Swift Linked Data Miner: Mining OWL 2 EL Class Expressions Directly from On-Line RDF Datasets

J. Potoniec[a,*], P. Jakubowski[a], A. Ławrynowicz[a]

[a] *Faculty of Computing, Poznan University of Technology, ul. Piotrowo 3, 60-965 Poznań, Poland*

Abstract

We present Swift Linked Data Miner, an interruptible algorithm that can directly mine an on-line Linked Data source (e.g. a SPARQL endpoint) for OWL 2 EL class expressions to extend an ontology with new SubClassOf: axioms. The algorithm works by downloading only a small part of the Linked Data source at a time, building a smart index in the memory and swiftly iterating over the index to mine axioms. We propose a transformation function from mined axioms to RDF Data Shapes. We show, by means of a crowdsourcing experiment, that most of the axioms mined by Swift Linked Data Miner are correct and can be added to an ontology. We provide a ready to use *Protégé* plugin implementing the algorithm, to support ontology engineers in their daily modelling work.

Keywords: linked data, on-line linked data mining, ontology learning, OWL 2 EL, RDF Data Shapes, *Protégé* plugin
2010 MSC: 68T05, 68T10, 68T27, 68T30

1. Introduction

It is reasonable to assume that an RDF (Resource Description Framework [49]) graph is not a random set of triples. That is, there is some reason for the graph to have this specific shape: some underlying data model or a process generating the graph. An element of the model is reflected as a pattern in the graph. The idea of Swift Linked Data Miner (SLDM) is to make use of this observation backward: if there is a pattern in the graph, then there is a good amount of chance that there should be a corresponding element in the model.

One of possible ways to express such a model is to generate an ontology. To express an ontology for an RDF graph, one can employ RDF Schema [15] or aim for some variant of OWL 2 (Web Ontology Language [34]). RDF Schema offers just a tiny bit of expressive power, so we decided to choose OWL 2 instead. One may argue that OWL 2 has exactly the opposite problem, as it is a very expressive language and thus reasoning complexity is unbearable. Fortunately, there are three OWL 2 profiles: OWL 2 QL, OWL 2 RL and OWL 2 EL, all of them provide polynomial-time reasoning algorithms. According to the OWL 2 specification [14], the first two are suitable for lightweight ontologies. On the other hand OWL 2 EL is a profile tailored specifically to deal with very large ontologies. It is our intuition that one can easily get a large ontology if one does not have to develop it by hand, but has it data mined instead. It must be noted here that the OWL Lite profile, defined for OWL [7] and inherited by OWL 2 has exponential reasoning complexity [21]. Taking all these into account, we decided that our primary way to express the model is an OWL 2 EL ontology, which we introduce in Section 3.

In the age of Big Data, we can not assume anymore that we can have a direct access to a graph all the time and fit the whole graph to the RAM. We must be able to process the graph chunk at a time, to sample it, and to retrieve the chunks from a remote location. Swift Linked Data Miner was developed with all this in mind. In Section 4, we show how we retrieve only a part of an RDF graph at a time and organize the retrieved part into a smart structure to facilitate pattern mining.

In Section 5 we describe SLDM, which can mine an RDF graph to extend an OWL 2 EL ontology. SLDM is composed of many small algorithms, each fitted to mine particular type of patterns. Section 6 provides a theoretical analysis of the algorithm, including both analyses of memory complexity and worst-case computational complexity.

Some users may not be interested in reasoning. Maybe the logical inference in their application is unnecessary and instead they would like to understand better their graph, or validate if a new graph they just received is compatible with the model. Especially to address these issues there is an on-going work on RDF Data Shapes[1], and we provide an alternative way to express patterns mined with SLDM. In Section 7, we show how the patterns can be transformed to RDF Data Shapes expressed in Shapes Constraint Language (SHACL) [24].

To facilitate incremental research and enable early adopters to use our ideas, we provide an implementation of Swift Linked Data Miner. In Section 8, we present a plugin to *Protégé*, which enables ontology engineer to use SLDM in just few clicks.

To validate SLDM we planned and conduced a crowdsourcing experiment using arguably the most popular Linked Data

*Corresponding author
Email addresses: jpotoniec@cs.put.poznan.pl (J. Potoniec), pjakubowski@cs.put.poznan.pl (P. Jakubowski), alawrynowicz@cs.put.poznan.pl (A. Ławrynowicz)

[1] https://www.w3.org/2014/data-shapes/charter

resource: *DBpedia* [4]. We used SLDM to mine patterns for multiple classes used in *DBpedia*, translated the patterns to English and asked the contributors of a crowdsourcing platform if these sentences correctly describe the classes. They decided that most of them are indeed correct. Details of the experiment are described in Section 9.1.

In Section 9.2, we show another use-case, base on myExperiment RDF dataset and the associated ontology [40]. We compare the mined patterns with documentation and pragmatics of the dataset. Finally, in Section 9.3, we discuss run-time properties of the algorithm, such as CPU time in function of various parameters of the algorithm.

2. Related work

The idea of automated and semi-automated creating and extending ontologies was considered in many papers. These papers can be roughly divided into five areas: i) ontology learning from text, ii) interactive ontology learning, iii) concept learning, iv) learning from local data, v) learning from on-line data. The first area is also the farthest from our work, and thus we will not discuss it. The interested reader is refereed to a comprehensive compendium on ontology learning [28].

One of the prominent ways of interactive ontology learning is an application of formal concept analysis to the Description Logics [2, 41, 37, 39]. The aim of these algorithms is to complete an ontology w.r.t all subsumptions of a given type, e.g. between named classes [2] or between named classes and domain and range restrictions [41]. The algorithms generate all possible subsumption axioms that are consistent with the ontology, but do not follow from it, and for each axiom the user is asked to either add the axiom to the ontology or to provide a counterexample.

In the same area fits the idea of games with a purpose, where a player receives rewards for completing tasks like creating new entities in an ontology, adding types or aligning an ontology with another one. The consensus is obtained by posing the same task to multiple players. A classical example of such a system are *OntoGames* [43]. Similar to the games with a purpose is the idea of using crowdsourcing services like *CrowdFlower*[2] or *Amazon Mechanical Turk*[3]. For example, in [16] the authors present a *Protégé* plugin that enables a user to post microtasks related to her ontology development process directly to a crowdsourcing website.

Concept learning by itself is not necessarily an ontology learning approach, as it is concerned with learning class expressions given a set of positive and negative examples. Nevertheless, such an approach can be used to extend an existing ontology with missing definitions. The authors of [9] propose DL-FOIL algorithm, which allows for learning class expressions in Description Logics underlying OWL-DL [46]. The learning algorithm is based on sequential covering and employs two refinement operators, one for specialization and the other one for generalization. [26] describes a concept learning software *DL-Learner*[4], which also employs refinement operators, but with different search strategy. In [27] application of *DL-Learner* specifically to learning ontologies is described, along with a *Protégé* plugin implementing the idea. An approach using a refinement operator with background knowledge to refine SPARQL queries was proposed in [25]. These queries are further used as binary features for a classical machine-learning classification algorithm. Due to certain properties of these SPARQL queries, they can be immediately transformed to OWL class expressions, and thus also used in ontology learning, as discussed in [38]. The authors of [13] employ techniques known from Inductive Logic Programming on top of their in-memory RDF store to mine association rules with variables. Such rules could be then transformed into an ontology.

The idea of learning ontological axioms from a static, fully-available dataset is very related to various forms of data mining in relational databases. For example, in [11] the authors propose a framework for mining association rules in relational databases guided by constrains on a shape of mined rules. By setting these constrains appropriately, one could obtain an ontology. In [48, 10] the authors propose an algorithm for mining an OWL 2 EL ontology from scratch. The input to the algorithm is an RDF graph, which is first transformed into a set of database tables, and then association rule mining is employed to discover the ontological axioms. Instead of considering an RDF graph and an ontology describing it, one could tackle the problem of extending an ontology using individuals contained in it. [42] provides a method for learning general class inclusions (GCIs) that takes into account results of reasoning with the ontology.

Finally, one may want to extend an ontology using data that are not entirely available at hand, but for example are available only in remote SPARQL endpoints [17]. To the best of our knowledge, [5] is the only work so far to address this problem. It is a top-down method, which first performs a data mining on a repository of ontologies to build a library of patterns and then the patterns are used to form SPARQL queries, which are posed to a SPARQL endpoint to discover axiom candidates. Unfortunately, the queries are computationally expensive for the endpoint, due to the heavy usage of `GROUP BY` and `COUNT DISTINCT`.

3. Preliminaries

3.1. OWL 2 EL

OWL 2 is a language designed to describe information about entities and relations between them. The language provides formally defined semantics and decidable reasoning procedures [34]. OWL 2 EL is a subset of OWL 2, tailored to support applications employing very large ontologies while having typical reasoning tasks tractable [32]. For example, this subset is used by a large clinical health ontology SNOMED CT [8].

[2] `http://www.crowdflower.com`

[3] `http://www.mturk.com`

[4] `http://dl-learner.org`

Throughout this work we write OWL expressions in Manchester Syntax [20]. We also use a set of well-known prefixes: `rdf:` for the namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, `rdfs:` for `http://www.w3.org/2000/01/rdf-schema#`, `owl:` for `http://www.w3.org/2002/07/owl#` and `xsd:` for `http://www.w3.org/2001/XMLSchema#`.

We start by defining an *OWL 2 EL class expression*. Let A be a named class, DT a datatype, p a named object property and r a named data property. Moreover, a denotes an individual and l a literal. Following [32], the OWL 2 EL class expression C^{EL} is defined as follows:

$$\begin{aligned} C^{EL} :=& A | C^{EL} \text{ AND } C^{EL} | \{a\} | p \text{ SOME } C^{EL} | p \text{ VALUE } a | p \text{ SELF} | \\ & r \text{ SOME } R^{EL} | r \text{ VALUE } l \\ R^{EL} :=& DT | R^{EL} \text{ AND } R^{EL} | \{l\} \end{aligned}$$

The datatypes in OWL 2 EL are severely limited compared to the full OWL 2. There are only 19 datatypes that were chosen such that their intersections are either infinite or empty [32]. Magka et al. proved that OWL 2 EL class expressions can be extended with inequalities over numeric domains [29]. For a SUBCLASSOF: axioms with a subclass expression in OWL 2 EL, a superclass expressions can be extended with $\geq$ and $\leq$ relations over real, rational and integer numbers, and with $\geq$ relation over natural numbers. Following the idea and using the same symbols as before, we can define the *OWL 2 EL superclass expression* C:

$$\begin{aligned} C :=& A | C \text{ AND } C | \{a\} | p \text{ SOME } C | p \text{ VALUE } a | p \text{ SELF} | \\ & r \text{ SOME } R | r \text{ VALUE } l \\ R :=& DT | R \text{ AND } R | \{l\} | LT[<= l] | GT[>= l] \\ LT :=& \texttt{owl:real} | \texttt{owl:rational} | \texttt{xsd:decimal} | \texttt{xsd:integer} | \\ & \texttt{xsd:dateTime} | \texttt{xsd:dateTimeStamp} \\ GT :=& LT | \texttt{xsd:nonNegativeInteger} \end{aligned}$$

According to the specification [3], types `xsd:dateTime` and `xsd:dateTimeStamp` closely correspond to decimal numbers, so we can safely incorporate them in the definition.

3.2. Datatypes in OWL

OWL heavily relies on the datatypes defined for XML Schema Definition Language [3]. These datatypes are organized into such a hierarchy, that a valid value for a subtype is also a valid value for a supertype, e.g. 3 is a valid value for the `xsd:nonNegativeInteger`, but also for `xsd:integer` and all its supertypes, including `rdfs:Literal`. In Figure 1 we present the datatype hierarchy for OWL 2 EL, based on the definitions of the value spaces for the datatypes [3, 15, 36].

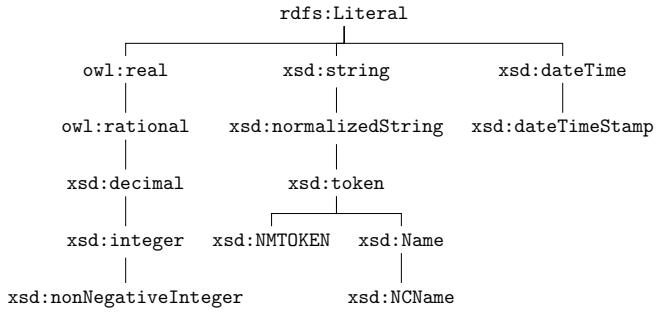


Figure 1: The hierarchy of datatypes in OWL 2 EL. For clarity, omitted are direct children of `rdfs:Literal` with no children themselves, i.e. `rdf:PlainLiteral`, `rdf:XMLLiteral`, `xsd:hexBinary`, `xsd:base64Binary`, `xsd:anyURI`.

4. Three level index for an efficient access to an RDF graph

We aim at data mining directly on Linked Data, that is we want to be able to operate in a setup with only parts of an RDF graph available at a time. We consider a SPARQL endpoint with a very simple SPARQL queries posed to it, but our approach could be easily adapted to other, less restrictive (in terms of server workload) frameworks available in the Web such as Linked Data Fragments, subject pages or even data dumps, all described in [47].

Precisely, for a given set of URIs $\mathcal{U}$ we build a set of SPARQL queries based on the following template:

```
select ?s ?p ?o
where
{
    ?s ?p ?o.
    values ?s {...}
}
```

We replace `...` with small disjoint subsets of $\mathcal{U}$. In the implementation described in Section 8 the number of items in a subset is a configurable parameter with a default value of 100 URIs. Such a query extracts all triples `(?s, ?p, ?o)` from the endpoint, but with a restriction that all the bindings for the variable `?s` must be from the set of URIs listed in the curly braces after the `values` keyword, i.e. from the set that replaced `...` in the template. The idea is to avoid posing $|\mathcal{U}|$ queries to the endpoint on one hand, thus lessening network and CPU load, but keep them simple enough to avoid issues with queries being executed too long.

If the set of URIs $\mathcal{U}$ is large, the number of queries posed by SLDM may be too large for the used SPARQL endpoint. To further decrease its load, it is possible to use sampling. We propose three different strategies for sampling:

uniform The set $\mathcal{U}$ is shuffled and a fixed number of items (e.g. 1000) is kept, the rest discarded. In other words, we perform simple random sampling without replacement. This is the baseline strategy, that does not use any additional knowledge during the sampling and does not pose any SPARQL query to the endpoint.

predicates counting As SLDM constructs pattern by detecting features shared by a substantial number of URIs, the richer the description of an URI in the RDF graph, the more patterns it can support. One of possible measures of descrip-

tion richness is the number of distinct predicates used to describe a particular URI. First, for every URI the number of distinct predicates occurring in triples with the URI as the subject is counted. It is relatively easy to obtain these numbers using SPARQL 1.1 queries constructed from the following template:

```
select ?s (count(distinct ?p) as ?c)
where
{
    ?s ?p [] .
    values ?s {...}
}
group by ?s
```

The ellipsis in the `values` clause is to be filled by splitting the set $\mathcal{U}$, as described earlier. The obtained numbers are then normalized by their sum (i.e., now the sum is equal to 1) and the normalized numbers are used as a probability of a particular URI being chosen.

triples counting This strategy employs a very similar idea, but it uses the number of triples instead of the number of distinct predicates as the measure of description richness. The following query template is used:

```
select ?s (count(?o) as ?c)
where
{
    ?s ?p ?o .
    values ?s {...}
}
group by ?s
```

Notice that the keyword `distinct` was removed from the query compared to the previous one. The obtained numbers are again normalized and used as the probability.

The uniform strategy guarantees reduction of the workload, while the other two strategies do so only if the endpoint is capable of efficient answering of group by queries, which may vary depending on the software running the endpoint. Using sampling we can significantly reduce the workload, but as with every sampling there is a price: it is possible to introduce mistakes to the mined patterns. As the counting strategies favor the URIs that are more richly described, we think that using them should mitigate some of the sampling issues in case of an RDF dataset of uneven quality. We provide experimental comparison of the strategies in Section 9.3.2.

The retrieved triples are stored in a three-level hash-based index structure with predicates in the first level, objects in the second level and subjects in the third level. An example of such an index is presented in Figure 2. Building the index using hashing is a relatively cheap task: three hash computations for every triple plus amortized constant time insertions. This particular organization of the index, i.e. first predicates, then objects, then subjects, is caused directly by the way our algorithm operates. Using a different order (e.g. predicates – subjects – objects) would increase the computational complexity of the algorithm. This is because the algorithm operates by grouping together different subjects from triples having the same predicate and object. The full algorithm for building an index is presented in Algorithm 1. Its input is a set of triples $\mathcal{T}$ and it outputs the index $\mathcal{I}$.

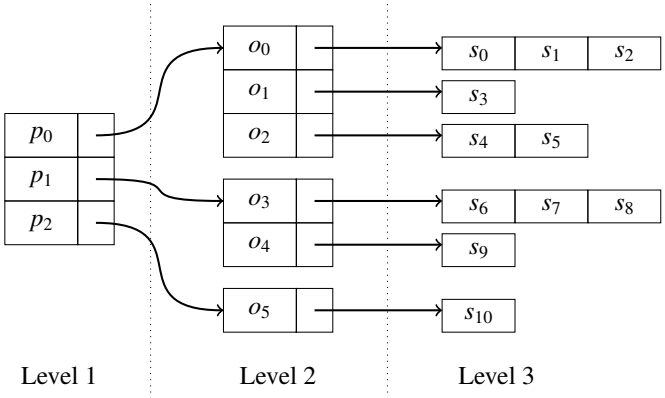


Figure 2: A sample three level index used in SLDM. The first level consists of predicates, the second level of objects and the third of subjects. Empty rectangles denote pointers to the next level. There are 11 triples in this index: 6 with predicate p_0, 4 with predicate p_1 and 1 with predicate p_2, namely (s_{10}, p_2, o_5).

```
1 function BuildIndex(T)
2     I ← ∅          // An empty three-level index
3     foreach s, p, o ∈ T
4         if p ∉ I
              // We encountered a new predicate,
                 initialize data structures for it
5             I[p] ← ∅
6         if o ∉ I[p]
              // We encountered a new object for a
                 predicate p
7             I[p][o] ← ∅
8         I[p][o] ← I[p][o] ∪ {s}
9     return I
```

Algorithm 1: The algorithm building a three-level index for a given set of triples $\mathcal{T}$. Square brackets are used as a notation to access the index, e.g. $\mathcal{I}[p]$ refers to a part of the second level of the index, which is pointed by a pointer attached to p in the first level.

5. Swift Linked Data Miner

5.1. A motivating example

Throughout the following sections we use a small example derived from *DBpedia* [4]. This example is about five books authored by J. R. R. Tolkien: *The Hobbit*, *The Silmarillion*, and three volumes of *The Lord of the Rings*, namely *The Fellowship of the Ring*, *The Two Towers* and *The Return of the King*. The whole graph we will use is presented in Listing 1. This graph contains cherry-picked triples to clearly and concisely support the example.

Assume that a set of URIs $\mathcal{U}_1$ consists of the following five URIs: `:The_Hobbit`, `:The_Silmarillion`, `:The_Fellowship_of_the_Ring`, `:The_Two_Towers`, `:The_Return_of_the_King`. In Figure 3 presented is a three-level index build from the triples having subjects in the set $\mathcal{U}_1$.

5.2. Frequent pattern

Frequent pattern mining is a task originating from data mining in databases. A typical example considers a database of market baskets of items bought together. The aim is to find all sets of items frequently bought together, that is occurring in at least given percent of the baskets in the database [1].

In this work, we are interested in a slightly different approach. We would like to find a set of possible superclasses for a given class, i.e. we would like to find patterns partially describing a given set of objects in a given RDF graph. By a *pattern* we understand an arbitrary OWL 2 EL superclass expression C, as defined in Section 3.1.

To define a frequent pattern, we first define a *matching function* $\mu_{\mathbb{G}}(a, C)$. The intuitive meaning for the function is that $\mu_{\mathbb{G}}(a, C) = 1$ iff an individual or a literal a matches a pattern C w.r.t a graph $\mathbb{G}$ and 0 otherwise. The full definition of the function is presented in Table 1.

For example, consider the RDF graph presented in Listing 1.

$$\mu_{\mathbb{G}}(\texttt{:The\_Hobbit}, \texttt{dbo:Book}) = 1$$

because there is a triple `(:The_Hobbit, rdf:type, dbo:Book)` in the graph. On the other hand,

$$\mu_{\mathbb{G}}(\texttt{:The\_Two\_Towers}, \texttt{dbo:illustrator} \text{ VALUE } \texttt{:J.\_R.\_R.\_Tolkien}) = 0$$

as there is no triple `(:The_Two_Towers, dbo:illustrator, :J._R._R._Tolkien)` in the graph.

We define frequency based on the measure of support. Consider a set of URIs $\mathcal{U}$ and a set of literals $\mathcal{L}$. A *weighting function* w is an arbitrary function $w\colon \mathcal{U} \cup \mathcal{L} \to [0, 1]$. A *support* of a subset $S \subseteq \mathcal{U} \cup \mathcal{L}$ given a weighting function w is defined as:

$$\sigma(S, w) = \sum_{s \in S} w(s)$$

Recall the graph from Listing 1. Consider first a uniform weighting function w_1, such that $w_1(s) = \frac{1}{5}$ for all $s \in \mathcal{U}_1$. The three volumes of *The Lord of the Rings* correspond to a set $S = \{$`:The_Fellowship_of_the_Ring`, `:The_Two_Towers`, `:The_Return_of_the_King`$\}$, and its support is $\sigma(S, w) = \frac{3}{5}$. The weighting function may be defined differently, e.g. let w_2 be $\frac{2}{6}$ for `:The_Fellowship_of_the_Ring` and $\frac{1}{6}$ for the rest of the set $\mathcal{U}_1$. In such a case, the support of the set S would be $\sigma(S, w_2) = \frac{4}{6}$. More details on the weighting function is given in Sections 5.7 and 5.8, where we show how it is used and recomputed in SLDM.

Observe, that given a set $\mathcal{U} \cup \mathcal{L}$ and a pattern C, the matching function defines the following set S:

$$S = \{s \in \mathcal{U} \cup \mathcal{L}\colon \mu_{\mathbb{G}}(s, C) = 1\}$$

That is, S is the subset of the set of URIs and literals $\mathcal{U} \cup \mathcal{L}$ such that its elements all match the pattern C. We will call such a set *a proof set* for a pattern C. This is to reflect the fact S proves that the corresponding pattern is frequent. Then, the support of a pattern C over the set $\mathcal{U} \cup \mathcal{L}$ can be defined as:

$$\sigma(C, w) = \sigma(S, w)$$

A *frequent pattern* C over a set $\mathcal{U} \cup \mathcal{L}$ is any such a pattern that $\sigma(C, w) \geq \theta_\sigma$, where θ_σ is a minimal support threshold parameter. A *frequent predicate* p over a set $\mathcal{U} \cup \mathcal{L}$ is any such a predicate that $\sigma(S, w) \geq \theta_\sigma$ for $S = \{s \in \mathcal{U} \cup \mathcal{L}\colon \exists o\,(s, p, o) \in \mathbb{G}\}$.

Again, consider the graph from Listing 1, the set $\mathcal{U}_1$ and the weighting function w_2, as defined above. Let C_1 be a pattern `dbo:Book`. Every of the five books occurs in a triple with the predicate `rdf:type` and the object `dbo:Book`, so the proof set $S_1 = \mathcal{U}_1$, and thus the support $\sigma($`dbo:Book`$, w_2) = 1$. Now consider C_2 to be a pattern `dct:subject` VALUE `dbc:1954_novels`. The proof set S_2 contains only `:The_Fellowship_of_the_Ring` and `:The_Two_Towers`and $\sigma(C_2, w_2) = \frac{2}{6} + \frac{1}{6} = \frac{3}{6}$. If we assume that the minimal support threshold θ_σ is 0.8, then the pattern C_1 `dbo:Book` is frequent, and the pattern C_2 `dct:subject` VALUE `dbc:1954_novels` is not frequent. Following the same line of reasoning, we notice that the property `dbo:illustrator` has the support $\frac{2}{6}$ and is not frequent, while `dct:subject` is frequent with the support 1.

By a depth $\mathfrak{d}(C)$ of a pattern C we understand the number of nested expressions in the pattern:

$$\begin{aligned}
\mathfrak{d}(A) &= 1 \\
\mathfrak{d}(C \text{ AND } D) &= \max\{\mathfrak{d}(C), \mathfrak{d}(D)\} \\
\mathfrak{d}(\{a\}) &= 1 \\
\mathfrak{d}(p \text{ SOME } C) &= 1 + \mathfrak{d}(C) \\
\mathfrak{d}(p \text{ VALUE } b) &= 1 \\
\mathfrak{d}(p \text{ SELF }) &= 1
\end{aligned}$$

Given a set of frequent patterns $\mathcal{P}$, C is the *shallowest frequent pattern* in the set $\mathcal{P}$ if and only if $\mathfrak{d}(C) \leq \mathfrak{d}(D)$ for all $D \in \mathcal{P}$. There may be multiple shallowest frequent patterns in the set $\mathcal{P}$. The set of frequent patterns $\mathcal{P}$ can be infinite, so we mine only a finite subset of it defined by the shallowest frequent patterns.

5.3. Language convention

In the following sections we present a family of algorithms for mining the shallowest frequent patterns, as defined in the previous section. These algorithms refer to a constant parameter θ_σ, which is a minimal support threshold, as discussed in the previous section.

The keyword `yield` used in the pseudo-code snippets should be understand as *add the argument to a temporary set and return this set at the end of the snippet*. It is similar to the `yield` keyword in *Python* programming language.

```
@prefix : <http://dbpedia.org/resource/> .
@prefix dbp: <http://dbpedia.org/property/> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix dbc: <http://dbpedia.org/resource/Category:> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

:The_Hobbit                     dbo:illustrator  :J._R._R._Tolkien ;
                                dbp:language     "English" ;
                                rdf:type         dbo:Book, dbo:CreativeWork ;
                                dct:subject      dbc:1937_novels .
:The_Fellowship_of_the_Ring     dbp:language     "English" ;
                                rdf:type         dbo:Book, dbo:CreativeWork ;
                                dct:subject      dbc:1954_novels, dbc:The_Lord_of_the_Rings ,
                                                 dbc:Novels_adapted_into_plays .
:The_Two_Towers                 dbp:language     "English" ;
                                rdf:type         dbo:Book, dbo:CreativeWork ;
                                dct:subject      dbc:1954_novels ,
                                                 dbc:The_Lord_of_the_Rings  .
:The_Return_of_the_King         dbp:language     "English" ;
                                rdf:type         dbo:Book, dbo:CreativeWork ;
                                dct:subject      dbc:1955_novels ,
                                                 dbc:The_Lord_of_the_Rings  .
:The_Silmarillion               dbo:illustrator  :J._R._R._Tolkien ;
                                dbp:language     "English" ;
                                rdf:type         dbo:Book, dbo:CreativeWork .
                                dct:subject      dbc:1977_books ,
                                                 dbc:The_Silmarillion .
dbc:1937_novels                 rdf:type         skos:Concept .
dbc:1954_novels                 rdf:type         skos:Concept .
dbc:1955_novels                 rdf:type         skos:Concept .
dbc:1977_books                  rdf:type         skos:Concept .
dbc:The_Silmarillion            rdf:type         skos:Concept .
dbc:The_Lord_of_the_Rings       rdf:type         skos:Concept .
dbc:Novels_adapted_into_plays   rdf:type         skos:Concept .
```

Listing 1: A simple RDF graph used in the examples, written in Turtle syntax [6].

Table 1: The definition of the matching function $\mu_{\mathbb{G}}(a, C)$, where a stands for an individual or a literal and C is a pattern.

$$\mu_{\mathbb{G}}(a, DT) = \begin{cases} 1 & a \in \text{value space of } DT \\ 0 & \text{otherwise} \end{cases}$$

$$\mu_{\mathbb{G}}(a, LT[<= M]) = \begin{cases} 1 & a \in \text{value space of } LT \wedge a \leq M \\ 0 & \text{otherwise} \end{cases}$$

$$\mu_{\mathbb{G}}(a, GT[>= m]) = \begin{cases} 1 & a \in \text{value space of } GT \wedge a \geq m \\ 0 & \text{otherwise} \end{cases}$$

$$\mu_{\mathbb{G}}(a, A) = \begin{cases} 1 & (a, \texttt{rdf:type}, A) \in \mathbb{G} \\ 0 & \text{otherwise} \end{cases}$$

$$\mu_{\mathbb{G}}(a, C \text{ AND } D) = \mu_{\mathbb{G}}(a, C) \cdot \mu_{\mathbb{G}}(a, D)$$

$$\mu_{\mathbb{G}}(a, \{b\}) = \begin{cases} 1 & a = b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu_{\mathbb{G}}(a, p \text{ SOME } C) = \begin{cases} 1 & \exists b\, (a, p, b) \in \mathbb{G} \wedge \mu_{\mathbb{G}}(b, C) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mu_{\mathbb{G}}(a, p \text{ VALUE } b) = \begin{cases} 1 & (a, p, b) \in \mathbb{G} \\ 0 & \text{otherwise} \end{cases}$$

$$\mu_{\mathbb{G}}(a, p \text{ SELF }) = \begin{cases} 1 & (a, p, a) \in \mathbb{G} \\ 0 & \text{otherwise} \end{cases}$$

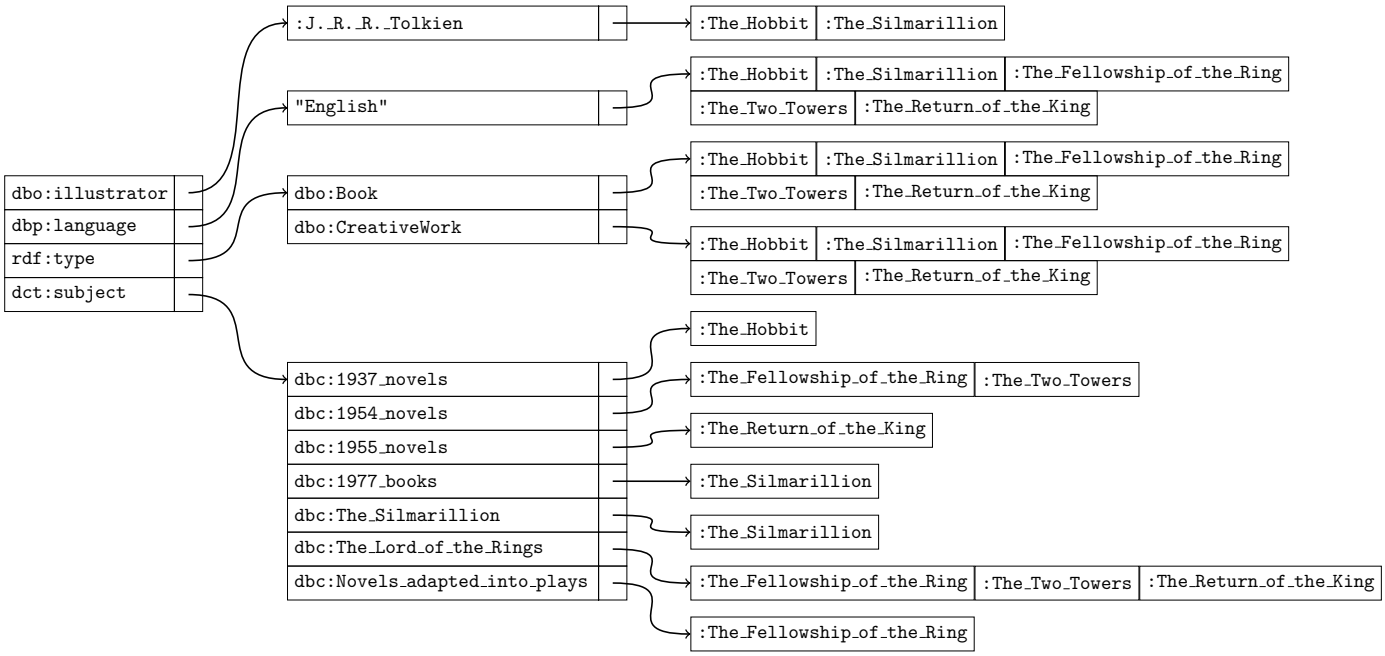


Figure 3: A tree-level index based on the triples from Listing 1 having one of the following subjects: `:The_Hobbit`, `:The_Silmarillion`, `:The_Fellowship_of_the_Ring`, `:The_Two_Towers`, `:The_Return_of_the_King`

Every of the presented mining functions `Mine...` and `SLDM` function returns a pair or a set of pairs: a pattern and its proof set. These proof sets are further used to combine patterns with AND , as it is explained in Section 5.6.

In the examples, we use the set $\mathcal{U}_1$ and the uniform weighting function w_1.

5.4. Mining frequent datatype patterns

We start the presentation of SLDM by presenting an algorithm for mining patterns DT, $LT[<= M]$, $GT[>= m]$. This algorithm requires as an input a set of literals $\mathcal{L}$ and a weighting function w.

Pattern DT. For mining datatype patterns DT, we propose a solution based on the definition of lexical spaces of the datatypes [3, 36]. For every untyped literal $l \in \mathcal{L}$ we check with the grammar of every allowed datatype if l is a valid literal of this datatype. This way, we generate a set D_l of possible datatypes for every literal. For a typed literal $l \in \mathcal{L}$, the set D_l contains only the specified type of l. For a literal with a language tag, according to the specification [18], the correct datatype is `rdf:PlainLiteral`, and so D_l contains only this type.

Patterns GT, LT. Having literals assigned to one or more datatype, we can mine patterns with relations $\geq$ and $\leq$. For a subset $L_{GT} \subseteq \mathcal{L}$ (resp. $L_{LT} \subseteq \mathcal{L}$) containing all literal in $\mathcal{L}$ of a type GT (resp. LT), we look for a minimal (resp. maximal) value m (resp. M) in the set L_{GT} (resp. L_{LT}). The type GT (resp. LT) must be one of the allowed types defined in the Section 3.1. A pattern $GT[>= m]$ (resp. $LT[<= M]$) by definition has support equal to the support of the set L_{GT} (resp. L_{LT}).

The pseudo-code for mining DT, LT and GT patterns is presented in Algorithm 2.

5.5. Mining frequent patterns for a given frequent predicate and index

In this section, we present four algorithms for mining frequent patterns for a given frequent predicate p and with a fixed three-level index I. Patterns that can be mined this way are: A, p VALUE b, r VALUE l, p SELF , $\{b\}$, $\{l\}$.

Pattern A. Following the defintion of the matching function μ, a proof set for a pattern A is defined as $S = \{s \in \mathcal{U} : (s, \texttt{rdf:type}, A) \in \mathbb{G}\}$. Finding all such frequent patterns can be accomplished with the algorithm presented in Algorithm 3. This algorithm can be applied only if the frequent predicate p is `rdf:type`, otherwise the algorithm described in the next paragraph should be used.

Recall the index from Figure 3 and consider p to be the predicate `rdf:type`. There are two values in the second level: `dbo:Book` and `dbo:CreativeWork`. Summing the weights of the URIs in the third level we obtain 1 in both cases, and thus we mine patterns `dbo:Book` and `dbo:CreativeWork`.

Patterns p VALUE b and r VALUE l. Following the definition of the matching function μ, a proof set for a pattern p VALUE b is defined as $S = \{s \in \mathcal{U} : (s, p, b) \in \mathbb{G}\}$. The very same reasoning applies to the patterns r VALUE l. Finding all such frequent patterns can be accomplished with the algorithm presented in Algorithm 4. This algorithm can be applied only if the frequent predicate p is not `rdf:type`.

Recall the index from Figure 3 and let p be the predicate `dbp:language`. There is exactly one possible value in the second level, i.e. `"English"`, and sum of the weights of the URIs

```
function MineDatatype(ℒ, w)
    foreach supported datatype DT
        σ_DT ← 0        // support of the datatype
        S_DT ← ∅                        // proof set
        m_DT ← undefined   // minimal value in the datatype
        M_DT ← undefined   // maximal value in the datatype
    foreach l ∈ ℒ
        D_l ← ∅                    // datatypes for l
        if l has type t
            D_l ← {t}
        else if l has language tag
            D_l ← {rdf:PlainLiteral}
        else
            foreach supported datatype DT
                if l matches the grammar of DT
                    D_l ← D_l ∪ {DT}
        foreach t ∈ D_l
            σ_t ← σ_t + w(l)
            S_DT ← S_DT ∪ {l}
            if t can occur in GT patterns and (m_DT is undefined or l < m_DT)
                m_DT ← l
            if t can occur in LT patterns and (M_DT is undefined or M_DT < l)
                M_DT ← l
    foreach supported datatype DT
        if σ_DT ≥ θ_σ
            if m_DT and M_DT are defined
                // It is impossible for M_DT to be defined and m_DT to be undefined at the same time, as every datatype that can occur in LT patterns, can also occur in GT patterns.
                yield DT AND GT[>= m_DT] AND LT[<= M_DT], S_DT
            else if m_DT is defined
                yield DT AND GT[>= m_DT], S_DT
            else
                yield DT, S_DT
```

Algorithm 2: The algorithm to find all frequent DT, LT and GT patterns given a set of literals $\mathcal{L}$ and a weighting function w

```
function MineType(ℐ, w)
    foreach A ∈ ℐ[rdf:type]
        σ ← 0   // a variable to compute support
        foreach s ∈ ℐ[rdf:type][A]
            σ ← σ + w(s)
        if σ ≥ θ
            yield A, ℐ[rdf:type][A]
```

Algorithm 3: The algorithm to find all frequent patterns A, given a three-level index $\mathcal{I}$ and a weighting function w.

in the third level of the index is 1. Thus, we mine a pattern `dbp:language VALUE "English"`.

Note that Algorithms 3 and 4 differ only in the resulting patterns, not in the idea of operation.

```
function MineValue(ℐ, p, w)
    foreach b ∈ ℐ[p]
        σ ← 0    // variable to compute support
        foreach s ∈ ℐ[p][b]
            σ ← σ + w(s)
        if σ ≥ θ_σ
            yield p VALUE b, ℐ[p][b]
```

Algorithm 4: The algorithm to find all frequent patterns p VALUE b, given a three-level index $\mathcal{I}$, a frequent predicate p, and a weighting function w.

Pattern p SELF . Following the definition of the matching function μ, a proof set for a pattern p SELF is defined as $S = \{s \in \mathcal{U} : (s, p, s) \in \mathbb{G}\}$. Finding all such frequent patterns can be accomplished with the algorithm presented in Algorithm 5.

```
function MineSelf(ℐ, p, w)
    σ ← 0        // variable to compute support
    S ← ∅                                // proof set
    foreach s ∈ ℐ[p]
        if s ∈ ℐ[p][s]
            σ ← σ + w(s)
            S ← S ∪ {s}
    if σ ≥ θ
        return p SELF , S
```

Algorithm 5: The algorithm to find all frequent patterns p SELF , given a three-level index I, a frequent predicate p and a weighting function w.

Patterns $\{b\}$ and $\{l\}$. Following the definition of the matching function μ, a proof set for a pattern $\{b\}$ (resp. $\{l\}$) is defined as $S = \{b\} \cap \mathcal{U}$ (resp. $S = \{l\} \cap \mathcal{L}$). Finding all such frequent patterns can be accomplished with the algorithm presented in Algorithm 6.

```
function MineEnum(𝒰, ℒ, w)
    foreach b ∈ 𝒰 ∪ ℒ
        if w(b) ≥ θ_σ
            yield {b}, {b}
```

Algorithm 6: The algorithm to find all frequent patterns $\{b\}$, given a set of URIs $\mathcal{U}$, a set of literals $\mathcal{L}$ and a weighting function w.

5.6. Mining frequent conjunctions

We build patterns that use conjunction, i.e. AND , by combining patterns with identical proof sets mined earlier for given sets of URIs and literals. It can be efficiently accomplished by using

a hash map. Following naming convention from frequent pattern mining, we call such a conjunction a *closed conjunction*, because it can not be extended (i.e. no other frequent pattern can be added to the conjunction) without shrinking the proof set and thus decreasing the support. The corresponding algorithm is presented in Algorithm 7. The algorithm requires as an input a set of frequent patterns and their proof sets $\mathcal{P}$, and outputs all closed conjunctions that can be build from this set along with their proof sets.

In Section 5.5 we mined the following three patterns: `dbo:CreativeWork`, `dbo:Book` and `dbp:language VALUE "English"`. All of them have exactly the same proof set, consisting of all five books, so we can safely join them into a single pattern: `dbo:CreativeWork AND dbo:Book AND dbp:language VALUE "English"`

function `MineClosedConjunctions`($\mathcal{P}$)
  $G \leftarrow \emptyset$ `// a map from proof sets to patterns`
  foreach $P, S \in \mathcal{P}$
    $G[S] \leftarrow G[S] \cup \{P\}$
  foreach $S \in G$
    $n \leftarrow |G[S]|$ `// a number of patterns in` $G[S]$
    yield $G[S][0]$ AND $G[S][1]$ AND ... AND $G[S][n], S$

Algorithm 7: The algorithm finding all closed conjunctions from a set of patterns $\mathcal{P}$.

5.7. The core algorithm

In Section 4 we discussed how to build a three-level index for a given set of URIs. In Sections 5.4–5.6 we proposed a family of algorithms for mining frequent patterns, all except these using SOME. Now, we combine these results into Swift Linked Data Miner: we show how to remove some unnecessary information from the index, apply the mining algorithms defined earlier and use SLDM recursively to mine SOME patterns.

The core of SLDM is presented in Algorithm 8. Its input consists of a set of URIs $\mathcal{U}$, a set of literals $\mathcal{L}$ and a weighting function w. It outputs a set of shallowest frequent patterns for the set $\mathcal{U} \cup \mathcal{L}$. To decide if a pattern is frequent, the algorithm also uses a minimal support threshold parameter θ_σ.

To ensure that the algorithm ends, we add a maximal depth threshold parameter $\theta_\mathfrak{d}$, and require that for all patterns C, the pattern depth $\mathfrak{d}(C)$ does not exceed the parameter $\theta_\mathfrak{d}$. Recall that the pattern depth increases only with SOME constructor, so this requirement is equivalent with limiting maximal recursion level. To understand how a pattern of an infinite depth could be mined, consider an RDF graph with the following two triples: $\{(a, p, b), (b, q, a)\}$, an uniform weighting function $w(a) = w(b) = \frac{1}{2}$ and a minimal support threshold $\frac{1}{4}$. Starting from a, SLDM follows to b using the predicate p, then goes back from b to a using the predicate q and again explores a, effectively getting stuck in an infinite loop while constructing a pattern p SOME (q SOME (p SOME ...)). Observe that every time the minimal support threshold is exceeded and so there is no stopping condition if the maximal depth threshold is not considered.

The algorithm starts by mining patterns that does not require obtaining any triples from the SPARQL endpoint, as both `MineEnum` and `MineDatatype` do not use an index. Then, it obtains new triples and builds a three-level index, following the idea described in Section 4. To store in the index only frequent predicates it uses a pruning function, that is presented in Algorithm 9. For the index presented in Figure 3, it means that the predicate `dbo:illustrator` is removed, because sum of weights for `:The_Hobbit` and `:The_Silmarillion` does not exceed the threshold θ_σ, and so the predicate is not frequent.

Later, SLDM iterates over each predicate in the index and uses functions defined in Section 5.5. If for a given frequent predicate p none of these functions yield a frequent pattern, the algorithm tries to mine deeper patterns with SOME . Such an approach is to ensure that SLDM mines only the shallowest frequent patterns. For this, a function `MineSome` defined in Algorithm 10 is used. Finally, obtained patterns are combined with AND operator using `MineClosedConjunctions` function, defined in Section 5.6.

function `SLDM`($\mathcal{U}$, $\mathcal{L}$, w)
  $\mathcal{P} \leftarrow$ `MineEnum`$(\mathcal{U}, \mathcal{L}, w) \cup$ `MineDatatype`$(\mathcal{L}, w)$
  Obtain a set of triples $\mathcal{T}$, such that their subjects are in $\mathcal{U}$
  $\mathcal{I} \leftarrow$ `BuildIndex`$(\mathcal{T})$
  $\mathcal{I} \leftarrow$ `PruneIndex`$(\mathcal{I}, w)$
  foreach $p \in \mathcal{I}$ `// Iterate over all frequent predicates`
    $\mathcal{P}_p \leftarrow \emptyset$
    if p *is* `rdf:type`
      $\mathcal{P}_p \leftarrow$ `MineType`$(\mathcal{I}, w)$
    else
      $\mathcal{P}_p \leftarrow$ `MineValue`$(\mathcal{I}, p, w) \cup$ `MineSelf`$(\mathcal{I}, p, w)$
    if $\mathcal{P}_p \equiv \emptyset$ *and the recursion level does not exceed* $\theta_\mathfrak{d}$
      $\mathcal{P}_p \leftarrow$ `MineSome`$(\mathcal{U}, \mathcal{L}, \mathcal{I}, w, p)$
    $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{P}_p$
  return `MineClosedConjunctions`$(\mathcal{P})$

Algorithm 8: Swift Linked Data Miner algorithm.

Patterns p SOME C and r SOME DT. Finding such frequent patterns requires a recursive call to SLDM. First we must compute a new set of URIs $\mathcal{U}_{new}$, a new set of literals $\mathcal{L}_{new}$ and a new weighting function w_{new}. Then, the recursive call is made, using $\mathcal{U}_{new}$, $\mathcal{L}_{new}$ and w_{new} as the input. Finally, after the recursive call returns, the generated frequent patterns are prefixed with p SOME , to obtain valid frequent patterns in current recursion level. The corresponding algorithm is presented in Algorithm 10.

p is a frequent predicate specified as a parameter to the `MineSome`, $\mathcal{U}_{new}$ is the set of all URIs occurring in the second level of the index, and $\mathcal{L}_{new}$ is the set of all literals occurring in the second level of the index. To compute w_{new} for a given object $o \in \mathcal{U}_{new} \cup \mathcal{L}_{new}$, we sum over all subjects s such that triple (s, p, o) is in the index, i.e. we sum over $\mathcal{I}[p][o]$. Weight

```
function PruneIndex(I, w)
    foreach p ∈ I
        S ← ∅  // A proof set for the predicate
        foreach o ∈ I[p]
            S ← S ∪ I[p][o]
        σ ← ∅    // A support for the predicate
        foreach s ∈ S
            σ ← σ + w(s)
        if σ < θ_σ
            // p is infrequent, we can drop the
               triples with it
            delete I[p]
    return I
```

Algorithm 9: The algorithm pruning a three-level index $\mathcal{I}$ to store only frequent predicates according to a weighting function w. The resulting index contains only frequent predicates.

```
function MineSome(U, L, I, p, w)
    U_new ← ∅
    L_new ← ∅
    foreach n ∈ U ∪ L
        den[n] ← 0
    // First, we compute all denominators
    foreach o ∈ I[p]
        if o is an URI
            U_new ← U_new ∪ {o}
        else
            L_new ← L_new ∪ {o}
        foreach s ∈ I[p][o]
            den[s] ← den[s] + 1
    // Then, the actual new weights w_new are
       computed
    foreach o ∈ I[p]
        w_new[o] ← 0 // we represent w_new as a map
        foreach s ∈ I[p][o]
            w_new[o] ← w_new[o] + w(s)/den[s]
    // Recursive call to the SLDM
    P ← SLDM(U_new, L_new, w_new)
    // Final step, prefixing the patterns from
       recursion
    foreach C, S ∈ P
        S' ← ∅
        foreach s ∈ S
            S' ← S' ∪ I[p][s]
        yield p SOME C, S'
```

Algorithm 10: The algorithm finding all frequent patterns p SOME given a set of URIs $\mathcal{U}$, a set of literals $\mathcal{L}$, a three-level index I, a frequent predicate p and a weighting function w.

Table 2: The weights computed for the recursive call in the example.

URI	weight
dbc:1937_novels	$\frac{6}{30}$
dbc:1954_novels	$\frac{5}{30}$
dbc:1955_novels	$\frac{3}{30}$
dbc:1977_books	$\frac{3}{30}$
dbc:The_Silmarillion	$\frac{3}{30}$
dbc:The_Lord_of_the_Rings	$\frac{8}{30}$
dbc:Novels_adapted_into_plays	$\frac{2}{30}$

of every subject s is equally divided over all objects x, such that a triple (s, p, x) is in the index $\mathcal{I}$.

$$w_{new}(o) = \sum_{s \in \mathcal{I}[p][o]} \frac{w(s)}{|\{x\colon (s,p,x) \in \mathcal{I}\}|}$$

Recall the index in Figure 3 and let p be dct:subject. In this case, the set $\mathcal{P}_p$ in the SLDM function is empty, and thus function MineSome is called. Recall that we use the set $\mathcal{U}_1$ and the weighting function w_1. The set $\mathcal{U}_{new}$ will consist of all values appearing in the second level of the index for the property dct:subject: dbc:1937_novels, dbc:1954_novels, dbc:1955_novels, dbc:1977_books, dbc:The_Silmarillion, dbc:The_Lord_of_the_Rings, dbc:Novels_adapted_into_plays. We also need to compute new weights w_{new} for these URIs. We do so by equally dividing the previous weights. dbc:1937_novels occurs for exactly one URI from the original set, i.e. :The_Hobbit, and there are no other triples with :The_Hobbit and dct:subject, so dbc:1937_novels receives all the weight of :The_Hobbit, i.e. $\frac{1}{5}$. dbc:Novels_adapted_into_plays also occurs exactly once, but for an URI that has three possible values for dct:subject, and so it receives weight $\frac{1}{3} \cdot \frac{1}{5} = \frac{1}{15}$. dbc:1954_novels occurs for two URIs: :The_Fellowship_of_the_Ring and :The_Two_Towers. It is one of three possible values for :The_Fellowship_of_the_Ring, and one of two possible values for :The_Two_Towers, so dbc:1954_novels is assigned weight $\frac{1}{3} \cdot \frac{1}{5} + \frac{1}{2} \cdot \frac{1}{5} = \frac{1}{6}$. dbc:1955_novels, dbc:1977_books and dbc:The_Silmarillion occur for exactly one URI each, and all three are one of two values for dct:subject for their respective URIs. They both receive half of the respective weight, that is $\frac{1}{2} \cdot \frac{1}{5} = \frac{1}{10}$. Finally, dbc:The_Lord_of_the_Rings occurs for three URIs and is one of two values for two of them, and one of three values for one of them, so the weight is $\frac{1}{3} \cdot \frac{1}{5} + \frac{1}{2} \cdot \frac{1}{5} + \frac{1}{2} \cdot \frac{1}{5} = \frac{4}{15}$. The new weights are presented in Table 2.

Now we perform the recursive call. We retrieve a new set of triples, having as subjects the URIs listed in Table 2. The corresponding three-level index is presented in Figure 4. Running the algorithm, we obtain a pattern skos:Concept. After returning from the recursion, we prefix it and obtain a pattern dct:subject SOME skos:Concept.

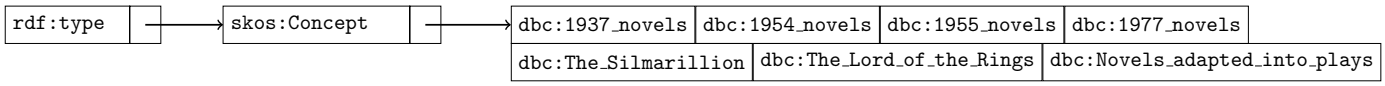


Figure 4: An index generated in the recursive call of the algorithm in the example.

5.8. Initial call to the algorithm

In the initial call to the SLDM, made by the user, we assume a simpler input, consisting only of a set of URIs $\mathcal{U}$. We assume an empty set of literals, because the goal of the user is to find patterns for given URIs described by an RDF graph. We also assume that a weighting function is uniform. Finally, if the user already has some ontology O, it may be desirable to remove the axioms that are already are entailed by the ontology. These assumptions are combined and presented in Algorithm 11.

Data: $\mathcal{U}$ a set of URIs
Result: a set of frequent patterns
$w \leftarrow \emptyset$ `// a map representing an initial weighting function`
$n \leftarrow |\mathcal{U}|$
foreach $s \in \mathcal{U}$
  $w[s] \leftarrow \frac{1}{n}$
foreach $C, S \in \texttt{SLDM}(\mathcal{U}, \emptyset, w)$
  if *C is not entailed by O*
    yield C

Algorithm 11: The suggested way of calling the SLDM algorithm by the user.

6. Complexity analysis of the algorithm

When computing the worst-case memory complexity two factors that must be taken into account: the size of the indexes and the size of the sets of the mined patterns along with the corresponding proof sets. Denote by n the number of triples present in the RDF graph. At any given point of the execution of the algorithm, the number of the recursive calls of SLDM is not greater than $\theta_\mathfrak{b}$, that is there are at most $\theta_\mathfrak{b}$ three-level indexes in the memory. Each of the indexes contain at most n triples, thus the overall memory complexity of the indexes is $O(n^{\theta_\mathfrak{b}})$. For each triple present in the index, SLDM generates at most one pattern, thus the number of patterns generated is no greater than n. The size of each proof set is bounded by the number of distinct URIs in the set $\mathcal{U}$, which is also no greater than n. Thus, in the worst-case, we arrive at n patterns, having the support set of n URIs each and the overall memory complexity of the mined patterns is $O(n^2)$. The overall memory complexity of SLDM is $O(n^{\max\{\theta_\mathfrak{b},2\}})$.

The reasoning behind the worst-case time complexity is similar: SLDM iterates over a three-level index, which is at most size n. The sets $\mathcal{U}$ and $\mathcal{L}$ are also always at most of size n, and the access to hash-based maps is $O(1)$, so the upper bound on the overall time complexity of a single call to SLDM is $O(n)$. Again, we must consider the recursive calls up to $\theta_\mathfrak{b}$ deep, each of them able to perform up to n recursive calls. This way, we obtain a geometric series $n + n^2 + \ldots + n^{\theta_\mathfrak{b}}$ and conclude that the upper bound for SLDM worst-case time complexity is $O(n^{\theta_\mathfrak{b}})$. This analysis does not include the polynomial complexity of checking if an axioms is already entailed by the ontology.

7. Relation to RDF Data Shapes

There are applications where using OWL ontologies is not desirable or sufficient. For example, one may want to avoid the complexity of reasoning procedures and only verify if a given RDF graph conforms to a model. To address such use cases, RDF Data Shapes were proposed [45]. Their formal representation can be expressed in Shapes Constraint Language (SHACL) [24]. Throughout this section we use prefix `sh:` for the namespace `http://www.w3.org/ns/shacl#`, as defined in [24].

An OWL 2 EL superclass expression generated by SLDM can be transformed to SHACL. Recall that SLDM results are based on on-line analysis of an RDF graph, so such a transformation is sanctioned. We start by transforming the examples used in the previous section, and later we give a full definition of the transformation.

7.1. Example

In Section 5.5 we mined three patterns: `dbo:CreativeWork`, `dbo:Book` and `dbp:language VALUE "English"`. Each of them was mined because there were triples with a fixed predicate and a fixed object in the RDF graph, so it is natural to use this knowledge to map the patterns to the RDF shapes. The obtained shapes are presented in Listing 2.

In Section 5.6 we joined the abovementioned patterns with AND operator, obtaining the pattern `dbo:CreativeWork AND dbo:Book AND dbp:language VALUE "English"`. The corresponding shape is presented in Listing 3.

Finally, in Section 5.7, we mined the pattern `dct:subject SOME skos:Concept`. This was performed by first mining the pattern `skos:Concept` and then adding the prefix to it. Therefore, we should follow the same principle during the transformation and refeer to some other shape in the shape corresponding to the pattern `dct:subject SOME skos:Concept`. The effect of the transformation is presented in Listing 4.

7.2. Transformation

Recall the definition of *OWL 2 EL superclass expression C* from Section 3.1. This is the grammar, which all patterns mined by SLDM follow. Below, we define a recursive transformation from C to SHACL. This transformation can be done in post-processing, i.e. first SLDM is used as defined in Section 5 or it can be incorporated into the algorithms, changing them into algorithms directly mining RDF Data Shapes. The transformation is defined as two functions τ and τ_d, that gets respectively

```
:shape1 rdf:type sh:Shape ;
        sh:property [
                sh:predicate rdf:type ;
                sh:hasValue dbo:CreativeWork ;
        ] .

:shape2 rdf:type sh:Shape ;
        sh:property [
                sh:predicate rdf:type ;
                sh:hasValue dbo:Book ;
        ] .

:shape3 rdf:type sh:Shape ;
        sh:property [
                sh:predicate dbp:language ;
                sh:hasValue "English" ;
        ] .
```

Listing 2: RDF Data Shapes expressed in SHACL, that correspond to the patterns `dbo:CreativeWork`, `dbo:Book` and `dbp:language` VALUE `"English"` mined in Section 5.5.

```
:shape4 rdf:type  sh:Shape ;
        sh:constraint [
                sh:and(:shape1 :shape2 :shape3)
        ] .
```

Listing 3: An RDF Data Shape expressed in SHACL, that corresponds to the pattern `dbo:CreativeWork` AND `dbo:Book` AND `dbp:language` VALUE `"English"` mined in Section 5.6. The shapes `:shape1, :shape2, :shape3` are defined in Listing 2.

```
:shape5 rdf:type sh:Shape ;
        sh:property [
                sh:predicate dct:subject ;
                sh:qualifiedValueShape [
                    sh:property [
                        sh:predicate rdf:type ;
                        sh:hasValue skos:Concept ;
                ]] ;
                sh:qualifiedMinCount 1 ;
        ] .
```

Listing 4: An RDF Data Shape expressed in SHACL, corresponding to the pattern `dct:subject` SOME `skos:Concept` mined in Section 5.7.

a class expression C or a data range R as an argument and returns the corresponding SHACL expression. The functions are fully defined in Table 3 and below we give an explanation of this definition.

We start by considering Algorithm 3. The algorithm yields the pattern A any time there is a large enough number of triples of form $(\cdot, \texttt{rdf:type}, A)$ in the graph. The corresponding RDF Data Shape must take into account all three properties: (a) a single triple, (b) with `rdf:type` as the predicate and (c) with A as the object. We may thus define the function τ such that it yields such an expression every time its argument is a named class A, as specified in the first row of Table 3.

Algorithm 4 works in a very similar fashion, looking for a large number of triples of form $(\cdot, p, b)$ for fixed values of p and b and yielding the pattern p VALUE b in case of success. The corresponding RDF Data Shape must look for a very similar properties as before: a single triple, with p as the predicate and b as the object. The corresponding SHACL expressions are specified as rows 2 and 3 in Table 3.

Algorithm 5 operates on a similar basis as the previous two, and counts the number of triples matching the pattern (s, p, s) for a fixed value of p. To the best of our knowledge, such a pattern can not be expressed using core constraints defined in [24]. Fortunately, SHACL is by design extensible with SPARQL queries. The query must return RDF nodes that violate the constrain, i.e. nodes s such that a triple (s, p, s) is not present in the graph. The desired behavior can be achieved using SPARQL `sameTerm` function and FILTER NOT EXISTS clause. The complete query, which returns a single binding for each triple violating the constraint, written in accordance with the requirements of SHACL specification and wrapped in appropriate RDF expressions, is presented in row 6 of Table 3.

Algorithm 7 does not operate on triples directly, but rather combines the results of the recursive calls of SLDM to construct conjunctions. We can follow the very same reasoning and combine recursively transform the operands of the conjunction to SHACL and then combine them with SHACL `sh:and` operator. For an OWL 2 EL class expression C_1 AND C_2 AND ..., each of the operands $C_1, C_2, \ldots$ must be first translated to SHACL using the τ function: $\tau(C_1), \tau(C_2), \ldots$ and only then they may be combined into a complex SHACL expression specified as rows 4 and 10 in Table 3.

Algorithm 6 also does not operate on triples, but only on the weighting function. Every time it yields a pattern $\{a\}$ (resp. $\{l\}$) it means that a given URI a (resp. a literal l) is frequent w.r.t. to the function. The corresponding SHACL expression must state that this URI (resp. literal) is expected to occur, and this can be achieved using SHACL `sh:in` operator, as stated in rows 5 and 11 of Table 3.

The existential quantification can not be directly expressed in SHACL, however such an expression is semantically equivalent to a minimal cardinality restriction with caridinality 1 (for the proof, compare the semantics of *ObjectMinCardinality* for $n = 1$ and *ObjectSomeValuesFrom* in [35]). Algorithm 10 uses recursion to first discover frequent patterns in a new subset of URIs or literals, and then prefixes them with p SOME . The same principle can be applied during the conversion to SHACL: first,

the nested pattern must be converted using an appropriate function: τ in case of a class expression C or τ_d in case of a data range R, obtaining an RDF node $\tau(C)$ or, respectively, $\tau_D(R)$. Then, the result is wrapped in a SHACL expression specifying the predicate p (using `sh:predicate`) and the minimal cardinality of 1 (`sh:qualifiedMinCount 1`). The resulting transformation is presented in rows 7 (for object properties) and 8 (for data properties) of Table 3.

Finally, Algorithm 2 counts triples having a literal of a given datatype DT and finds appropriate minimal and maximal values. The resulting RDF Data Shapes should thus verify if the value in question is of a given datatype and if it fits between the minimal and maximal value. The expected datatype of a value can be expressed in SHACL using `sh:datatype` and the minimal and maximal value, using, respectively, `sh:minInclusive` and `sh:maxInclusive`. The corresponding SHACL expressions are presented as rows 9, 12 and 13 of Table 3.

8. *Protégé* plugin

In Figure 5 we present a screenshot of the *Protégé* plugin, which enables the user to mine new superclass expressions directly from *Protégé*. The user must just select a class in the class hierarchy view, enter the address of a SPARQL endpoint she wants to use and click `Run`. The axioms are displayed right after they are mined, not after the whole mining ends, so the user does not need to waste the time waiting. Mining can be stopped at any time by clicking `Stop` button. On the `Expert` tab (Figure 6), it is possible to configure SLDM parameters, such as the minimal support threshold or if sampling is to be used. One can also specify predicates that should be ignored during the mining. For example, one may want to ignore provenance-related predicates (e.g. `dbo:wikiPageID`), as not directly related to the semantics of the mined class.

The plugin is written in *Java*, using the OWL API[5] [19] and *Apache Jena*[6] [30]. The source code of the plugin is available in *Git* repository `https://bitbucket.org/jpotoniec/sldm` along with the compilation and installation instruction. There is also a link to a precompiled version of the plugin.

9. Experimental evaluation

9.1. Extending the DBpedia ontology

This section presents all the steps that were undertaken in order to prepare and conduct an experiment on a crowdsourcing platform *CrowdFlower*[7]. Our aim was to answer the following research question: can SLDM mine new, meaningful axioms, that can be added to the ontology. To answer the question, we used *DBpedia* 2015-04 with the *DBpedia* ontology and we followed the experimental protocol described below:

1. We conducted exploratory data analysis to select a set of classes.
2. For the selected classes, we used SLDM to generate superclass expressions, and used them to obtain a set of SubClassOf: axioms for a selected class, with the class in the left-hand side, and an expression in the right-hand side of an axiom.
3. We translated the generated axioms into natural language sentences.
4. We generated test questions to ensure that participants of the experiment are paying attention to their tasks.
5. These sentences were then posed to *CrowdFlower* for verification by the contributors.
6. We collected and analyzed the results of the verification.

In the following sections, the details of the experimental protocol are explained.

9.1.1. Exploratory data analysis

To select a set of classes from the *DBpedia* ontology, what would allow us to conduct a high quality, statistically reliable experimental evaluation, we performed exploratory data analysis. For every class in the ontology, we computed the following characteristics using *DBpedia* 2015-04:

1. the number of class instances,
2. the number of different triples, for which the subject belongs to the class,
3. the number of different predicates, for which there exists a triple in the dataset with a given predicate, and the subject belonging to the class,
4. the depth of the class in the subsumption hierarchy in the *DBpedia* ontology (the shortest path from the root of the hierarchy `owl:Thing` to the class).

Histograms of the obtained values are presented in Figures 7a–7d. On the basis of the histograms, we chose a set of criteria that the selected classes should fulfill.

To provide enough statistical support, we chose classes with more that 1000 instances and every instance occurring as a subject on average in more than 50 triples. To avoid generating a very large number of axioms, which would increase the costs of the crowdsourcing verification, we decided to keep the number of different predicates in range from 20 to 35. Finally we decided on selecting classes with the depth at least 3, and in such a manner that all selected classes should have pairwise different parents and at least three different grandparents.

We selected 5 classes, which meet all the aforementioned criteria: `Journalist`, `ProgrammingLanguage`, `Book`, `MusicGenre`, `Crater` together with their ancestors: `Agent`, `Person`, `Work`, `Software`, `WrittenWork`, `TopicalConcept`, `Genre`, `Place`, `NaturalPlace`. Full hierarchy is presented in Figure 8. For each of these 14 classes, we used SLDM to generate two sets of axioms, the first one using the minimal support threshold $\theta_\sigma = .5$ and the second one with $\theta_\sigma = .8$. The obtained axioms are available in the *Git* repository `https://bitbucket.org/jpotoniec/sldm`, in the subfolder `CF_source`.

[5] `http://owlapi.sourceforge.net/`
[6] `https://jena.apache.org/`
[7] `https://www.crowdflower.com/`

Table 3: A formal definition of functions τ and τ_d, i.e. a transformation from a SLDM pattern expressed as OWL 2 EL class expression to an RDF Data Shape expressed in SHACL. Each of the top nodes in the transformation should be defined to be a `sh:Shape`, i.e. there should be an additional triple `... rdf:type sh:Shape`. We omitted them to make the table more readable.

#	C	$\tau(C)$
1	A	`[sh:property [sh:predicate rdf:type; sh:hasValue` A `]]`
2	p VALUE a	`[sh:property [sh:predicate` p`; sh:hasValue` a`;]]`
3	r VALUE l	`[sh:property [sh:predicate` r`; sh:hasValue` l`;]]`
4	C_1 AND C_2	`[sh:constraint [sh:and(`$\tau(C_1)$ $\tau(C_2)$`) ]]`
5	$\{a\}$	`[sh:constraint [sh:in (`a`)]]`
6	p SELF	`[sh:constraint [rdf:type sh:SPARQLConstraint; sh:sparql "SELECT $this ($this AS ?subject) (`p` AS ?predicate) (?value AS ?object) WHERE { $this` p` ?value . FILTER NOT EXISTS (sameTerm($this, ?value)) }" ]]`
7	p SOME C	`[sh:property [sh:predicate` p`; sh:qualifiedMinCount 1; sh:qualifiedValueShape` $\tau(C)$ `]]`
8	r SOME R	`[sh:property [sh:predicate` p`; sh:qualifiedMinCount 1; sh:qualifiedValueShape` $\tau_d(R)$ `]]`
#	R	$\tau_d(R)$
9	DT	`[sh:constraint [sh:datatype` DT`;]]`
10	R_1 AND R_2	`[sh:constraint [sh:and(`$\tau_d(R_1)$ $\tau_d(R_2)$`) ]]`
11	$\{l\}$	`[sh:constraint [sh:in (`l`)]]`
12	LT[<= l]	`[sh:constraint [sh:datatype` LT`; sh:minInclusive` l`]]`
13	GT[>= l]	`[sh:constraint [sh:datatype` GT`; sh:maxInclusive` l`]]`

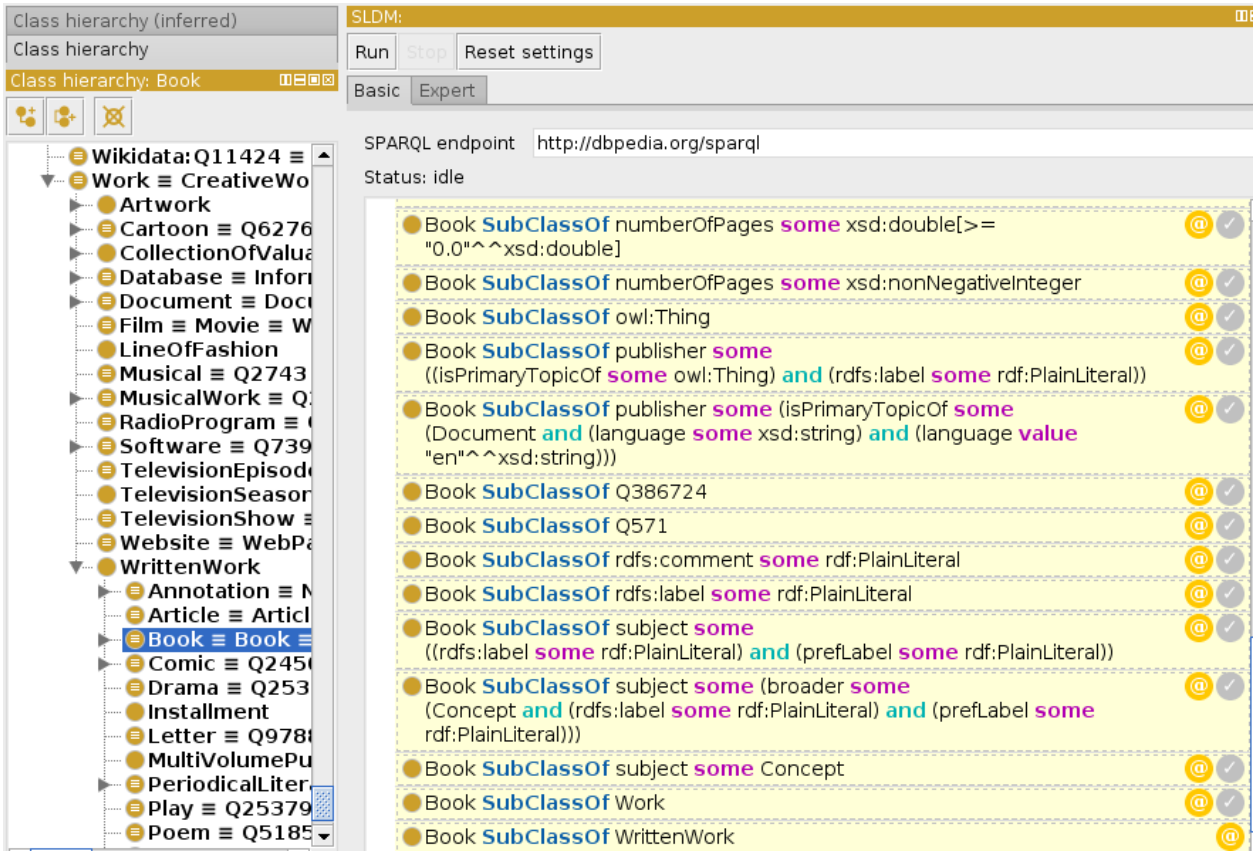


Figure 5: The *Protégé* plugin with some axioms mined for the class `dbo:Book` using the *DBpedia* endpoint. On the right-hand side of a mined axiom there are two buttons: the first one (with @ symbol) to display additional information about the axiom, e.g. support, and the other one (with ✓symbol) to add the axiom to the ontology. The last axiom in the list has only the first button, because it already is asserted in the ontology.

Figure 6: The `Expert` tab of the *Protégé* plugin. The user can configure here a minimal support threshold, a maximal depth of the axioms, sampling parameters and predicates ignored during the mining (using regular expressions).

To get some insight into novelty of the mined axioms, we used HermiT reasoner[8] [33] and for each of the mined set of axioms we calculated how many of them were already logically entailed by the DBpedia ontology and how many of them were logically entailed by the DBpedia ontology enriched with the mined axioms for the superclass. The detailed statistics are presented in Table 4. For example, for the class Book and the minimal support threshold $\theta_\sigma = 0.8$, 35 axioms were mined by SLDM, out of which 7 were logically entailed by the ontology and 15 were logically entailed by the ontology with the mined axioms for the class WrittenWork asserted. It must be noted that in all the cases, SLDM was able to discover more than it was already present in the ontology.

9.1.2. Translation of the axioms to natural language

Ontological axioms expressed in OWL (e.g. using Turtle) cannot be easily understood by English speakers that are not familiar with the Semantic Web technologies. Therefore we proposed a procedure of translation of OWL axioms to English. We wanted to use a simple variant of the language, so we decided to choose Attempto Controlled English [23]. It is a controlled version of normal English language that involves advantages of formal representation (well defined syntax, possibility of automatic processing) and natural language (expressiveness and ease of understanding) [12]. Each person that knows basics of English should be able to understand a translated sentence without any knowledge about its formal representation. Usage of controlled language has one other, very important feature: the translation is fully reversible, so we do not lose any information.

We decided to create a whole translator on our own. The reason for that was the fact that existing tools (e.g. *OWL Verbalizer*[9] [22]) are restricted and work only for some of our examples.

In the OWL axioms generated by SLDM, we identified a set of structural templates and for every template we provided a corresponding template in English. The URIs in the axioms were replaced by their corresponding labels during the translation. The core idea of the translation tool is to analyze an axiom level by level and match it to the templates. The output is a set of simple sentences, that represent more and more specific parts of constrains. Sample axiom, that pertains `dbo:Journalist` class

```
dbo:Journalist SubClassOf: dbo:nationality
        some (dbo:governmentType some owl:Thing
        and dbo:leader some owl:Thing)
```

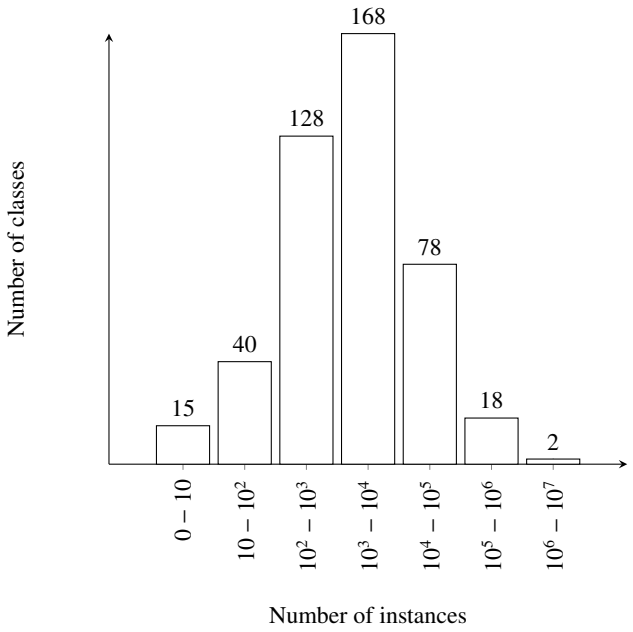
can be translated into sentences *Every journalist has nationality. Nationality has government type. Nationality has leader.* During the translation, we also performed some pruning, in order to make the final sentences more readable and limit the costs of the experiment. We removed axioms that contained concepts which are characteristic for internal structure of *DBpedia* or act as metadata, e.g. predicate `dbp:hasPhotoCollection`. We reason that such axioms are very hard to understand for a non-expert, and thus can not be efficiently verified by the contributors of a crowdsourcing platform. We also removed axioms containing namespaces from other Linked Data sets, e.g. *Wikidata* namespace, in order to decrease number of axioms to verify, and avoid displaying numerical URIs to the users, e.g. we removed the axiom `dbo:Book` SubClassOf: `wikidata:Q1930187`[10].

After application of the translation tool to the axioms generated by SLDM, we obtained a set of sentences. Each of these sentences was then used as a base to form a question. A question consists of: a sentence, which is to be verified; a set of

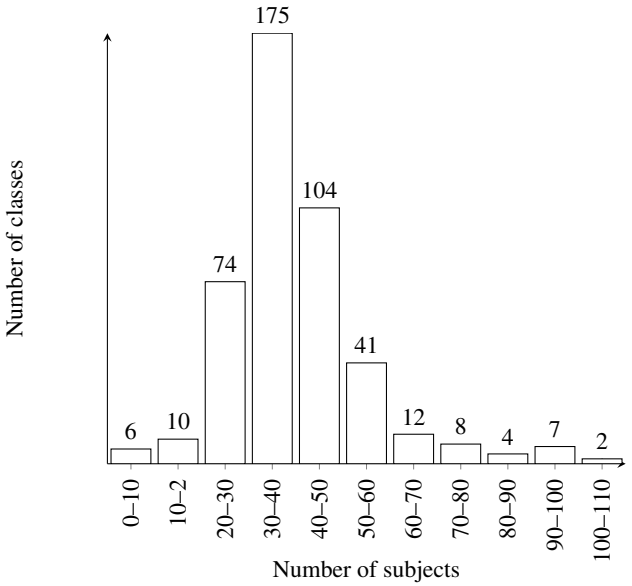
[8] `http://www.hermit-reasoner.com/`

[9] `https://github.com/Kaljurand/owl-verbalizer`

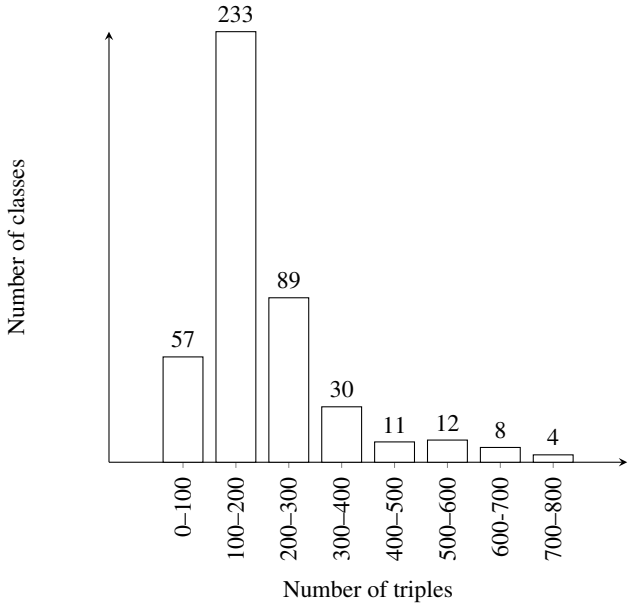
[10] The prefix `wikidata:` corresponds to `http://www.wikidata.org/entity/`. The complete list of removed axioms is available in the *Git* repository, in the file `CF_source/removed_axioms.txt`.



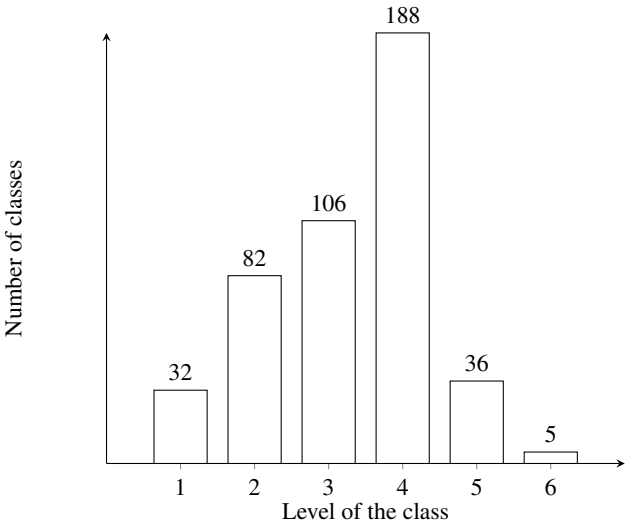
(a) A histogram of number of class instances in *DBpedia* for the classes from the *DBpedia* ontology.



(b) A histogram of average number of different predicates, for which there is a triple in *DBpedia* with the subject from a given class.



(c) A histogram of average number of triples in *DBpedia* for which the subject belongs to a given class.



(d) Level of classes in the subsumption hierarchy in the *DBpedia* ontology.

Figure 7: Histograms for characteristics used to select the set of classes from the *DBpedia* ontology to perform the experiment on.

Table 4: Number of axioms mined for each of the classes listed in Figure 8 and each of the two minimal support thresholds θ_σ. The column *overall* presents the overall number of mined axioms for a given class, the column *in the ontology* presents how many of them were logically entailed by the DBpedia ontology and, finally, the column *in the superclass* presents the same, but also asserting all the mined axioms for the superclass.

class	number of mined axioms $\theta_\sigma = 0.8$			number of mined axioms $\theta_\sigma = 0.5$		
	overall	in the ontology	in the superclass	overall	in the ontology	in the superclass
Agent	9	3	-	22	3	-
Person	9	7	9	22	7	20
Journalist	109	8	10	131	8	21
Work	11	3	-	14	3	-
Software	17	4	12	45	4	14
ProgrammingLanguage	16	6	12	28	6	13
WrittenWork	14	4	12	63	4	15
Book	35	7	15	108	7	29
TopicalConcept	10	3	-	13	3	-
Genre	13	4	11	27	4	13
MusicGenre	13	5	13	27	5	27
Place	21	3	-	186	5	-
NaturalPlace	13	4	12	24	4	24
Crater	12	5	11	25	5	12

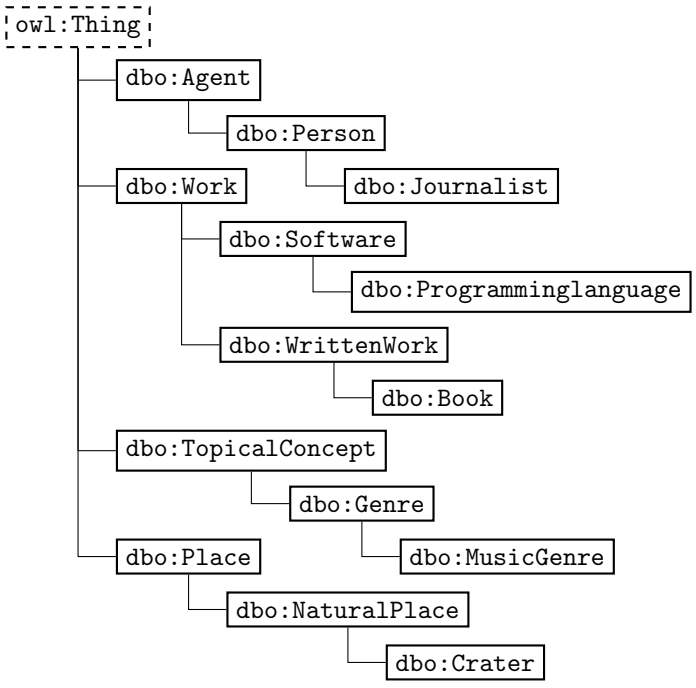


Figure 8: The set of classes from the *DBpedia* ontology used in the experiment. `owl:Thing` is depicted in the picture only for reference and was not used in the experiment.

three allowed answers, from which only one is to be selected: *Yes*, *No*, *I don't know*; an optional field to explain why a particular answer was selected.

9.1.3. Test questions

The quality of the results achieved from crowdsourcing experiment can be significantly improved by introducing test questions [31]. The right answer to these questions is known before the experiment. They are used to check reliability of crowdsourcing platform contributors. They can be used in two ways:

1. One prepares a quiz for the contributors, that contains only the test questions. If a contributor passes the test, she is allowed to answer payable questions.
2. For each set of questions, that are presented to a contributor, one question is a test question. If the contributor does not answer the test question correctly, the other answers from her are discarded, and she does not get paid for them.

During the experiment we used the second solution, because it requires contributor attention for every set of questions. A good practice recommends having 10-20% test questions in the input dataset[11]. Some of our test questions were correct (i.e. required an answer *Yes*) and some were incorrect (i.e. required an answer *No*), in order to ensure that a contributor can not select always the same answer and ignore the questions completely.

To obtain the test questions, we used reasoner *Pellet* [44] to find in the set of axioms generated by SLDM axioms that logically follows from the ontology. The questions corresponding to these axioms were then used as the test questions with a known correct answer *Yes*.

To obtain test questions with a correct *No* answer, we selected some of the axioms generated by SLDM containing

[11] `http://www.success.crowdflower.com`

only a named class in the right-hand side and replaced the class by some other, unrelated class, obtaining e.g. an axiom `dbo:Journalist` SOME `dbo:Book`. We also generated some false axioms by adding NOT to the left-hand side of an axiom inferred from the ontology, obtaining e.g. a sentence *Not every software is software*.

9.1.4. CrowdFlower experiment setup

The last activity to do before starting a crowdsourcing experiment is to setup the settings of the experiment and create an instruction for the contributors. Both of these steps are crucial with respect to ensuring quality of the experiment results.

We set up the settings in the following way[12]. As our questions are quite simple, we requested for contributors of the lowest level, as this allowed us to obtain the results faster. We decided on presenting 10 questions at once (i.e. on a single page) to a single contributor, as it should not take more than a few minutes to answer all of them. We chose to pay 0.03 USD for answering one page of questions. This is a typical payment on *CrowdFlower* for the contributors of the lowest level, and should maintain their commitment. We chose to request answers from 20 distinct contributors to one question. In the preliminary experiments we requested only 3 answers, but in such a case a single disagreement (e.g. when a contributor does not understand a question) makes the result unreliable. On the other hand, we did not want to increase the number too much, to keep the costs under control.

An instruction for a crowdsourcing experiment should be as simple as possible, yet answer all questions a contributor can ask. Moreover, it must contain examples of real questions, both positive and negative, and all steps that should be undertaken to solve them. For our experiment, we inform the contributors, that axioms are represented as sentences and their task is to decide whether a given sentence is true, false or is not clear. In the instruction we also mentioned one true sentence, one false sentence and one not clear sentence with an explanation in each case. The full text of the instruction is also available in the *Git* repository.

9.1.5. Experiment results

For the crowdsourcing experiment we generated two sets of axioms, one with the minimal support threshold $\theta_\sigma = 0.5$ and the other one with the threshold $\theta_\sigma = 0.8$. From each of the sets, we removed axioms that were logically entailed by the ontology or by the mined axioms for a superclass. The first set was translated to 425 payable questions and 61 test questions. Each of the payable questions was asked to 20 distinct contributors, leading to 8500 trusted answers. The second set was translated to 168 payable questions and 56 test questions, the payable questions yielded $20 \cdot 168 = 3360$ trusted answers. Reliability of all contributors was checked with the test questions and when it dropped below 70% (more than 30% of the presented test questions had wrong answers) they were refused to continue answering the questions.

[12] See `https://success.crowdflower.com/hc/en-us/articles/201855719-Guide-to-Basic-Job-Settings-Page` for additional explanation of the settings

Table 5: A summary of the results of the crowdsourcing experiment. For each question, we counted the numbers of *Yes*, *No* and *I don't know* answers and divided them into buckets: 0–5, 6–10, 11–15 and 16–20. In the table, we report the number of questions having all three counts in the buckets given by the first three columns of the table. In the last two columns specified are numbers of questions in the group, as an absolute number and relatively to the overall number of questions in the respective set. For example, 3 questions (i.e., 0.71% of all questions) in the first set were answered *Yes* by 6 to 10 contributors, *No* by 11 to 15 contributors and *I don't know* by 0 to 5 contributors.

Yes	*No*	*I don't know*	$\theta_\sigma = 0.5$		$\theta_\sigma = 0.8$	
16–20	0–5	0–5	295	69.41%	65	38.69%
11–15	6–10	0–5	40	9.41%	41	24.40%
11–15	0–5	0–5	78	18.35%	45	26.79%
6–10	11–15	0–5	3	0.71%	3	1.79%
6–10	6–10	0–5	5	1.18%	13	7.74%
0–5	16–20	0–5	2	0.47%	0	0.00%
0–5	11–15	0–5	2	0.47%	1	0.60%
overall			425	100.00%	168	100.00%

A summary of the results is presented in Table 5. In the first set 69.41% (resp. 38.69% in the second set) of the axioms were accepted by at least 80% of the contributors and 97.17% (resp. 89.88%) by at least 55% of the contributors. Our aim was to verify whether SLDM can mine new, meaningful axioms that can be added to the ontology. We find both results to answer positively to the question. All the verified knowledge was new, because the axioms that could be inferred from the ontology were used as the test questions.

To measure the level of disagreement, for each question we treated the numbers of *Yes*, *No* and *I don't know* answers as coordinates in a space and measured the Euclidean distance from all three crisp answers. We treated the minimal distance as the disagreement measure, where higher value means higher disagreement. For example, a question with 7 *Yes*, 8 *No* and 5 *I don't know* has coordinates $(7, 8, 5)$ and the nearest crisp answer is all *No* with coordinates $(0, 20, 0)$ (distance: $\sqrt{218}$). We present top 5 questions with the highest disagreement, selected from the axioms mined with the minimal support threshold $\theta_\sigma = 0.5$ in Table 6.

Question 1 probably refers to a geographical position of a crater, but the name of the predicate is very vague and, as it is in the `dbp` namespace, it lacks a description. Question 2 was mined, because 31172 out of 51019 instances, i.e. over 50%, of `dbo:WrittenWork` is asserted to the class `http://purl.org/ontology/bibo/Book`. Questions 3 and 4 display a similar problem with a complex structure and verbalization requiring knowledge about KR. Finally, question 5, on top of requiring knowledge about KR, requires also expert knowledge from the domain of law.

9.2. Extending the myExperiment ontology

To provide to the reader an insight how SLDM performs on a dataset different than DBpedia, we performed an experi-

Table 6: The top 5 questions with the most disagreement between the contributors. Every question is presented with the axiom it originated from and the number of contributors which submitted a given answer. For easier reading, the questions are numbered.

#	*Yes*	*No*	*I don't know*	A question and the axiom it originated from
1	7	8	5	"Every crater has E or W. E or W is Literal." `dbo:Crater SubClassOf: dbp:eOrW some rdf:PlainLiteral`
2	10	9	1	"Every written work (Written work is any text written to read it (e.g. - books, newspaper, articles)) is Book *." `dbo:WrittenWork SubClassOf: <http://purl.org/ontology/bibo/Book>`
3	10	8	2	"Every genre has instrument. instrument has is Primary Topic Of *. is Primary Topic Of * is Thing. instrument has label. label is Literal." `dbo:Genre SubClassOf: dbo:instrument some ((foaf:isPrimaryTopicOf some owl:Thing) and (rdfs:label some rdf:PlainLiteral))`
4	10	8	2	"Every genre has instrument. instrument has is Primary Topic Of *. is Primary Topic Of * is Document *. is Primary Topic Of * has Language (A language of the resource.). Language (A language of the resource.) is string. is Primary Topic Of * has Language (A language of the resource.) which value is en." `dbo:Genre SubClassOf: dbo:instrument some foaf:isPrimaryTopicOf some (foaf:Document and dc:language some xsd:string and dc:language value "en"^^xsd:string)`
5	10	6	4	"Every journalist has nationality. nationality has Legislature. Legislature has see Also. see Also is Thing. Legislature has type that is Bicameralism *." `dbo:Journalist SubClassOf: dbo:nationality some (dbp:legislature some ((rdfs:seeAlso some owl:Thing) and (dbo:type value dbr:Bicameralism)))`

mental evaluation on the RDF dataset published by myExperiment, a website designed for sharing and collaboration on scientific workflows describing experiments [40]. The dataset is published according to the Linked Data principles at `http://rdf.myexperiment.org/`. It consists of 2,907,345 triples, describing 526,201 different subjects using 115 different predicates and 44 different types. The used vocabulary is gathered in an ontology comprising of 10 modules and described in a detailed way on the website[13]. We used the RDF dataset as-is, without performing any logical reasoning on it.

We decided to run SLDM on every class that have at least 100 instances asserted to it in the dataset, i.e. on 34 classes. To keep the mined axioms simple and highly supported by the data, we set the minimal support threshold θ_σ to 0.9, maximal depth to 1 and disabled sampling. The full results are available in the *Git* repository and below we provide a discussion of the mined patterns for the classes from the module Annotations.

Table 7 presents these of the mined axioms, that were not logically entailed by the ontology. All five of the considered classes exhibited the same pattern w.r.t. the usage of property `dct:hasFormat` (axioms 1, 3, 5, 6, 10). While not presented in the description of the ontology, it seems that the property is used according to its intended usage[14], i.e., to link various representations of the same resource. W.r.t. axiom 2, the documentation explains that this is a functional property to attach submission text to a submission. The superclass in the axiom is Comment and it seems reasonable that a comment has some text. Axiom 4 correctly discovered part of knowledge hidden only in a description of rating-score property stating that a rating score must be between 1 and 5 (the other part is already present in the ontology). Axiom 7 is indirectly confirmed by the rest of the ontology, as the class Tagging seems to be reification of a ternary relation between an annotator, an annotated object and a tag. Axiom 8 is represented in the ontology in a weaker form `dct:title some rdfs:Literal` and, arguably, the weaker form may be preferred by some ontology engineers, but in the dataset itself every time the predicate `dct:title` is used, the literal is of type `xsd:string`. Finally, axiom 9 is not present in the documentation, but apparently it is used to link a tag to its HTML website.

This experiment shows that SLDM indeed can discover new facts about the data, these that are available only in textual form in the documentation, as well as these that reflect some, possibly unconscious, decisions about the design of a particular computer system handling the data.

9.3. Further experimental analysis of the algorithm

9.3.1. Comparison for different random seeds

In order to see how stable the results are depending on the choice of the random seed, we picked ten different random seeds using `random.org` and performed the following experiment: we fixed the sample size to be 250, the minimal support threshold θ_σ to be 0.95 and used the five classes from the bottom of the hierarchy depicted in Figure 8. This way, for each of the classes, we obtained 10 sets of axioms. For every axiom, we counted how many of the sets (extended with the DBpedia ontology) entailed the axiom. The detailed results are presented in Table 8.

[13] `http://rdf.myexperiment.org/ontologies/`

[14] `http://dublincore.org/documents/dcmi-terms/#terms-hasFormat`

Table 7: Axioms mined for the classes from the module Annotations of the myExperiment ontology. Presented are only the axioms that are not logically entailed by the ontology. The axioms are grouped by their subclass and numbered for easier reading.

#	axiom
	annotations:Comment SubClassOf:
1	dct:hasFormat some owl:Thing
2	base:text some xsd:string
	annotations:Favourite SubClassOf:
3	dct:hasFormat some owl:Thing
	annotations:Rating SubClassOf:
4	annotations:rating-score some xsd:decimal[<= 5]
5	dct:hasFormat some owl:Thing
	annotations:Tagging SubClassOf:
6	dct:hasFormat some owl:Thing
7	annotations:uses-tag some owl:Thing
	annotations:Tag SubClassOf:
8	dct:title some xsd:string
9	foaf:homepage some owl:Thing
10	dct:hasFormat some owl:Thing

In general we obtained a very good stability, with at least 68% of the axioms being mined for all of the random seeds. In case of the ProgrammingLanguage class, every time exactly the same set of axioms was obtained. For the Book class, 16 out of 18 axioms were entailed by all the sets and the remaining 2 axioms by 8 out of 10 sets. Similar was the case of the Journalist class, but with, respectively, 11 and 5 out of 16 axioms. Finally, for the Crater class, one axiom was entailed by 4 sets, and for the MusicGenre class, single axioms were entailed by 3, 4 and 6 sets.

9.3.2. Comparison of different sampling strategies

To gauge the variability of the obtained results with respect to the minimal support threshold θ_σ, sample size, sampling strategy and class, we executed the following experiment. We selected all 5 classes from the bottom of the hierarchy presented in Figure 8, we considered the minimal support thresholds θ_σ from the set $\{0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ and the sample sizes n from the set $\{50, 100, 200, 500, 1000\}$. We also considered all three sampling strategies. This way we obtained $5 \cdot 6 \cdot 5 \cdot 3 = 450$ configurations of SLDM. For each of them, we executed SLDM, removed from the obtained set of axioms these that are logically entailed by the DBpedia ontology and counted the remaining ones. The results are presented in Table 11.

In 99/150, i.e. 66%, of the cases, the predicates counting strategy yielded the highest number of mined axioms (ties included). To further analyze the differences between the strategies, we wanted to see in how many cases all of the axioms mined using one strategy are logically entailed by the axioms mined with the other strategy. The comparison presented in Table 9 hints that the counting strategies generally perform better than the uniform strategy. It is of no surprise since they both employ additional knowledge derived from computing statistics on the SPARQL endpoint. The performance of both of the counting strategies is very similar, with a little bit of advantage for the predicates counting strategy. Given that the SPARQL query (c.f. Section 4) corresponding to the predicates counting strategy is simpler, probably this strategy should be favored over the triples counting strategy.

9.3.3. Runtime performance

To judge the performance of the implementation described in Section 8 we mined axioms for the DBpedia class Book using various settings. We used uniform sampling strategy and randomly selected 10 different random seeds to account for different samples. We considered each of the following 6 minimal support thresholds θ_σ: $\{0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ and each of the following 5 sample sizes: $\{50, 100, 200, 500, 1000\}$, obtaining $10 \cdot 6 \cdot 5 = 300$ different settings. Moreover, each of the experiments was repeated 10 times to account for the variability normal to the computer system, yielding overall 3000 experiments being run in the timespan of 15 hours. The experiments were run on a server equipped with two *Intel Xeon E5-2630 v3* running at 2.4 GHz each and with 256 GB of the RAM. The SPARQL endpoint was set up on the same machine.

We measured overall CPU time of the mining, including starting of the Java Virtual Machine (JVM) and including the system CPU time accounted to the process. A single run of the experiment never took more than 2 minutes, even for the lowest minimal support threshold and the highest sample size. The detailed statistics are presented in Figure 9.

While performing the experiment, we also measured maximal memory consumption of the JVM. As Java is a garbage collecting language, we report only the highest values in Table 10. Without detailed profiling of the execution it is unclear if, and to what extent, these could be lowered.

Finally, during the experiment, we counted the number of queries posed to the SPARQL endpoint. The number never exceeded 400 and the detailed statistics, aggregated in the same manner as for CPU time, are presented in Figure 10.

10. Conclusions

In this paper we presented Swift Linked Data Miner, a novel approach for mining ontological axioms directly from on-line RDF datasets. Capabilities of SLDM cover the whole grammar of OWL 2 EL superclass expressions. SLDM is readily available for use as a *Protégé* plugin available for download from the *Git* repository `https://bitbucket.org/jpotoniec/sldm`.

We also showed that SLDM can be used to mine RDF Data Shapes instead of ontological axioms. We presented a transformation from SLDM axioms to the corresponding shapes expressed in Shapes Constraint Language (SHACL).

To validate that SLDM can be applied in a real use-case, we mined axioms for a set of classes from the *DBpedia* ontology, and conducted a crowdsourcing experiment to validate them. The experiment confirmed our hypothesis, as most of the axioms was accepted by the contributors of a crowdsourcing platform. All materials used for the experiment and the obtained results are published to the above-mentioned *Git* repository. We

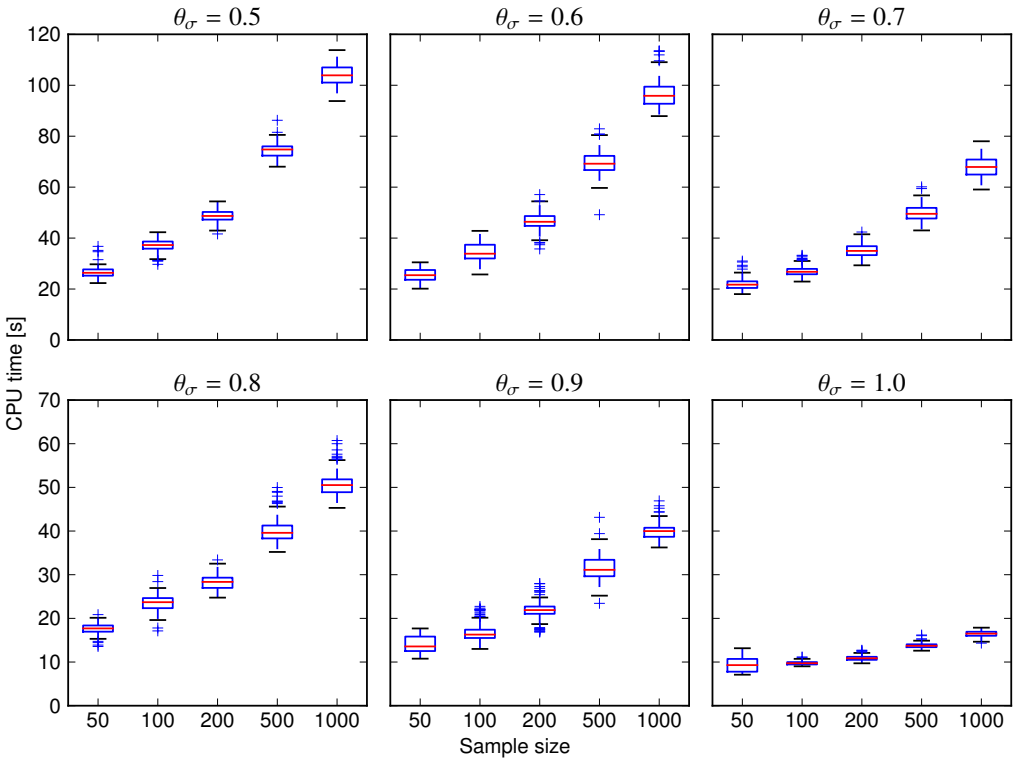


Figure 9: CPU time used by SLDM for mining axioms for the Book class in DBpedia, averaged across 10 repeats for each of 10 different random seeds. Each chart corresponds to a fixed minimal support threshold θ_σ. The charts all share the same horizontal axis presenting the sample size. The vertical axes are shared within rows and present the overall CPU time of the mining (including starting of Java Virtual Machine (JVM)).

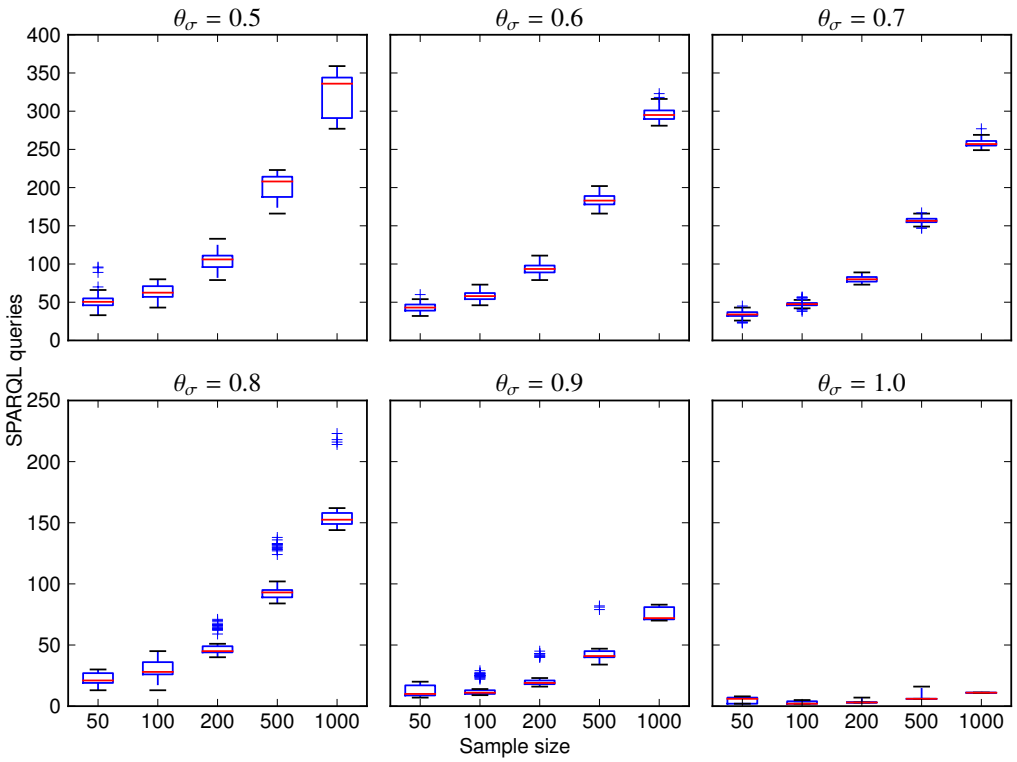


Figure 10: Number of SPARQL queries posed to the SPARQL endpoint by SLDM for mining axioms for the Book class in DBpedia, averaged across 10 repeats for each of 10 different random seeds. Each chart corresponds to a fixed minimal support threshold θ_σ. The charts all share the same horizontal axis presenting the sample size and the vertical axes are shared within rows.

Table 8: Experimental analysis of the stability of the algorithm. For each of the five classes ten sets of axioms were generated using different random seed. The column *distinct axioms* gives the number of distinct axioms mined, while the column *all axioms* gives the number of all axioms in all of the mined sets. The numbered columns correspond to the numbers of sets and the values in cell are the overlap size. For example, value 5 in the column 8 of the row Journalist means that in case of 5 axioms they were logically entailed by 8 out of 10 sets of the axioms. These 8 sets may differ from axiom to axiom.

class	distinct axioms	all axioms	number of axioms entailed by given number of the sets									
			1	2	3	4	5	6	7	8	9	10
Book	18	160	0	0	0	0	0	0	0	2	0	16
Crater	13	120	0	0	0	1	0	0	0	0	0	12
Journalist	16	150	0	0	0	0	0	0	0	5	0	11
MusicGenre	16	124	0	0	1	1	0	1	0	0	0	13
ProgrammingLanguage	15	150	0	0	0	0	0	0	0	0	0	15

Table 9: Comparison of different sampling strategies: *uni.* stands for uniform, *pc* for predicates counting, *tc* for triples counting. A row corresponds to the set of axioms for which entailment is checked, while the columns to the set of axioms serving as the reference for the entailment checking. For example, in 122 out of 150 cases (different classes, supports and samples sizes) all of the axioms mined using the uniform strategy were all logically entailed by the axioms mined using the same settings, but with the predicates counting strategy.

	entailed by		
	uni.	pc	tc
uni.	–	122/150=81%	121/150=81%
pc	63/150=42%	–	100/150=67%
tc	69/150=46%	99/150=66%	–

Table 10: Memory consumption of the whole Java Virtual Machine (JVM) while mining for the axioms for the DBpedia class Book. The reported values are maximal over 10 repeats for each 10 different random seeds and for each of 5 different sample sizes given a fixed minimal support threshold θ_σ.

θ_σ	0.5	0.6	0.7	0.8	0.9	1.0
Memory [GB]	9.82	9.81	5.44	4.85	2.64	0.79

also show that the required computational resources are in very reasonable limits and are easy to provide using contemporary computers.

In the future, we would like to cover more expressive variants of OWL 2, e.g. by including universal quantification (ONLY). We also would like to extend SLDM to mine more types of axioms, for example to express disjointness.

Acknowledgements. Jedrzej Potoniec acknowledges the support from the Polish National Science Center (Grant No 2013/11/N/ST6/03065). This work was partially supported by the PARENT-BRIDGE program of Foundation for Polish Science, co-financed from European Union, Regional Development Fund (Grant No POMOST/2013-7/8). Agnieszka Ławrynowicz acknowledges the support from the Polish National Science Center (Grant No 2014/13/D/ST6/02076).

[1] Aggarwal, C. C., Han, J. (Eds.), 2014. Frequent Pattern Mining. Springer.

[2] Baader, F., Ganter, B., Sertkaya, B., Sattler, U., 2007. Completing description logic knowledge bases using formal concept analysis. In: Veloso, M. M. (Ed.), IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. pp. 230–235.

[3] Biron, P. V., Malhotra, A., Oct. 2004. XML Schema Part 2: Datatypes Second Edition. W3C recommendation, W3C, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[4] Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S., 2009. DBpedia - A crystallization point for the Web of Data. J. Web Sem. 7 (3), 154–165.
URL `http://dx.doi.org/10.1016/j.websem.2009.07.002`

[5] Bühmann, L., Lehmann, J., 2013. Pattern based knowledge base enrichment. In: Alani, H., Kagal, L., Fokoue, A., Groth, P. T., Biemann, C., Parreira, J. X., Aroyo, L., Noy, N. F., Welty, C., Janowicz, K. (Eds.), The Semantic Web - ISWC 2013. Vol. 8218 of Lecture Notes in Computer Science. Springer, pp. 33–48.

[6] Carothers, G., Prud'hommeaux, E., Feb. 2014. RDF 1.1 turtle. W3C recommendation, W3C, http://www.w3.org/TR/2014/REC-turtle-20140225/.

[7] Dean, M., Schreiber, G., Feb. 2004. OWL web ontology language reference. W3C recommendation, W3C, http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[8] Donnelly, K., 2006. SNOMED-CT: The Advanced Terminology and Coding System for eHealth. In: Bos, L and Roa, L and Yogesan, K and OConnell, B and Marsh, A and Blobel, B (Ed.), MEDICAL AND CARE COMPUNETICS 3. Vol. 121 of Studies in Health Technology and Informatics. I O S PRESS, NIEUWE HEMWEG 6B, 1013 BG AMSTERDAM, NETHERLANDS, pp. 279–290, 3rd International Congress on Medical and Care Compunetics, The Hague, NETHERLANDS, JUN 07-09, 2006.

[9] Fanizzi, N., d'Amato, C., Esposito, F., 2008. DL-FOIL concept learning in description logics. In: Železný, F., Lavrac, N. (Eds.), Inductive Logic Programming, 18th International Conference, ILP 2008, Prague, Czech Republic, September 10-12, 2008, Proceedings. Vol. 5194 of Lecture Notes in Computer Science. Springer, pp. 107–121.

[10] Fleischhacker, D., Völker, J., 2011. Inductive learning of disjointness axioms. In: Meersman, R., Dillon, T. S., Herrero, P., Kumar, A., Reichert, M., Qing, L., Ooi, B. C., Damiani, E., Schmidt, D. C., White, J., Hauswirth, M., Hitzler, P., Mohania, M. K. (Eds.), On the Move to Meaningful Internet Systems: OTM 2011. Vol. 7045 of Lecture Notes in Computer Science. Springer, pp. 680–697.

[11] Fu, Y., Han, J., 1995. Meta-rule-guided mining of association rules in relational databases. In: KDOOD/TDOOD. pp. 39–46.

[12] Fuchs, N. E., Kaljurand, K., Kuhn, T., 2008. Attempto controlled english for knowledge representation. In: Baroglio, C., Bonatti, P. A., Maluszynski, J., Marchiori, M., Polleres, A., Schaffert, S. (Eds.), Reasoning Web, 4th International Summer School 2008, Venice, Italy, September 7-11, 2008, Tutorial Lectures. Vol. 5224 of Lecture Notes in Computer Science. Springer, pp. 104–124.

[13] Galárraga, L. A., Teflioudi, C., Hose, K., Suchanek, F. M., 2013. AMIE: association rule mining under incomplete evidence in ontological knowledge bases. In: Schwabe, D., Almeida, V. A. F., Glaser, H., Baeza-Yates, R. A., Moon, S. B. (Eds.), 22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013. International World Wide Web Conferences Steering Committee / ACM, pp. 413–422.

[14] Group, W. O. W., Dec. 2012. OWL 2 Web Ontology Language Document Overview (Second Edition). W3C recommendation, W3C, https://www.w3.org/TR/owl2-overview/.

[15] Guha, R., Brickley, D., Feb. 2014. RDF schema 1.1. W3C recommendation, W3C, http://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[16] Hanika, F., Wohlgenannt, G., Sabou, M., 2014. The ucomp protégé plugin: Crowdsourcing enabled ontology engineering. In: Janowicz, K., Schlobach, S., Lambrix, P., Hyvönen, E. (Eds.), Knowledge Engineering

Table 11: Number of axioms mined for a given class, using a given sampling strategy (*uni.* stands for uniform, *pc* for predicates counting, *tc* for triples counting), sample size n and minimal support threshold θ_σ. The axioms logically entailed by the DBpedia ontology were removed from the count.

		Book			Crater			Journalist			MusicGenre			ProgrammingLanguage		
θ_σ	n	uni.	pc	tc	uni.	pc	tc	uni.	pc	tc	uni.	pc	tc	uni.	pc	tc
0.5	50	54	88	85	19	23	20	50	19	66	18	40	63	49	34	61
	100	87	102	87	22	23	21	69	19	64	13	35	50	32	57	51
	200	81	109	99	22	23	23	69	96	76	20	40	59	25	45	41
	500	96	106	102	20	20	20	96	77	96	20	34	47	22	37	38
	1000	96	111	113	20	20	20	113	125	113	22	30	31	22	22	22
0.6	50	58	61	60	16	21	20	38	79	57	7	25	30	23	31	28
	100	46	67	54	16	17	17	62	63	56	7	24	38	15	27	24
	200	51	66	61	16	16	17	61	65	60	9	29	34	13	20	23
	500	55	68	68	16	16	16	83	68	64	11	27	34	13	19	19
	1000	60	69	75	16	16	16	116	110	110	14	18	18	13	13	13
0.7	50	26	37	29	10	17	16	36	55	49	7	21	28	11	11	19
	100	39	42	41	7	16	10	53	55	58	7	17	29	11	16	15
	200	31	49	54	10	16	10	53	61	56	7	17	29	11	13	17
	500	41	52	55	10	10	10	85	83	71	7	13	23	11	11	13
	1000	40	53	54	10	10	10	105	105	91	8	9	10	11	11	11
0.8	50	14	29	17	7	10	7	34	61	16	7	11	17	9	11	13
	100	19	28	25	7	7	10	16	16	16	7	8	19	9	11	11
	200	21	26	25	7	7	7	16	16	16	7	8	18	9	11	11
	500	26	27	31	7	7	7	41	26	17	7	7	16	11	11	11
	1000	24	36	27	7	7	7	85	78	32	8	8	8	10	10	10
0.9	50	10	14	9	7	7	7	15	15	15	7	7	15	9	11	11
	100	9	17	15	7	7	7	13	15	15	7	8	11	9	11	9
	200	11	12	10	7	7	7	13	15	15	7	7	13	9	11	11
	500	9	12	14	7	7	7	15	15	15	8	8	8	9	10	9
	1000	10	12	12	7	7	7	15	15	15	8	8	8	9	9	9
1.0	50	7	9	9	7	7	7	3	14	13	6	6	6	8	5	6
	100	4	5	6	7	5	6	3	11	3	6	5	5	7	7	5
	200	2	6	4	5	5	5	3	3	3	1	3	5	5	7	5
	500	2	2	2	1	1	1	3	3	3	1	5	5	5	5	5
	1000	2	2	2	1	1	1	3	3	3	1	3	3	0	0	0

and Knowledge Management. Vol. 8876 of Lecture Notes in Computer Science. Springer, pp. 181–196.

[17] Harris, S., Seaborne, A., Mar. 2013. SPARQL 1.1 query language. W3C recommendation, W3C, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[18] Hawke, S., Motik, B., Bao, J., Polleres, A., Patel-Schneider, P., Dec. 2012. rdf:plainliteral: A datatype for RDF plain literals (second edition). W3C recommendation, W3C, http://www.w3.org/TR/2012/REC-rdf-plain-literal-20121211/.

[19] Horridge, M., Bechhofer, S., 2011. The OWL API: A java API for OWL ontologies. Semantic Web 2 (1), 11–21.

[20] Horridge, M., Patel-Schneider, P., Dec. 2012. OWL 2 Web Ontology Language Manchester Syntax (Second Edition). W3C note, W3C, http://www.w3.org/TR/2012/NOTE-owl2-manchester-syntax-20121211/.

[21] Horrocks, I., Patel-Schneider, P. F., 2004. Reducing OWL entailment to description logic satisfiability. J. Web Sem. 1 (4), 345–357.
URL http://dx.doi.org/10.1016/j.websem.2004.06.003

[22] Kaljurand, K., 2007. Attempto Controlled English as a Semantic Web Language. Ph.D. thesis, Faculty of Mathematics and Computer Science, University of Tartu.
URL http://attempto.ifi.uzh.ch/site/pubs/papers/phd_kaljurand.pdf

[23] Kaljurand, K., Fuchs, N. E., 2007. Verbalizing OWL in attempto controlled english. In: Golbreich, C., Kalyanpur, A., Parsia, B. (Eds.), Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions, Innsbruck, Austria, June 6-7, 2007. Vol. 258 of CEUR Workshop Proceedings. CEUR-WS.org.

[24] Knublauch, H., Ryman, A., Jan. 2016. Shapes Constraint Language (SHACL). W3C working draft, W3C, http://www.w3.org/TR/2016/WD-shacl-20160128/.

[25] Lawrynowicz, A., Potoniec, J., 2014. Pattern based feature construction in semantic data mining. Int. J. Semantic Web Inf. Syst. 10 (1), 27–65.

[26] Lehmann, J., 2009. DL-learner: Learning concepts in description logics. Journal of Machine Learning Research 10, 2639–2642.

[27] Lehmann, J., Auer, S., Bühmann, L., Tramp, S., 2011. Class expression learning for ontology engineering. J. Web Sem. 9 (1), 71–81.

[28] Lehmann, J., Völker, J., 2014. Perspectives on Ontology Learning. Vol. 18. IOS Press.

[29] Magka, D., Kazakov, Y., Horrocks, I., 2011. Tractable extensions of the description logic $\mathcal{EL}$ with numerical datatypes. Journal of Automated Reasoning 47 (4), 427–450.

[30] McBride, B., 2002. Jena: A semantic web toolkit. IEEE Internet Computing 6 (6), 55–59.

[31] Mortensen, J., 2013. Crowdsourcing ontology verification. In: Alani, H., Kagal, L., Fokoue, A., Groth, P. T., Biemann, C., Parreira, J. X., Aroyo, L., Noy, N. F., Welty, C., Janowicz, K. (Eds.), The Semantic Web - ISWC 2013. Vol. 8219 of Lecture Notes in Computer Science. Springer, pp. 448–455.

[32] Motik, B., Grau, B. C., Horrocks, I., Fokoue, A., Wu, Z., Dec. 2012. OWL 2 Web Ontology Language Profiles (Second Edition). W3C recommendation, W3C, http://www.w3.org/TR/2012/REC-owl2-profiles-20121211/.

[33] Motik, B., Shearer, R., Horrocks, I., 2009. Hypertableau Reasoning for Description Logics. Journal of Artificial Intelligence Research 36, 165–228.

[34] Parsia, B., Rudolph, S., Krötzsch, M., Patel-Schneider, P., Hitzler, P., Dec. 2012. OWL 2 web ontology language primer (second edition). Tech. rep., W3C, http://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[35] Patel-Schneider, P., Motik, B., Grau, B. C., Dec. 2012. OWL 2 web ontology language direct semantics (second edition). W3C recommendation, W3C, http://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/.

[36] Patel-Schneider, P., Parsia, B., Motik, B., Dec. 2012. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). W3C recommendation, W3C, http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

[37] Potoniec, J., 2014. Towards ontology refinement by combination of machine learning and attribute exploration. In: Lambrix, P., Hyvönen, E., Blomqvist, E., Presutti, V., Qi, G., Sattler, U., Ding, Y., Ghidini, C. (Eds.), Knowledge Engineering and Knowledge Management. Vol. 8982 of Lecture Notes in Computer Science. Springer, pp. 225–232.

[38] Potoniec, J., Lawrynowicz, A., 2015. Combining ontology class expression generation with mathematical modeling for ontology learning. In: Bonet, B., Koenig, S. (Eds.), Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA. AAAI Press, pp. 4198–4199.

[39] Potoniec, J., Rudolph, S., Lawrynowicz, A., 2014. Towards combining machine learning with attribute exploration for ontology refinement. In: Horridge, M., Rospocher, M., van Ossenbruggen, J. (Eds.), Proceedings of the ISWC 2014 Posters & Demonstrations Track a track within the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Italy, October 21, 2014. Vol. 1272 of CEUR Workshop Proceedings. CEUR-WS.org, pp. 229–232.

[40] Roure, D. D., Goble, C., Stevens, R., 2009. The design and realisation of the virtual research environment for social sharing of workflows. Future Generation Computer Systems 25 (5), 561 – 567.
URL http://www.sciencedirect.com/science/article/pii/S0167739X08000939

[41] Rudolph, S., 2008. Acquiring generalized domain-range restrictions. In: Medina, R., Obiedkov, S. A. (Eds.), Formal Concept Analysis, 6th International Conference, ICFCA 2008, Montreal, Canada, February 25-28, 2008, Proceedings. Vol. 4933 of Lecture Notes in Computer Science. Springer, pp. 32–45.

[42] Sazonau, V., Sattler, U., Brown, G., 2015. General terminology induction in OWL. In: Arenas, M., Corcho, Ó., Simperl, E., Strohmaier, M., d'Aquin, M., Srinivas, K., Groth, P. T., Dumontier, M., Heflin, J., Thirunarayan, K., Staab, S. (Eds.), The Semantic Web - ISWC 2015. Vol. 9366 of Lecture Notes in Computer Science. Springer, pp. 533–550.

[43] Siorpaes, K., Hepp, M., 2008. Games with a Purpose for the Semantic Web. IEEE Intelligent Systems 23 (3), 50–60.

[44] Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., Katz, Y., 2007. Pellet: A practical OWL-DL reasoner. J. Web Sem. 5 (2), 51–53.

[45] Steyskal, S., Coyle, K., Jan. 2016. SHACL Use Cases and Requirements. W3C working draft, W3C, https://www.w3.org/TR/2016/WD-shacl-ucr-20160122/.

[46] van Harmelen, F., McGuinness, D., Feb. 2004. OWL web ontology language overview. W3C recommendation, W3C, http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[47] Verborgh, R., Hartig, O., De Meester, B., Haesendonck, G., De Vocht, L., Vander Sande, M., Cyganiak, R., Colpaert, P., Mannens, E., Van de Walle, R., Oct. 2014. Querying datasets on the Web with high availability. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C., Vrandečić, D., Groth, P., Noy, N., Janowicz, K., Goble, C. (Eds.), Proceedings of the 13th International Semantic Web Conference. Vol. 8796 of Lecture Notes in Computer Science. Springer, pp. 180–196.

[48] Völker, J., Niepert, M., 2011. Statistical schema induction. In: Antoniou, G., Grobelnik, M., Simperl, E. P. B., Parsia, B., Plexousakis, D., Leenheer, P. D., Pan, J. Z. (Eds.), The Semantic Web: Research and Applications. Vol. 6643 of Lecture Notes in Computer Science. Springer, pp. 124–138.

[49] Wood, D., Lanthaler, M., Cyganiak, R., Feb. 2014. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.