since we only want to recommend properties. If you only consider types as candidates, it is also possible to recommend suitable types to a set of properties by using the same schema tree.

4 BACKOFF STRATEGIES

In the tests we see that the standard schematree already performs good on the wikidata set. Nevertheless, we encounter border cases where the schematree recommender performs weak, i.e. those cases where we request a recommendation for an entity which already got many properties assigned (fig. 7). In those cases it is possible that the recommender can not perform any recommendation at all.

Let us have a further look at this phenomenon: Assume that the recommender runs on the schematree in fig. 1 with the incoming property set $\{p_1, p_2, p_3, p_4\}$. p_4 is the property with the lowest support and therefore the starting point for the recommender. Only the left-most 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.

The concept we present is to handle exactly those border cases via *Backoff Strategies*, which is only enabled if a predefined *Condition* is met. The idea of all backoff strategies is the same: In case that the input set of properties requires an extra treatment (meaning the the condition is not satisfied), we just reduce the set of input properties so that we enable recommendations by the standard recommender again, using only a subset of the actual properties. For instance in the described case above the recommender could actually make a recommendation if we would leave out p_4 in the request. That would result in recommendation p_5, p_6 with their respective probabilities.

In the following we will introduce conditions defining *when* a backoff is needed, as well as two backoff strategies which define *how* we handle those cases:

4.1 Condition

In the following we present two different conditions which enable backoff strategies:

```
TooFewRecommendations(propertySet, threshold){
  rec := StandardRecommender(propertySet);
  if len(rec) ≤ threshold {
    rec = backoff(propertySet);
  }
  return rec;
}
```

Listing 1: Condition *TooFewRecommendations*

In listing 1 we see the first condition which fires the backoff strategy if the number of returned properties of the standard recommender is below a threshold. If that is not the case the recommendation of the standard recommender is returned. E.g. if we choose the threshold to be one, then the backoff strategy is fired iff the recommender returned an empty property recommendation.

```
TooUnlikeRecommendation(propertySet, thresh){
  rec := StandardRecommender(propertySet);
  if rec.top10AvgProbability() ≤ threshold {
    rec = backoff(propertySet);
  }
  return rec;
}
```

Listing 2: Condition *TooUnlikelyRecommendation*

As comparison we introduce a second condition which computes the average probability of the 10 recommendations returned by the standard recommender. Is that value below a threshold, the backoff strategy is fired. With that condition we enable firing a backoff strategy in case of a recommendation containing only unlikely properties. Mind the special case that less than 10 properties are recommended by the standard recommender: Not made recommendations are treat like property recommendations with probability equal to zero.

4.2 Backoff: Split Property Set

The *Split Property Set Backoff* reduces the incoming property set in the following way:

(1) Sort incoming properties according to their frequency (which we get from the tree), resulting in list $P = (p_1, ..., p_n)$ with $supp(p_i) > supp(p_j), i < j$ for $i, j \in [n], n \in \mathbb{N}$
(2) Split the property set into two subsets P'_1 and P'_2.
(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 property subset resulting in cleaned recommendations R_1 and R_2
(5) Merge recommendation R_1 and R_2 to R, which is finally returned as result of the backoff strategy.

We present two ways to perform splitting:

- **Every Second Item**: $P'_1 = \{p_i \mid i\%2 = 0, i \in [n]\}$ and $P'_2 = \{p_i \mid i\%2 = 1, i \in [n]\}$, i.e. the set P'_1 contains all the even indexes and P'_2 all the odd, so that we result in two subsets with more or less equal distribution of high and low frequency properties.
- **Two Frequency Ranges**: $P'_1 = \{p_i \mid i \leq \frac{n}{2}\}$ and $P'_2 = \{p_i \mid i > \frac{n}{2}\}$, i.e. the set P'_1 contains the 50% most frequent recommendations and P'_2 the 50% less frequent recommendations.

In the end, the two resulting recommendations must be merged. Note, that the two recommendations' intersection might not be empty, meaning that there can be properties recommended based on either P_1 or P_2. Therefore merging is not just concatenating R_1 and R_2. To merge the two resulting (sub)recommendations we apply simple aggregation. Either the maximum probability of both recommendations or the average of the both recommendation is taken. Note that in case that a property is only recommended on one subset, its probabilty halves in case of average merging.

Let us have a look at an example (fig. 2): Imagine the incoming entity got properties ($P21, P495, P5, P193$), ordered according to the support of the property in descending order. Then the *Every Second Item Splitter* is used to feed two standard recommenders with distinct subsets of the incoming property set. Those two recommenders, executed in parallel, return two independent recommendations: $P201$ and $P14$ for the first subset as well as $P14$ and $P403$ for the second one. Finally the two sets are merged by averaging the recommended properties' probabilities.

4.3 Backoff: Delete Low Frequency

As in the previous backoff strategy, the *Delete Low Frequency* approach also executes the recommender on several subsets of the original property set in parallel. This time the approach does not split the set of properties into two equally sized property subsets but deletes more an more low frequency properties until a proper

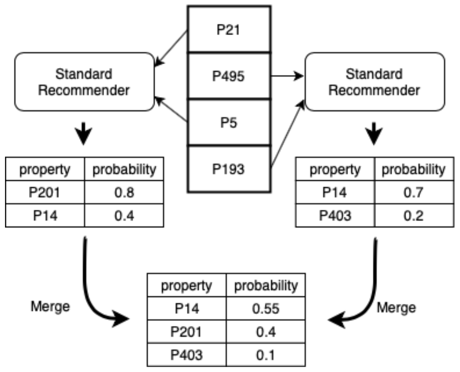


Figure 2: Schematic view of a split property set backoff execution with *Every Second Item* Splitter and average merge.

recommendation is possible. For $q \in \mathbb{N}$ parallel recommenders, the execution would look as follows:

(1) Sort incoming properties according to their frequency (which we get from the tree). I.e. we have a list $P = (p_1, ..., p_n)$ with $supp(p_i) > supp(p_j), i < j$ with $i, j \in [n], n \in \mathbb{N}$.
(2) Create q subsets $P'_i, i \in [q]$ by deleting the $d(i)$ least frequent items from the original property set P. The function$d : \mathbb{N} \to \mathbb{N}$ determines the number of low frequent properties deleted form P in run i.
(3) Run the recommender on the subset $P'_i, i \in [q]$ in parallel, obtaining recommendations $R_i, i \in [q]$.
(4) Choose that recommendation R_i which satisfies the condition and deleted the least number of properties.

There are several possible ways to define the number of least frequent properties $d(i)$, which are deleted from P in run $i \in [p]$

- **Linear Stepsize**: $d_L(i) = i$ i.e. set P_i does not contain the least i properties. Meaning that with every further parallel execution we remove one more item from the property set.
- **Proportional Stepsize**: $d_P(i) = 0.4 * n * \frac{i}{q}$. I.e. we remove up to 40% of the properties in p equally large steps.

The disadvantage of the first approach is the need for many parallel recommendations when a lot of properties have to be erased to make a proper recommendation. The second approach manages to cover a wider range of property deletions while running a lower number of executions of the recommender is needed. This comes with the tradeoff of a less tight stepsize function, possibly deleting too many properties to find a condition satisfying recommendation. Note that this has negative effect on the recommender's precision.

In any case, the previously deleted items need to be erased from the recommendation if the recommender actually recommended these back to the property set.

Let us have a look at the example in fig. 3: We spot the incoming entity's property set in upper part of the figure. The properties are sorted by their support, from high (left) to low (right). The strategy runs three recommenders in parallel. The first recommender (most right) gets the whole property set except the least frequent property $P93$. The second recommender performs its recommendation on the whole set without the two least frequent properties $P90$ and $P93$. Accordingly, the third one only regards the whole property set excluding $P389, P90, P93$. Assume the recommendation of the first recommender would still not satisfy the condition which triggered the backoff strategy in first place, but the second and third do. The the recommendation of the second recommender is returned, since it additionally minimizes the number of adaptions to the incoming property set to perform a satisfying recommendation.

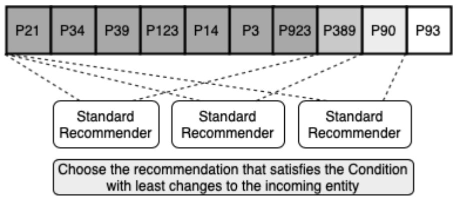

P21
P34
P39
P123
P14
P3
P923
P389
P90
P93
Standard Recommender
Standard Recommender
Standard Recommender
Choose the recommendation that satisfies the Condition with least changes to the incoming entity

Figure 3: Schematic view of a Delete Low Frequency Execution with *lineaer stepsize* function.

5 EVALUATION

This section describes the procedures and handlers and that are used to evaluate the performance and quality of the recommender. Furthermore, the result aggregation strategies and metrics will be presented. The evaluations are executed on two machines, an Intel Xenon Gold 5122 with 16 x 3,6GHz and 131GB RAM, and an Intel Core i7 8700k with 12 x 3,7GHz and 65 GB RAM.

5.1 Procedure

To assess the quality of the recommender, different tests were executed. A test is defined by the evaluation handler used, and the test set. For each entity, we gather the full set properties and split that set into a pair of subsets: the input set and the left-out set. Then we call the recommender on the input set and evaluate how well it recommended the properties that were left out. The split into input and left-out set is determined by the evaluation handler, which can generate multiple splits for each entity. Each pair of sets is defined as a test case and the following results are stored:

- `SetSize`: The number of properties in the input set.
- `NumTypes`: The number of type properties in the entity.
- `NumLeftOut`: The number of properties in the left-out set, L for short.
- `Rank`: For a single left-out property, the rank of the entity equals the position of the left-out in the sorted recommendations list. For multiple left-outs, the rank equals the number of false positive recommendations in the recommendation list, until the last left-out is found. In both cases, if a left-out is missing, rank 10000 is assigned. A rank of 1 denotes a perfect recommendation, with all the left-outs taking the first spots in the recommendations list.
- `Duration`: The duration of the recommender call for this entity in seconds.
- `NumTP`: The number of true positive recommendations, i.e. the properties contained in the recommendations list and the left-out set.
- `NumTPAtL`: The number of true positive recommendations in the first L recommendations.

- `NumFP`: The number of false positive recommendations, i.e. the properties contained only in the recommendations list and not in the left-out set.
- `NumTN`: The number of true negative recommendations, i.e. the properties neither contained in the recommendations list nor in the left-out set.
- `NumTP`: The number of false negative recommendations, i.e. the properties contained in the left-out set, but not in the recommendations list.

This procedure ignores the ordering of the left-out set and focuses on the position in the recommendations list. Additionally, it provides comparable metrics for different left-out sizes.

5.2 Handler

The handler type determines the distribution into left-out and input property set. A property, that is removed, is moved to the left-out set, and not contained in the input set. Each handler can be configured to work with type information or ignore it. The following handler types were applied:

- `TakeOneButType`: The TakeOneButType evaluation handler removes a single property of an entity, but never type information. This is repeated for all properties of an entity and all entities in the test set. This way, every property of an entity is left out exactly once. This handler performs the same, whether there is type information or not.
- `TakeAllButBest`: This evaluation handler removes all properties of an entity, that contain no type information. For the untyped case, the handler removes all but an identical number of the most frequent properties as there are type properties. Entities, that possess no type property, are ignored. This is done for each entity in the test set. Note, that it was not actually used in the following evaluation but is implemented in the evaluation framework.
- `TakeMoreButCommon`: This handler starts with a reduced set that contains all type properties and the most frequent non-type property. Then it adds non-type properties to the reduced set, one-by-one, until no more properties are left out. The properties are added according to their frequency. For the untyped case, the initial set contains only the most frequent property. This is done for each entity in the test set.

5.3 Grouping

Each test run returns as many results, as combinations that the handlers will generate. Some handlers may generate a single combination per entity while others generate multiple combinations. To be able to extract meaningful statistics from this data, different grouping strategies were applied. For each strategy, the first group aggregates all results, to calculate an overall average. The following groups contain all entities that share the same value of interest according to the grouping strategy. These three different strategies were used:

- `SetSize` groups the results according to the size of the input set - this shows us how well the recommender performs for a given amount of input information;
- `NumLeftOut` groups the results by the size of the left-out set - this helps us to visualize how well the recommender can reconstruct properties that have been left out;
- `NumNonTypes` groups the results by the amount of non-type properties in both the input and the left-out sets - we apply this grouping on evaluations when we test typed and untyped SchemaTrees and want to guarantee that the same entities stay in the same groupings.

5.4 Metrics

Since the results that are calculated for each entity are not very meaningful on its own, this section introduces the metrics which are used to present the evaluation. The metrics aggregate the results for all entities in one group and are calculated for each group. The following metrics are used:

- `RankAvg`: RankAvg is the average over all entity ranks, including 10000 for the properties not included in the recommendations list.
- `RankIfHitAvg`: RankIfHitAvg is the average rank of all entities, whose left-out set was completely contained in the recommendations list.
- `Duration`: The average duration of all recommender calls for entities in this group.
- `Recall`: The ratio of properties, that could be found in the recommendations list, to the total number of left-out properties. This is calculated with the data of all entities of a group.
- `Precision`: The number of relevant properties in the recommendations list by the size of the recommendations list. This is again calculated with the data of all entities of a group.
- `PrecisionAtL`: The number of relevant properties found regarding only the first L ranks divided by L. This is again calculated with the data of all entities of a group.
- `TopX`: The ratio of ranks better or equal than X by the total number of recommendation attempts.
- `Median`: The median of all ranks in a group.
- `StdDev`: The standard deviation of the rank average for all ranks in a group.
- `SubjectCount`: The number of cases that got grouped together by the given grouping strategy.

5.5 Test and Training Set

The training set is used to construct the schematree in first place, while the test set is then used to calculate previously introduced measures. We downloaded the Wikidata RDF Dump[4] and split it into two sets representing training and testing set. The training set contains 99.9% of the dump while the test set contain 0.1%. Let us have a look at the results in the following:

5.6 Results - Best Backoff Strategy

In section 4, it was visible that there are many possibilities to choose parameters and combine backoff strategies with conditions. There are different merger and splitter strategies inside the *Split Property Backoff Strategy* as well as several options to choose a stepsize

[4]https://www.wikidata.org/wiki/Wikidata:Database_download

function and the number of parallel execution inside the *Delete Low Frequency Backoff Strategy*. Additionally, it is necessary to to define thresholds inside the conditions, which finally 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 generated 95 different configuration files. The configuration files are run on the presented split Wikidata dataset, once with the `TakeOneButType` handler grouped by to define Recall and once with the `TakeMoreButCommon` handler to define the PrecisionAtL. In both cases we group by `NumNonTypes`, and calculate the average over all groups. We choose different parameters for each condition and backoff strategy by combining the different backoff strategies in the upper sub-table with the different combinations of the condition configurations in the lower sub-table of table 2.

Backoff	**Variations**	
Split Property Set	Splitter	Every Second Item, Two Frequency Ranges
	Merger	avg, max
Delete Low Frequency	Stepsize	linear, proportional
	Parallel runs	{1,..,6}

Condition	**Variations**	
Too Few Recommendations	Threshold	{1,...,3}
Too Unlikely Recommendations	Threshold	{0.033, 0.066, 0.1}

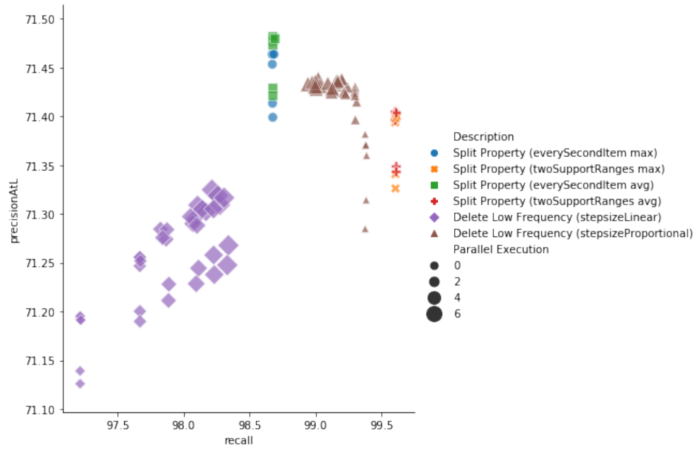
Table 2: Tested combinations of workflow configurations.

The results in terms of their `TakeOneButType` recall and their `TakeMoreButCommon` precisionAtL can be found in figs. 4a and 4b. In both figures we can spot the 95 configurations and their average precision and recall scores. In fig. 4a the backoff strategy of each configuration is highlighted, while in fig. 4b the condition is visualized. Additionally, we spot the number of parallel executions as the size of the markers, since we will prefer less parallel executions for the sake of scalability.

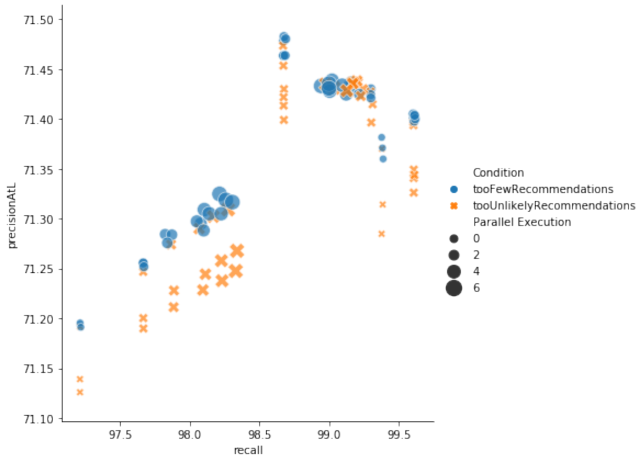
We see that the DeleteLowFrequency approach with a linear stepsize function performed worst in terms of both regarded measures. The reason for this is the too low number of parallel executions which results in deleting not enough properties from the initial property set, to compute a proper recommendation. Therefore we also see a clear trend that increasing number of parallel executions improve both measures slightly. Better performed the Split Property Backoff strategy with EverySecondItem splitter. Here, we see that the average merger performs slightly better than the max merger in terms of precision. Slightly less precise but performing better in terms of recall is the Delete Low Frequency with Proportional Stepsize function. The best results in terms of recall, while decreasing precision only slightly, are generated by the Split Property Backoff, with two support ranges split and average merge. Note that especially the Split Property Backoff with two Support Ranges and the Delete Low Frequency with proportional stepsize perform comparably good. Future work will include further investigation in increasing the upper bound of the proportional stepsize function from 40% to a higher value since it seems to be more promising to split the set at least into half, when executing the Split Property Backoff.

When we compare the different conditions, then we see the trend that tooFewRecommendation performs better than the tooUnlikelyRecommendation condition.

Concluding we choose the Split Property Backoff strategy with Two Support Ranges Splitter and average merger as backoff, which is triggered by the tooFewRecommendation condition. Since the threshold parameter for the condition doesn't make a significant difference we set it to one.



(a) Backoff Strategies



(b) Conditions

Figure 4: Comparison of 95 different backoff configurations, w.r.t. their average recall at TakeOneButType and average precisionAtL at TakeMoreButCommon.

5.7 Results - Benchmark Test

Let us compare the standard recommender to the typed as well as to the typed and backoffed recommender:

First we will take a look at figure 5 for the results we obtained with the *TakeOneButType* handler and *NumNonTypes* grouping. This test will evaluate how well the Recommender can recommend a property that we just took out from from an entity, mimicking the introduction of the last property of every entity, independent of the property order.

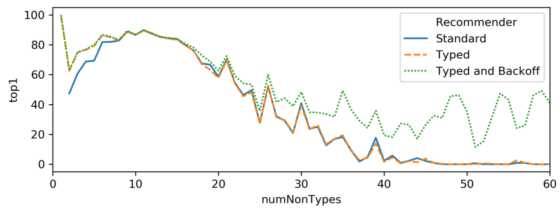


Figure 5: Results for TakeOneButType handler over multiple models.

The Standard Recommender is at peak performance when the entity contains around 10 properties, with perfect recommendations 80% of the time.

When there is a lack of input properties, the typing information will provide their biggest benefit, increasing the perfect recommendation rate by 20%. As more and more properties exist on an entity, types become inconsequential as they can be derived from the properties, and overall perfect recommendations become less frequent as entities become more specialized.

At around 25 properties, the back-off strategies start to kick in, making improvements of up to 40%. This is due to the fact that we enable property recommendations where the standard recommender was not able to recommend any property at all due to the to specific nature of the incoming property set. Those enabled recommendations are often that good that they end up at position one of the returned recommendation.

The small peak at 1 *numNonTypes* is due to the ubiquitous Type property that every entity holds. If the entity has a type to use as input for the recommendations, it will also have a property where said type was defined.

The reasoning stays the same at higher thresholds of topN, with values getting better on average due to our less strict definition of success.

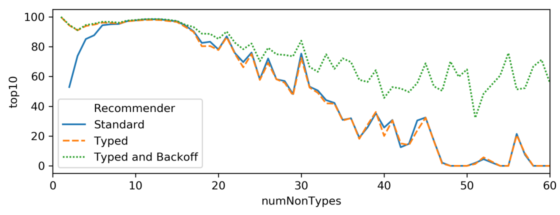


Figure 6: Results for TakeOneButType for top10 metric

The effect of the backoff strategies can be seen at its best when we look at the recall rate of the left-out property in figure 7. When the SchemaTree stops making recommendations because of unseen property combinations, the back-off will tone down the behaviour by making the recommender perform more general recommendations.

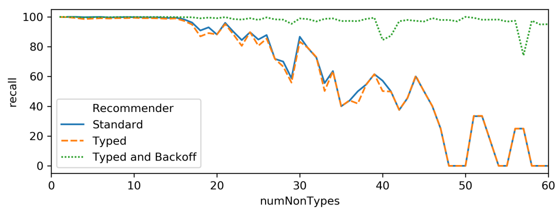


Figure 7: Ability to recall left-out properties with increasing property sizes

When we look at handler *TakeMoreButCommon* (figure 8) we simulate the reconstruction capability of the recommender with varying degrees of left-out properties. The `PrecisionAtL` metric will calculate how close to perfect this reconstruction is and should show us the impact of how type information improves the quality of recommendations.

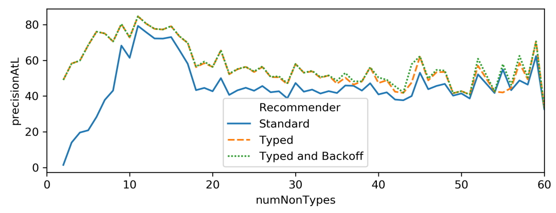


Figure 8: Quality of recommendations over total property sizes.

We see a clear indication of how additional type information improves the recommendation quality. At low number of input properties it has the biggest impact and can turn barely indistinguishable input sets into sets that can make perfect recommendations half of the time. It confirms that the definition of a type is a good indication of what properties we want to add next.

As the standard recommender reaches its peak at 10 total properties, the type information will add less extra value to the recommendation quality. It still sees improvements of around 5% with numbers reaching 10 to 15% at higher amounts of properties. The effect at play could be that types increase the certainty of how entities with high amount of properties will specialize themselves.

Overall, the addition of types is a huge improvement for the Wikidata optimizations, as it is a source of additional context information that the standard recommender did not use previously. When talking about recommendation quality the backoff strategies will not change much, as they are not aware of additional information and are primarily intended to increase the number of recommendations when there are few.

6 CONCLUSION

Throughout this laboratory we managed to make Wikidata-specific improvements in the recommender system, especially in cases where low amount of input is given, and in cases where a surplus of input is present. By introducing type information, the recall and precision for sparse entities was increased a lot, while backoff strategies especially improved the recall of entities with already a lot of properties. Our work has also improved the underlying structure to extend the standard SchemaTree with additional context-specific optimizations and has given some hints to what alternative measurements of recommendation quality could be defined.

7 FUTURE WORK

For future work, we would recommend some extensions to the evaluation handlers that currently exist. While using the *TakeMore-ButCommon* handler we used a single ordering of the properties according to their global count, with input sets preferring the most common properties. This handler could be expanded into a method that also evaluates multiple other orderings.

When obtaining the evaluation statistics, one could also look at bi-dimensional groupings to better understand how the recommender performs in all axes of invariants.

Additionally, further investigation in defining an optimal backoff configuration is needed. It is likely that the current backoff configuration is not yet optimal. Increasing the limit of the proportional stepsize function from 40% to a higher value should be part of future work as well as checking higher parameters for the tooFewRecommendation. Nevertheless, we showed that the chosen backoff configuration already improved the baseline recommender a lot. Finding better parameters therefore may increase the results further.

And lastly, if we somehow get information about the order on which each property was added to the system, we could build an optimal handler that mimics the existence of every entity over its lifetime.

REFERENCES

[1] Ziawasch Abedjan and Felix Naumann. 2013. Improving rdf data through association rule mining. *Datenbank-Spektrum* 13, 2 (2013), 111–120.

[2] Ziawasch Abedjan and Felix Naumann. 2014. Amending RDF entities with new facts. In *European Semantic Web Conference*. Springer, 131–143.

[3] Charu C Aggarwal and S Yu Philip. 1998. A new framework for itemset generation. In *PODS*, Vol. 98. 18–24.

[4] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, Vol. 1215. 487–499.

[5] Omar Alonso and Atul Kumar. 2006. System and method for search and recommendation based on usage mining. US Patent 7,092,936.

[6] Tim Berners-Lee, James Hendler, Ora Lassila, et al. 2001. The semantic web. *Scientific american* 284, 5 (2001), 28–37.

[7] Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee. 2008. Linked data on the web (LDOW2008). In *Proceedings of the 17th international conference on World Wide Web*. ACM, 1265–1266.

[8] Peter Buneman. 1997. Semistructured data. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 117–121.

[9] Michael J Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. 2008. Webtables: exploring the power of tables on the web. *Proceedings of the VLDB Endowment* 1, 1 (2008), 538–549.

[10] Wolfgang Gassler, Eva Zangerle, and Günther Specht. 2014. Guided curation of semistructured data in collaboratively-built knowledge bases. *Future Generation Computer Systems* 31 (2014), 111–119.

[11] Lars Christoph Gleim, Md Rezaul Karim, Lukas Zimmermann, Oliver Kohlbacher, Holger Stenzhorn, Stefan Decker, and Oya Beyan. 2018. Schema Extraction for Privacy Preserving Processing of Sensitive Data. In *Joint Proceedings of MEPDaW, SeWeBMeDA and SWeTI 2018*, Debattista et al. (Eds.). CEUR Workshop Proceedings (2112), Aachen, 36–47.

[12] Cornelia Gyorodi, Robert Gyorodi, T Cofeey, and S Holban. 2003. Mining association rules using Dynamic FP-trees. In *Proceedings of Irish Signals and Systems Conference*. 76–81.

[13] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. In *ACM sigmod record*, Vol. 29. ACM, 1–12.

[14] Taesung Lee, Zhongyuan Wang, Haixun Wang, and Seung-won Hwang. 2013. Attribute extraction and scoring: A probabilistic approach. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 194–205.

[15] Aiman Moyaid Said, PDD Dominic, and Azween B Abdullah. 2009. A comparative study of fp-growth variations. *International journal of computer science and network security* 9, 5 (2009), 266–272.

[16] Börkur Sigurbjörnsson and Roelof Van Zwol. 2008. Flickr tag recommendation based on collective knowledge. In *Proceedings of the 17th international conference on World Wide Web*. ACM, 327–336.

[17] Ching Y Suen and Rajjan Shinghal. 2014. *Operational expert system applications in Canada*. Elsevier.

[18] Natalia Vanetik, Ehud Gudes, and Solomon Eyal Shimony. 2002. Computing frequent graph patterns from semistructured data. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings*. IEEE, 458–465.

[19] Johanna Völker and Mathias Niepert. 2011. Statistical schema induction. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6643 LNCS, PART 1 (2011), 124–138.

[20] Eva Zangerle, Wolfgang Gassler, Martin Pichl, Stefan Steinhauser, and Günther Specht. 2016. An empirical evaluation of property recommender systems for Wikidata and collaborative knowledge bases. In *Proceedings of the 12th International Symposium on Open Collaboration*. ACM, 18.

[21] Eva Zangerle, Wolfgang Gassler, and Günther Specht. 2010. Recommending structure in collaborative semistructured information systems. In *Proceedings of the fourth ACM conference on Recommender systems*. ACM, 261–264.

APPENDICES

A IMPLEMENTATION

To implement the recommender we only need a valid RDF dataset. Given that we want to provide recommendations for Wikidata, the best possible dataset to use is the most recent dataset that Wikidata can provide. The dataset that we acquire is then passed through a preparation step and finally used to construct our models.

A.1 Initial Dataset

The most preferred dataset to use is the latest dataset that Wikidata provides. Wikidata will publish periodic dumps of the entire dataset in the form of periodic dumps[5]. Multiple versions are uploaded, but the one we choose is labelled as *latest-truthy*. It consists of an N-Triple file with only true statements i.e. it does not contain statements that are derived from other statements.

There are two big groups of subjects in Wikidata. Subjects can have either be an *Item* or a *Property*.

- *Item* subjects describe entities and generally have predicates describing their metadata, their properties, and their multilingual labels and descriptions.
- *Property* subjects describe properties and generally have predicates describing their metadata, their relation to other properties, and their multi-lingual labels and descriptions.

Listing 3: Example of an Item subject

```
<http://www.wikidata.org/entity/Q31>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://wikiba.se/ontology-beta#Item>
  .
<http://www.wikidata.org/entity/Q31>
  <http://schema.org/name>
  "Belgium"@en
  .
<http://www.wikidata.org/entity/Q31>
  <http://www.wikidata.org/prop/direct/P31>
  <http://www.wikidata.org/entity/Q6256>
  .
<http://www.wikidata.org/entity/Q31>
  <http://www.wikidata.org/prop/direct/P227>
  "4005406-8"
  .
```

A.2 Data Preparation

This dataset still contains excess of information so we pass it through a data preparation step. This preparation step is not strictly necessary, but it will remove an amount of excess information that is not usable by the tree, and with it increase recommendation speed, *SchemaTree* construction speed and evaluation speed.

Our first step is to identify the types of all subjects, so that we end up having only "Item"-type subjects in our dataset. However, we will actually split the dataset in 3 different subsets and have an use for all of them:

- a subset for all *Item*-typed subjects, which will be used by the recommender;
- a subset for all *Property*-typed subjects, which will be used by the glossary that will be explained further below in a separate section;
- a subset where all remaining subjects are send to — this subset can be analyzed manually to confirm that the split operation is working correctly.

[5]accessible via https://dumps.wikimedia.org/wikidatawiki/latest/

Each subject has a predicate called `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` that is used to decide the type of a subject. For *Items*, the object will be `<http://wikiba.se/ontology#Item>` while *Properties* use `<http://wikiba.se/ontology#Property>`.

After we get all *Item*-typed subjects in a single dataset we execute an additional filter step to remove all statements that are ubiquitous to all subjects. These statements usually include meta information and human-language descriptions and, since all subjects have them, will not be interesting for a recommendation system to recommend. This filter works by removing a predefined set of predicate IRIs that we manually define as being uninteresting by looking at their frequencies-to-subject ratios.

After the full data preparation, we obtain a dataset that contains only *Item* subjects with property predicates. That will be the dataset that is used to construct the SchemaTree.

A.3 Train and Test Split

If we want to construct a model for evaluation we make an additional split of the dataset into a training set and a test set. That split is done with regards of the number of subjects, with attention to keeping all statements of a subject in the same dataset.

When constructing the final model for production use we do not need to perform this split.

A.4 SchemaTree

The final step is the actual construction of the SchemaTree.

Since all recommendations are executed as read-only operations on the tree, we can construct it once and then store it as a binary tree for all later uses.

The tree consists of nodes and items. An Item is a struct that represents properties as well as types. An Item is identified by its string representation. To distinguish between properties and types, type representing Items have a prefix that classifies them as types. The property map includes all items that are built into the tree. The nodes are organized in a linked node structure. Starting from the root node, every node links to its children and its parent. Each node is associated with exactly one Item, while every Item is associated with one or more nodes. To associate items with their nodes, an Item has a traversal pointer linking to the first of its nodes. Furthermore, a traversal pointer connects all nodes corresponding to a specific Item.

We use a two-pass tree construction. In the first pass, we count how often every type and property appears in the dataset. Based on this information, we assign a sort order index to every Item. Every Item has a unique sort order index regardless of being a type or property. To detect the types, we have a slice containing the string representations of classifying properties. The values of such classifying properties are recognized as types. Currently, we are only using the P31("InstanceOf") property of wikidata. By adding more classifying properties, the tree construction can be used for further datasets. In the second pass, we are creating a path of nodes through the tree. If a node does not exist, we create it; otherwise, we increment its support.

A.5 Glossary

The SchemaTree operates by using the IRIs of properties, but user interaction with the recommender requires us to show a human-readable representation of every property. The *Glossary* is a structure that allows us to map the IRI of every property to a multi-lingual representation of it. The human-readable representation comes in the form of two predicates:

- labels, that give the property a succinct name;
- descriptions, that provides a summary of the property.

Every representation can come in multiple languages. When the user interacts with the recommender, they may select a language for that interactive. Whenever possible, the glossary will append human-readable representations to all properties that it recommends.

B SOFTWARE

The Recommender software consist of a code repository written in the *Go* language, which can can be built into 2 command-line applications. One, the main application, that holds most of the tools, and another used for all evaluation purposes.

The multiple tools are stored as sub-packages on the repository. Some sub-packages can be used in isolation, while others will build on top of existing sub-packages. The entry functions of both the main and evaluation applications will then orchestrate all necessary tools together to provide its features.

Operations on files will generally never mutate those files and will instead create new ones with similar names. Every operation will take the name of the original file and append the operation name to it. What we get is a good description of everything that happened to a file, which helps in keeping them organized, at the mild inconvenience of having big file names.

Listing 4: File names at different stages of the pipeline

```
truthy.nt.gz
truthy-item.nt.gz
truthy-item-filtered.nt.gz
truthy-item-filtered-1in1000-test.nt.gz
```

B.1 Main Application

The main application is a command-line interface allows us prepare datasets, build SchemaTrees, build glossaries, and setup a webserver that will answer to recommendation requests. The interface is build using the Cobra library [6] and features a commonly seen style of `APPNAME COMMAND ARG --FLAG`, which is used in a wide range of high-profile command-line interfaces.

Every operation that can be executed with the CLI can be performed separately. This is done to allow for a fair amount of flexibility when operating on the datasets and models. However, in order to execute the entire workflow, starting from data preparation to SchemaTree construction, the operator is required to chain multiple CLI operations together. Whenever the CLI generates new files it does so in a fixed manner, so it is very possible to execute the entire workflow as a batch script for easier automation.

The possible commands with their respective sub-commands are:

- **split-dataset** - Contains multiple procedures to split a N-Triple RDF dataset. Split operations will never change the input file and will always generate new files. The chosen splitting method will define what criteria is used for the split and what files are generated.
  - **1-in-n** - Accepts a dataset where subjects are grouped together in contiguous lines and will make a split by sending a number of subjects into one file and the rest into another file. The ratio is calculated by $\frac{1}{N}$, where N can be chosen by the user. This splitting method is generally used to split datasets into training and testing sets, or alternatively, into training and validation sets.
  - **by-type** - This splitting method will classify each subject according to the object of a specific predicate. After the classification, it will send all of the subjects statements into one of three possible files: a file containing all items, one containing all properties and one for all remaining statements.
  - **by-prefix** - Like the *by-type* splitter, it will also send the statements into one of the three mentioned files, but the criteria to do so is evaluated by checking the prefix of the subject.
- **filter-dataset** - Dataset filtering implements simple filter-out operations, where statements with specific sets of predicates are removed from the dataset. The input dataset file is not altered and instead a new file is generated that holds statements that passed the filtering. A sub-command is used to select which set of predicates is used for the filter-out operation.
  - **for-schematree** - This set of predicates will filter-out statements with the goal of removing all statements that are superfluous to the recommendation performed by the SchemaTree. The predicates that are filtered-out consist of meta information and human-readable descriptions. Executing this step will make the SchemaTree smaller, improve the recommendation speed by reducing the amount of traversed nodes, and prevent it from making recommendations of non-properties.
  - **for-glossary** - This set of predicates will filter-out statements that are irrelevant for the glossary. It is typically applied to a dataset that only contains properties. The most commonly removed statements include descriptions for non-Wikidata namespaces.
  - **for-evaluation** - Will remove statements with the aim of reducing the time spent on evaluation without altering the results. Typically all statements that are not included in the tree, so that future leave-out-N techniques on the properties will only deal with properties that exist. Without this, it could happen properties are left out that do not

[6]https://github.com/spf13/cobra

exist in the SchemaTree, resulting in duplicate evaluation results, since non-existing properties are silently ignored.

- **build-tree** - Will use a dataset to build a SchemaTree. In contrast to the tree created by *build-tree-typed*, the Schematree will only include properties as nodes and make no use of type information.
- **build-tree-typed** - This command will construct a SchemaTree with typing information. This model will make use of additional information provided by predicates that are associated with a type system.
- **build-glossary** - Will build a mapping between subject IRIs and their human-readable and multi-lingual descriptions. For every subject, multiple pairs of (label, description) mapped, one pair for each language code. This glossary is later used to enrich the recommendations with their human-readable representations.
- **serve** - Sets up a webserver that handles HTTP requests for recommendations. To setup the server it will require a SchemaTree and a Glossary.

As previously mentioned, every splitting, filtering and building step can be executed separately to allow for more flexibility. This is important because there are at least two pipelines on how the software can be used.

- **For evaluation:** When we are measuring new techniques and backoff strategies we are only interesting in constructing the SchemaTree. In this case we need to execute more splits on our ”Items“ dataset in order to generate training and test sets. We might also require the generation of multiple permutations of backoff configurations, which are explained in the section below [?], whereas any operation concerning the Glossaries can be skipped.
- **For production:** When we are certain of which backoff strategy is used for recommendations in production environment we can construct a pipeline that automatically downloads the latest wikidata dump, prepares the data, builds both the SchemaTree and Glossary, and starts the webserver for incomming user requests. In this case we do not need to make splits based on training and test data, but have to deal with the ”Property“ datasets.

B.2 Evaluation Software

The application with which all evaluations are made is part of the same repository, but is not integrated into the same CLI as the others. The reason for this is both the back-office nature and the code history. The evaluation sub-package is a software capable of obtaining performance metrics from built models. These models can be combined with multiple models, workflows, methods for generating input sets, and statistic aggregations.

Flexibility in the evaluation is attained by allowing the operator to mix any combination of 4 components and extract the resulting statistics. The components are: the Model (Standard SchemaTree or Typed SchemaTree), the Workflow (using a direct recommendation of the model or any custom configuration of back-off strategies), the Handler (method used to define property query and expectation sets), and the Grouping (how statistics are aggregated together).

The end result of an evaluation is a CSV file with the metrics described in the Evaluation section. The file will also make use of the naming scheme and describe exactly which components have been used to generate the file.

B.3 Workflow and Assessments

In the following section we deal with the *workflow structure*. The workflow structure is the framework where we can easily plug in different conditions and backoff strategies. A workflow contains layers $l_1, ..., l_m$ each containing a condition and a backoff strategy. When a new recommendation request enters the recommender, then the it passes l_1 first. If l_1 condition is met, the backoff strategy is executed and the resulting recommendation returned. If not, the next layer's condition is tested. The last layer l_m then should be the standard recommender which is always executed (condition is always true), so that if a recommendation request does not trigger any condition, the standard recommender is executed. In fig. 9 we see an two exemplary workflows: Left the standard recommender in the workflow format and right a workflow with one backoff layer and the fallback layer.

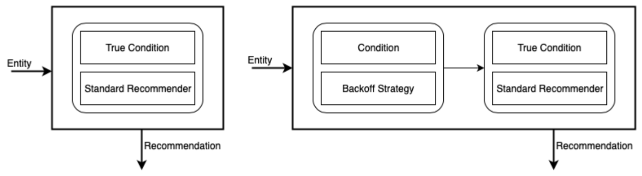


Figure 9: Two exemplary workflows. Left, the standard recommender represented in the workflow model. Right, a backoff layer followed by the standard recommendation layer is visible.

The workflow itself is implemented as a stack-like structure. It is possible to add further layers by pushing those to the workflow, enabling high flexibility. The flexibility is enhanced further by allwoing to define the workflow via configuration json files, which defined each layer of the workflow in a easy human readable (and writable) way. An example of such a configuration file can be found in listing 5. The exact definition of the attributes and structure can be found in the projects `configurations/README.md`.

```
{"Layers":[
  {"Condition":"tooFewRecommendations",
   "Backoff":"deleteLowFrequency",
   "Threshold":1,
   "Stepsize":"stepsizeProportional",
   "ParallelExecutions":6},
  {"Condition":"always",
   "Backoff":"standard"}
]}
```

Listing 5: Config File Format Example

Since many conditions actually need to perform the standard recommendation, there is high potential of re-usability. Therefore we introduced assessments, which cache the standard recommender's recommendation as soon as it is computed the first time. Again look at fig. 9 (right) and assume that the standard recommender is executed inside the first condition check. The the assessment caches the result, such that in case of not firing the backoff strategy, the recommender does not compute the same result in the second layers' standard recommendation and instead uses the cache.

B.4 Parallelism in Backoff Strategies

Backoff Strategies run the standard recommender in parallel. In case of the Split Property Recommender, the main process needs to wait for both go routines to finish. As a side note, the code is constructed in such a way that even more splits are possible to run in parallel. This feature is not used though, since splitting in two subsets already deletes too much context of each subset, resulting in less good results than the Delete Low Frequency approach. In case of the Delete Low Frequency approach we can early continue with the main process as soon as a recommender with low changes to the incoming property set computes a recommendation which satisfies the condition of that layer. In such a case the main process does not wait for all go routines to complete.

B.5 Configuration Batch Testing

To find optimal parameters for the backoff strategies, we implemented a framework that tests a bunch of config files at a time and creates an aggregated view on each configuration in a single csv table. This result is in the end visualized via python matplotlib. The config files are extensions of the json format in **??**. Since generating a huge number of configuration files is a tedious task, the framework also contains a config file generator, which gets a config configuration file and generates a bunch of workflow configurations from that. A further explanation of the configuration formats can be found in `evaluation/configs/README.md`.

B.6 Integration and Testing

This subsection describes the git work flow and the continuous integration pipeline. To ensure stability of the code base, we used a master branch and feature branches. Developers work on their feature branch, and merge into the master branch, when the task is completed. The master branch contains a continuous integration set up, which executes an automatic build, when a feature branch is merged. This ensures, that the master branch always builds with no errors. Additionally, two testing approaches are implemented. After the build, the whole system is tested by performing a test run with a test data set. This detects commits, that harm the overall system quality. To test specific methods, unit tests were implemented. These help to detect errors in individual parts of the system.

C INDIVIDUAL IMPRESSIONS

This section contains feedback from the team members.

C.1 Maximilian Peters

My first task in the recommender project was to set up a continuous integration pipeline for the already existing Go project. I already was familiar with GitLab CI, but not together with the programming language Go. Since I did neither knew the language, nor how to build its sources, this was not trivial. After that, I started to learn the language itself. Again, since we already had a complex project, I needed to understand that, too. This had two sides, a more technical side regarding the implementation, and a more theoretical point of view. Additionally, I had to deal with concurrency, which sometimes can be tricky. My last practical assignment was to figure out a way to evaluate the recommender properly. We tackled that as a team, since it was challenging. It again included a theoretical point of view, which needed to be thought of. Apart from the practical tasks, I learned how to work on a project as a team. This includes using tools like GitLab to organize issues and branches and Slack for communication. Additionally, it was interesting to learn, how to assign subtasks to the most capable or willing. Finally, it was very helpful for me to have people that I could ask if I didn't understand a code line or theoretical idea. Overall, I enjoyed the project even tho it was a lot of work, due to an interesting task and a great team.

C.2 Dominik Hüser

The first task of mine was to get myself familiar with the topic of recommenders. Understanding the schematree recommender and wikidata model was crucial for the further steps. The next step was to define improvements for the case of to many properties in the incoming entity. I came up with the definition of the different conditions and backoff strategies. I wrote batch tests to find proper parameters and combinations of the conditions and backoffs (config generator and batch evaluation). I additionally created the visualization of all results via python matplotlib and seaborn. Finally me and Rafael were responsible to run the test servers.

During the lab I learned a lot: First, I gained knowledge in the area of recommendation systems and how the wikidata project works. RDF wasn't new to me but I still gained further knowledge here. Golang was completly new to me, so that this was nice to learn, too. I really like coding in go. Additionally I could gain more experience in working with python matplotlib and seaborn as well as to work server-side, which I wasn't too familiar with beforehand. Since my bachelor lab did not focus a lot on the software development aspect, I improved in working "professionally" too. Working with multi-feature branches, kanban and CI was a great experience and I think I could actually take a lot with me here. Defining tasks in a team, managing team meeting together, creating presentations etc. was definitely improving soft skills, too. All in all it was a fun time, a great team, and a lot of work, but worth it.

C.3 Rafael Schimassek

The project started with learning how the SchemaTree operates, learning all the fundamentals of the *Go* programming language and understanding how Wikidata handles its data.

My main role in the team was to support other features by building the underlying coding structures and refactoring the existing code to make them easier to extend. I've build the abstractions for the workflow, assessments and the Glossary. An improvement of the N-Triple reader was also required so that it would read quoted content and support multi-language string literals. Some refactoring was also necessary to allow easier extension of the splitter, evaluation and webserver. Lastly, the previous applications were all unified under one main CLI application to make operations easier.

Some study was needed to understand the structure of the wikidata dump, as it differed from the initial 10M testset that we had, and joint work with Dominik was done in order to get these dumps to run on the testing servers.

Near the end I contributed with some intuitions behind the evaluation metrics and extended the ideas behind the naming scheme to make the overview a bit easier. Organization and modular code is of big importance to me as I believe that it allows research to be done more smoothly and lowers the barrier of entry for researchers to test out new methods on the computer.

Overall I really enjoyed working with the team. They were motivated and took care of what needed to be done. Communication was on point and task allocation was very smooth.

I left this laboratory with better knowledge about RDF Knowledge datasets, improved understanding on the intuition behind evaluation metrics, and experience with a modern approach to a statically-typed garbage-collected programming language.

C.4 Christoph Krämer

Before the lab, I had little practical experience. My expectation on this lab was to get more experience in working on software projects. Also, I was interested in how knowledge graphs are used in practical real-world scenarios. My main responsibility was the integration of type information into the recommendation system. I needed to learn the theoretical aspect of the recommendation as well as its actual implementation. Further, I needed to learn how types are working and find a way to integrate type information into the existing tree. I decided to transfer the method used by wiki data on the schema tree, as this method seems to work quite well on the schema tree. Nevertheless, we needed to evaluate our results. We needed to think about how we compare the typed recommendations to the untyped recommendations.

The lab worked quite well in increasing my practical experience. I learned the programming language GO, which was new to me. I already knew gitlab, but I never actually used gitlab within teamwork. So I learned how gitlab (and slack) can be utilized to support the workflow in larger software projects. I also benefited from my competent team in a way that I got to know new software architectures/design pattern. Working with my team was generally a nice experience. Ultimately this lab gave me more confidence in joining software projects.