Querying RDF by Example using Least General Generalisation

Lorenz Bühmann and Jens Lehmann

Universität Leipzig
Postfach 100920, 04009 Leipzig, Germany
{buehmann,lehmann}@informatik.uni-leipzig.de
http://aksw.org

Abstract. Semantic technologies offer high flexibility and rich modelling possibilities for creating knowledge bases. However, querying those knowledge bases is often not straightforward and requires expert knowledge, in particular the ability to formulate queries. In this article, we present the supervised machine learning method QTL2, which allows to learn SPARQL queries by example. This allows non-experts to query a knowledge base without needing to know the underlying schema or having expertise in the SPARQL query language. A relevant feature of the algorithm in the context of SPARQL is that it can work using only few examples as input to minimise human effort. We developed the QTL2 algorithm based on the so called Least General Generalisation (lgg) operation which we define for a fragment of SPARQL. An evaluation on the well known DBpedia dataset shows that the method is robust against noisy input and is able to learn complex queries.

1 Introduction

The Web of Data has continued to grow over the past years from very few datasets being available in 2007 to almost 10 000 publicly available RDF datasets[1] covering various domains such as geospatial data, bibliographic information, music, arts and encyclopedic knowledge. However, accessing this knowledge via queries usually requires expertise in both the query language (usually SPARQL) and the modelling of the underlying knowledge base. An approach, with increasing popularity in research, to overcome this problem is the use of question answering methods capable of converting natural language input text to SPARQL queries. Systems such as TBSL [1], Ginseng [2], NLP-Reduce[2] or PowerAqua [3] support this functionality. However, those systems can be brittle as they usually rely on a sequential processing chain in which errors can occur at each stage. This means that the probability of correctly answering a query roughly corresponds to the product of the probabilities of each element in the processing chain.

In this paper, we propose a supervised machine learning algorithm for SPARQL, which can, in particular, be employed as feedback channel or alternative to question answering systems. Given a labelled set of RDF resources, the algorithm constructs a

[1] http://lodstats.aksw.org/
[2] http://www.ifi.uzh.ch/ddis/research/talking-to-the-semantic-web/nlpreduce/

ranked list of SPARQL queries each of which returns (the majority of) resources in the input set.

For instance, given the resources $\mathrm{Common\_carotid\_artery}$, $\mathrm{Median\_sacral\_artery}$ and $\mathrm{Renal\_artery}$ in DBpedia[3] [4] it learns the query[4]

```
SELECT DISTINCT ?s WHERE {
  ?s a dbo:Artery .
  ?s dbo:branchFrom ?o .
  ?o a dbo:Artery .
  ?o dbo:vein :Inferior_vena_cava .}
```

which can be described as "arteries that are branching from some other artery which have some dependencies to vein $\mathrm{Inferior\_vena\_cava}$ ".

The algorithm builds on the concept of least general generalisation and is particularly suited to exploiting a low number of examples, which is a typical scenario when relying on manual user input. As we will demonstrate, it is capable of working on large knowledge bases and can learn nested queries involving a high number of triple patterns.

Specifically, we developed the QTL2 algorithm and provide the following extensions over the state of the art:

- Designing the first supervised, noise tolerant algorithm for learning SPARQL queries from RDF resources for a well-defined fragment of SPARQL.
- Inference-aware formalisation of RDF query trees and their least general generalisation for SPARQL.
- Very high precision and high recall in a challenging evaluation setting.

The paper is structured as follows: Section 2 describes query trees as the underlying data structure corresponding to a fragment of SPARQL. Based on those, we define query tree subsumption and least general generalisation with inference in 3. The actual algorithm is covered in Section 4. The experimental setup, research questions and results are covered in Section 5. Finally, we describe related work in Section 6 and conclude in Section 7.

2 Preliminaries

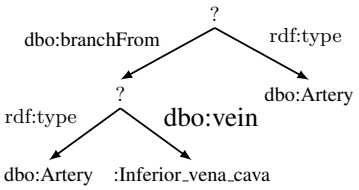
We will often use standard notions from the RDF[5] and SPARQL[6] specifications, e.g. triple (s, p, o) consisting of a subject s, a predicate p and an object o; RDF graph as a set of triples; triple pattern and basic graph pattern. We denote the set of RDF resources with R and the set of RDF literals with L.

[3] we refer to DBpedia version 2015-04

[4] we use the prefixes dbo for http://dbpedia.org/ontology and : for http://dbpedia.org/resource/ in the paper

[5] http://www.w3.org/TR/rdf-concepts

[6] http://www.w3.org/TR/rdf-sparql-query




```
SELECT DISTINCT ?s WHERE {
  ?s rdf:type dbo:Artery .
  ?s dbo:branchFrom ?o .
  ?o rdf:type dbo:Artery .
  ?o dbo:vein :Inferior_vena_cava .
}
```

Fig. 1: Query tree (left) and corresponding SPARQL query (right).

Query Trees

This section reuses definitions and operations from the work in [5] which are necessary for the subsequent sections.

We call the structure which is used internally by the QTL2 algorithm, a *query tree*. A query tree is the tree-like representation of information about a resource resp. a set of resources. If the query tree describes a set of resources, it roughly corresponds to a SPARQL query, but not all SPARQL queries can be expressed as query trees.

Definition 1 (Query Tree). *A* query tree *is a rooted, directed, labelled tree* $T = (V, E, \ell)$*, where* V *is a finite set of* nodes*,* $E \subset V \times R \times V$ *is a finite set of* edges*,* $NL = L \cup R \cup \{?\}$ *is a set of* node labels *and* $\ell : V \to NL$ *is the* labelling function*. The* root *of* T *is denoted as root*(T)*. If* $\ell(root(T)) =?$*, we call the query tree* projectable*. The set of all query trees is denoted by* $\mathcal{T}$ *and* $\mathcal{T}_C$ *for projectable query trees. We use the notions* $V(T) := V$*,* $E(T) := E$*,* $\ell(T) := \ell$ *to refer to nodes, edges and label function of a tree* T*. We say* $v_1 \xrightarrow{e_1} \cdots \xrightarrow{e_n} v_{n+1}$ *is a* path *of length* n *from* v_1 *to* v_{n+1} *in* T *iff* $(v_i, e_i, v_{i+1}) \in E$ *for* $1 \leq i \leq n$*. The depth of a tree is the length of its longest path from the root to a leaf.*

As common for tree-based datastructures, we treat subtrees as query trees themselves, which makes operations like tree traversal much more convenient.

Definition 2 (Subtrees as Query Trees). *If* $T = (V, E, \ell)$ *is a query tree and* $v \in V$*, then we denote by* $T(v)$ *the tree* $T' = (V', E', \ell')$ *with root*$(T') = v$*,* $V' = \{v' \mid$ *there is a path from* v *to* $v'\}$*,* $E' = E \cap V' \times R \times V'$ *and* $\ell' = \ell_{|V'}$*.*

Query trees act as a bridge between the description of a resource in an RDF graph and SPARQL queries containing the resource in their result set (see Fig. 1). Using them enables us to define a very efficient learning algorithm for SPARQL queries.

We can map each resource in an RDF graph to a query tree. Intuitively, the tree corresponds to the neighbourhood of the resource in the graph, which is sometimes also referred to as *Concise Bounded Description* (CBD).[7] According to the W3C submission, a CBD denotes a subgraph consisting of those statements which together constitute a focused body of knowledge about the resource denoted by that particular node. For reasons of efficiency, we have to limit ourselves to a "depth" of the subgraph from the

[7] http://www.w3.org/Submission/CBD/

starting node, in order to map a resource to a tree. This recursion depth corresponds to the maximum nesting of triple patterns, which can be learned by the QTL2 algorithm, which we will detail in Section 4. Technically, a query tree T with a maximum depth d for a given resource r can be constructed by recursively fetching outgoing triples with subject r in an RDF graph G, denoted as $T = map(G, d, r)$. For each fetched triple (r, p_i, o_i) a child with label o_i will be added to the root node labeled with r by using edge label p_i. To avoid redundancy, we do this only for resources that haven't already been visited on the same path in the tree and to ensure a tree structure, we create a new child node for each o_i, thus, multiple nodes in the tree can be assigned with the same label.

We use the mapping function $sparql(T, ?var)$ to generate a SPARQL query with the single projection variable $?var$ being $root(T)$ for a projectable query tree T. This can be easily done by preorder traversal on T such that a corresponding triple pattern is added for each edge. See Fig. 1 for a mapping example.

Note that a query tree T does not contain cycles, whereas an RDF graph G can, of course, contain cycles. Also note that query trees intentionally only support a limited subset of SPARQL.

Least General Generalisation

In the following, we define subsumption of query trees analogous to [5]. Intuitively, if a query tree T_1 is subsumed by T_2, then the SPARQL query corresponding to T_1 returns fewer results than the SPARQL query corresponding to T_2. The definition of query tree subsumption will be done in terms of the SPARQL algebra. Similar as in [6,7], we use the notion $[[q]]_G$ as the evaluation of a SPARQL query q in an RDF graph G.

Definition 3 (Query Tree Subsumption). *Let T_1 and T_2 be projectable query trees. T_1 is subsumed by T_2, denoted as $T_1 \preceq T_2$, if we have $[[sparql(T_1, ?x)]]_G(?x) \subseteq [[sparql(T_2, ?x)]]_G(?x)$ for any RDF graph G.*

Based on the above definition, we require a procedure to decide subsumption between query trees:

Definition 4 (Structural Query Tree Subsumption). *Let T_1 and T_2 be query trees. T_1 is structurally subsumed by T_2, denoted as $T_1 \leq T_2$, if*

1. *$l(root(T_2)) = ?$ and*
   - *for each edge $(root(T_2), e_i, v_j) \in E(T_2)$ there is an edge $(root(T_1), e_i, v_k) \in E(T_1)$ such that $T(v_j) \leq T(v_k)$*
2. *otherwise, if $\ell(root(T_1)) = \ell(root(T_2))$*

[5] showed that structural query tree subsumption implies query tree subsumption under SPARQL semantics, i.e. $T_1 \leq T_2 \Rightarrow T_1 \preceq T_2$.

The least general generalisation (lgg) operation takes two query trees as input and returns the most specific query tree which subsumes both input trees. It can be defined algorithmically as shown in Function lgg which we extended and generalised from [5].

It takes two query trees T_1 and T_2 as input and returns T as their lgg. If the root nodes of both query trees denote the same resource or literal, both trees are equal and any one of

```
1  v1 = root(T1); v2 = root(T2)
2  if ℓ(v1) ≠ ? ∧ ℓ(v1) = ℓ(v2) then return T1
3  init T = (V, E, ℓ) with V = {v}, E = ∅, ℓ(v) = ?
4  P = relatedPredicates(T1, T2)
5  foreach ⟨p1, p2, p⟩ ∈ P do
6      foreach v1' with (v1, p1, v1') ∈ E(T1) do
7          foreach v2' with (v2, p2, v2') ∈ E(T2) do
8              v' = root(lgg(T(v1'), T(v2'))); add = true
9              foreach v_prev with (v, p, v_prev) ∈ E(T) do
10                 if add = true then
11                     if isSubTreeOf(T(v_prev), T(v')) then add = false
12                     if isSubTreeOf(T(v'), T(v_prev)) then remove edge (v, p, v_prev) from T
13             if add = true then add edge (v, p, v') to T
14 return T
```

Function lgg(T_1, T_2)

them (here T_1) can be returned immediately (Line 2). Otherwise, T is initialised as empty tree with ? used as label in Line 3. Line 4 computes triples of related predicates (see definition below) of T_1 and T_2 – only triples of predicates that are related by subsumption will be part of the lgg by using the more general predicate p as edge then. Line 6 and 7 are used for comparing pairs of edges in T_1 and T_2. For each combination, the lgg is recursively computed (Line 8). However, in order to keep the resulting tree small, only edges which do not subsume another edge are preserved (Lines 9 to 12). Finally, Line 13 adds the computed lgg to the tree T, which is returned.

The base algorithm can be obtained by using the following definitions for subtrees and related predicates:

1. $\mathrm{relatedPredicates}(T_1, T_2)$ returns predicates that occur in both trees, more formally triples $\langle p, p, p\rangle$ such that $(\mathit{root}(T_1), p, v) \in E(T_1)$ and $(\mathit{root}(T_2), p, v') \in E(T_2)$ for arbitrary v, v'[8]
2. $\mathrm{isSubTreeOf}(T_1, T_2)$ returns true if $T_1 \leq T_2$, otherwise false

Note that extracting those concepts from other work is required to establish inference based lggs in the subsequent section. We denote this definition as $lgg_{\leq}$ and in [5] we have proved that it returns not only a more general tree but also the least one, i.e. the following holds:

Proposition 1. *Let T_1 and T_2 be query trees and $T = lgg_{\leq}(T_1, T_2)$. Then the following results hold:*

1. *$T_1 \leq T$ and $T_2 \leq T$ (i.e. $lgg_{\leq}$ generalises)*
2. *for any tree T', we have $T_1 \leq T'$, $T_2 \leq T'$ implies $T \leq T'$ (i.e. $lgg_{\leq}$ is least)*

An example of the application of $lgg_{\leq}$ is shown in Figure 2.

[8] note that this looks currently overly complicated but will change in Section 3 with entailment enabled

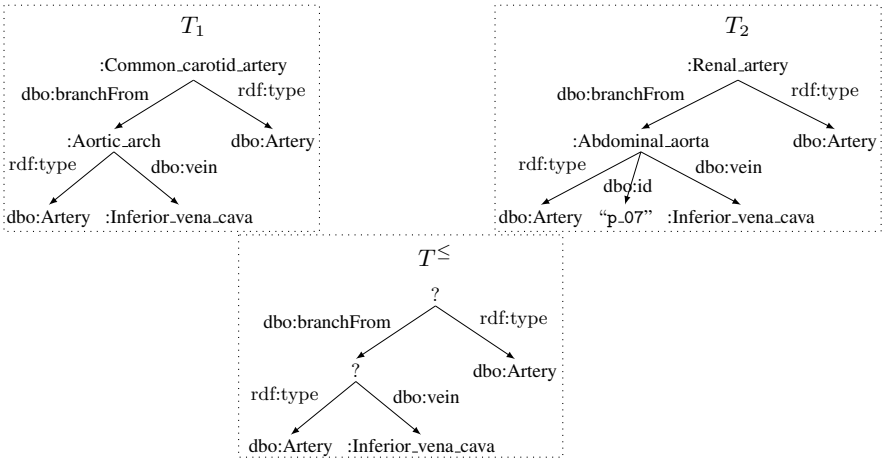


Fig. 2: Query tree $T^{\leq}$ is the $lgg_{\leq}$ of T_1 and T_2.

3 LGG under Entailment

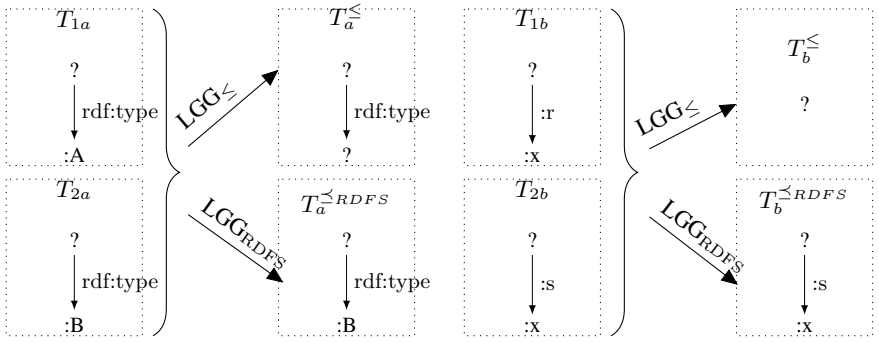
All the previously defined operations are based on the structure of the trees but without considering also the semantics we could lose information. Consider for instance the trees T_{1a} and T_{2a} in Figure 3a, with T_{1a} describing resources whose (direct) type is A and T_{2a} representing resources of type B, which is a subtype of A. The application of $lgg_{\leq}$ will result in tree $T_1^{\leq}$ which just describes resources that have any type. Another example is shown in Figure 3b. Both trees T_{1b} and T_{2b} represent resources that are related to the same resource x, but with different predicates r and s. Although it is known from the RDF graph G that r is a subproperty of s, the result of lgg will be $T_b^{\leq}$, a tree which basically describes anything. It is easy to see that with using only the structure for tree subsumption, we will lose information which can indeed be quite important.

To overcome such weaknesses, we extend the definition of tree subsumption to tree subsumption under entailment. With this we make use of so-called SPARQL entailment regimes[9], which were introduced in the SPARQL 1.1 standard. The idea is to not only use the explicitly given graph structures but semantic entailment relations for basic graph pattern matching.

Definition 5 (Query Tree Subsumption under Entailment). *Let T_1 and T_2 be projectable query trees and $\mathcal{E}$ be an entailment relation such as RDFS or OWL RL entailment. T_1 is $\mathcal{E}$-subsumed by T_2, denoted as $T_1 \preceq_{\mathcal{E}} T_2$, if we have $[[sparql(T_1)]]_G^{\mathcal{E}}(?x0) \subseteq [[sparql(T_2)]]_G^{\mathcal{E}}(?x0)$ for any RDF graph G.*

Many different types of entailments regimes have been defined over the past decade, with the most prominent among others being RDFS [8], pD* [9] or the more recent OWL2 RL[10] entailment. Each of those entailments has different properties w.r.t. expres-

[9] http://www.w3.org/TR/sparql11-entailment/
[10] http://www.w3.org/TR/sparql11-entailment/#OWL2RLDS



(:A rdfs:subClassOf :B) $\in G$ (:r rdfs:subPropertyOf :s) $\in G$

(a) rdfs:subClassOf entailment (b) rdfs:subPropertyOf entailment

Fig. 3: Examples of differences in the result of lgg when applied with and without RDFS entailment on an RDF graph G.

siveness and computational complexity, e.g. under RDFS entailment we can solve the problems exemplified in Figure 3 based on a decidable but NP-complete algorithm[11]. Especially for RDFS entailment, this complexity can be reduced by the restriction on a (minimal) subset of RDFS proposed in [10], which includes all the relevant RDFS keywords rdf:type, rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain, and rdfs:range. This fragment preserves the original RDFS semantics and allows RDFS inference by just checking the existence of some paths in the original graph, obtaining a good theoretical upper-bound on the cost of RDFS entailment.

Again, to avoid the evaluation of SPARQL queries, we extend the definition of structural query tree subsumption (Def. 4) to cover property and tree entailment.

Definition 6 (Structural Query Tree Subsumption under Entailment). *Let T_1 and T_2 be query trees. T_1 is structurally $\mathcal{E}$-subsumed by T_2, denoted as $T_1 \leq_{\mathcal{E}} T_2$, if*

1. $l(root(T_2)) =?$ *and*
   - *for each edge $(root(T_2), e_i, v_2) \in E(T_2)$ there is an edge $(root(T_1), e_j, v_1) \in E(T_1)$ such that $e_j \sqsubseteq_{\mathcal{E}} e_i$ and*
     (a) if $e_i = e_j =$ `rdf : type` and $typeOf(T(v_1)) \sqsubseteq_{\mathcal{E}} typeOf(T(v_2))$
     (b) otherwise, $T(v_1) \leq_{\mathcal{E}} T(v_2)$
2. *otherwise, if $\ell(root(T_1)) = \ell(root(T_2))$*

The following is a consequence of the definition of $\leq_{\mathcal{E}}$. It connects the structure of query trees with the semantics of SPARQL.

Proposition 2. *Let T_1 and T_2 be projectable query trees. $T_1 \leq_{\mathcal{E}} T_2$ implies $T_1 \preceq_{\mathcal{E}} T_2$.*

Proof. We prove Proposition 2 by induction over the depth of T_2. Let G be an RDF graph.

[11] Without blank nodes RDFS entailment is in P.

Induction Base ($depth(T_2) = 0$*)* :In this case, $[[sparql(T_2, ?x0)]]^{\mathcal{E}}_G(?x0)$ is the set of all resources occurring in subjects of triples in G and, therefore, $T_1 \preceq_{\mathcal{E}} T_2$. holds.

Induction Step ($depth(T_2) > 0$*)* : $sparql(T_1, ?x0)$ and $sparql(T_2, ?x0)$ have a basic graph pattern in their WHERE clause, i.e. a set of triple patterns. Due to the definition of $sparql$, the triple patterns with subject $?x0$ have one of the following forms:

1. $?x0\ ?y\ ?z$
2. $?x0\ p\ m$ with $m \in R \cup L$
3. $?x0\ p\ ?xi$

Each such pattern in a SPARQL query is a restriction on $?x0$. To prove the proposition, we show that for each such triple pattern in $sparql(T_2, ?x0)$, there is a triple pattern in $sparql(T_1, ?x0)$, which is a stronger restriction of $?x0$, i.e. leads to fewer results for $?x0$. We do this by case distinction:

1. $?x0\ ?y\ ?z$: The same pattern exists in $sparql(T_1, ?x0)$.
2. $?x0\ p\ m$ with $m \in R \cup L$: Thus, there is an edge $(\mathit{root}(T_2), p, v_2)$ with $\ell(v_2) = m$ in T_2. Because of the definition of $\leq_{\mathcal{E}}$, there is an edge $(\mathit{root}(T_1), p_i, v_1)$
   (a) with $p_i \sqsubseteq_{\mathcal{E}} p$ and $\ell(v_1) = m$ in T_1
   (b) with $p = p_i =$ `rdf:type` and $typeOf(v_1) \sqsubseteq_{\mathcal{E}} typeOf(v_2)$
   both of which are stronger restrictions on $?x0$ and result in fewer or equally many matches when evaluated via $[[]]^{\mathcal{E}}_G$
3. $?x0\ p\ ?xi$: This means that there is an edge $(\mathit{root}(T_2), p, v_2)$ with $\ell(v_2) = ?$ in T_2. Let $(\mathit{root}(T_1), p_1, v_1)$ with $p_1 \sqsubseteq_{\mathcal{E}} p$ and $\ell(v_1) = s$ be a corresponding edge in T_1 according to the definition of $\leq_{\mathcal{E}}$. We distinguish two cases:
   (a) $s \neq ?$. In this case, $sparql(T_1, ?x0)$ contains the pattern $?x0\ p_1\ s$, which is a stronger restriction on $?x0$ than $?x0\ p\ ?xi$.
   (b) $s = ?$. In this case, $sparql(T_1, ?x0)$ contains the pattern $?x0\ p_1\ ?xj$. By induction, we know $T(v_1) \leq_{\mathcal{E}} T(v_2)$ and, consequently, we know that $[[sparql(T(v_1), ?x0)]]^{\mathcal{E}}_G(?x0) \subseteq [[sparql(T(v_2), ?x0)]]^{\mathcal{E}}_G(?x0)$, i.e. the pattern is a stronger restriction on $?x0$, because there are fewer or equally many matches for $?xj$ than for $?xi$.

□

We can now define *lgg under entailment* $\mathcal{E}$ denoted as $lgg_{\mathcal{E}}$ by reusing Function lgg with its sub-functions revised as follows:

1. $\mathrm{relatedPredicates}(T_1, T_2)$ returns all triples of predicates $\langle p_i, p_j, p\rangle$ such that there is an edge $(\mathit{root}(T_1), p_i, v_1') \in E(T_1)$ and there is an edge $(\mathit{root}(T_2), p_j, v_2') \in E(T_2)$ and p_i, p_j are related by subsumption, i.e. $p_i \sqsubseteq_{\mathcal{E}} p_j$ (resp. $p_j \sqsubseteq_{\mathcal{E}} p_i$) and $p = p_j$(resp. $p = p_i$)
2. $\mathrm{isSubTreeOf}(T_1, T_2)$ returns true if $T_1 \leq_{\mathcal{E}} T_2$, otherwise false

Proposition 3. *Let* T_1 *and* T_2 *be projectable query trees,* $\mathcal{E}$ *an entailment relation and* $T = lgg_{\mathcal{E}}(T_1, T_2)$*. Then the following results hold:*

1. $T_1 \leq_{\mathcal{E}} T$ *and* $T_2 \leq_{\mathcal{E}} T$ *(i.e. lgg generalises)*
2. *for any tree* T'*, we have* $T_1 \leq_{\mathcal{E}} T'$*,* $T_2 \leq_{\mathcal{E}} T'$ *implies* $T \leq_{\mathcal{E}} T'$ *(i.e.* $lgg_{\mathcal{E}}$ *is least)*

Proof. 1.) We prove by induction over the depth of T. Without loss of generality, we show $T_1 \leq_{\mathcal{E}} T$.

Induction Base (depth$(T) = 0$*)* : If $\ell(root(T)) \neq ?$, then by Function lgg $\ell(root(T_1)) = \ell(root(T))$ (see also table below).

Induction Step (depth$(T) > 0$*)* : We have to show that for any edge $e = (root(T), p, v) \in E(T)$ there is at least one edge $e' = (root(T_1), p', v')$ such that $p' \sqsubseteq_{\mathcal{E}} p$ and $T(v') \leq_{\mathcal{E}} T(v)$. By Function lgg, $e = (root(T), p, root(\text{lgg}(T(v_1), T(v_2))))$ was created from two edges $(root(T_1), p_1, v_1) \in E(T_1)$ and $(root(T_2), p_2, v_2) \in E(T_2)$ such that either $p_1 \sqsubseteq_{\mathcal{E}} p_2$ and $p = p_2$ or $p_2 \sqsubseteq_{\mathcal{E}} p_1$ and $p = p_1$. If $\ell(v) \neq ?$ (Definition 6, condition 1), then $\ell(v_1) = \ell(v)$ by Line 2 of Function lgg. If $\ell(v) = ?$ (condition 2), then $T(v_1) \leq_{\mathcal{E}} T(v)$ follows by induction.

2.) We use induction over the depth of T.

Induction Base (depth$(T) = 0$*)* : We first show $depth(T') = 0$. By contradiction, assume that T' has at least one edge. Let p be the label of an outgoing edge from the root of T'. By Definition 6, both T_1 and T_2 must therefore also have outgoing edges from their respective root nodes such that they are $\mathcal{E}$-subsumed by p. Consequently, $T = \text{lgg}_{\mathcal{E}}(T_1, T_2)$ has an outgoing edge from its root by Function lgg (Lines 4-13 create at least one such edge). This contradicts $depth(T) = 0$.

We make a complete case distinction on root node labels of T_1 and T_2 (note that $m \neq ?$, $n \neq ?$):

$\ell(root(T_1))$	$\ell(root(T_2))$	$\ell(root(T))$ according to Function lgg
m	m	m
m	$n(\neq m)$	$?$
m	$?$	$?$
$?$	m	$?$
$?$	$?$	$?$

In Row 1, $\ell(root(T'))$ is either m or $?$, but in any case $T \leq_{\mathcal{E}} T'$. For Rows 2-5, $\ell(root(T')) = ?$, because otherwise $T_1 \not\leq_{\mathcal{E}} T'$ or $T_2 \not\leq_{\mathcal{E}} T'$. Again, we have $T \leq_{\mathcal{E}} T'$.

Induction Step (depth$(T) > 0$*)* : Again, we show $T \leq_{\mathcal{E}} T'$ using Definition 6:

Condition 1 : $? \neq \ell(root(T')) = \ell(root(T_1)) = \ell(root(T_2))$ (from $T_1 \leq_{\mathcal{E}} T'$ and $T_2 \leq_{\mathcal{E}} T'$) $= \ell(root(T))$ (from Function lgg)

Condition 2 : Let $(root(T'), p, v')$ be an edge in T'. Due to $T_1 \leq_{\mathcal{E}} T'$ and $T_2 \leq_{\mathcal{E}} T'$, there is an edge $(root(T_1), p_1, v_1)$ in T_1 and an edge $(root(T_2), p_2, v_2)$ in T_2 such that p subsumes p_1 and p_2.

2a): $? \neq \ell(v') = \ell(v_1)) = \ell(v_2)$ (from $T_1 \leq_{\mathcal{E}} T'$ and $T_2 \leq_{\mathcal{E}} T'$) $= \ell(v)$ (from Function lgg)

2b): Due to $\ell(v') = ?$, we get $T(v_1) \leq_{\mathcal{E}} T(v')$ and $T(v_2) \leq_{\mathcal{E}} T(v')$. Hence, we can deduce $T(v) = \text{lgg}(T(v_1), T(v_2)) \leq_{\mathcal{E}} T(v')$ by induction.

□

Referring to the two examples, when using RDFS entailment we can generate more specific trees by lgg_{RDFS}. For the example in Figure 3a we get $T_a \preceq_{RDFS}$ because of RDFS entailment rule rdfs9, which is defined as

xxx rdfs:subClassOf yyy .
zzz rdf:type xxx . $\Rightarrow$ zzz rdf:type yyy .

The example shown in Figure 3b would result in tree $T_b^{\preceq_{RDFS}}$ by RDFS entailment rule rdfs7, that is

aaa rdfs:subPropertyOf bbb .
xxx aaa yyy . $\Rightarrow$ xxx bbb yyy .

4 The QTL2 Algorithm

The Query Tree Learner (QTL2) integrates the formal foundations from Sections 2 and 3 into a light-weight learning algorithm. It is a supervised algorithm, i.e. the input are positive and (possibly) negative examples which in fact are RDF resources. The first step comprises the generation of query trees for all pos. and neg. examples as shown in Line 1 and 2. The mapping follows the definition in Sec. 2. It retrieves information about a resource from an RDF graph G by means of SPARQL CONSTRUCT queries and generates a query tree whose maximum depth is limited by the recursion depth parameter d.

input : RDF graph G, max. recursion depth d, objective function τ,
pos. examples $E^+ = \{e_1^+, \ldots, e_n^+\} \subset R$, $E^+ \neq \emptyset$,
neg. examples $E^- = \{e_1^-, \ldots, e_m^-\} \subset R$
output : Sorted set of solution trees S

1 $\mathcal{T}^+ = \{T_i^+ \mid \exists i.e_i^+ \in E^+, T_i^+ = \text{map}(G, d, e_i^+)\}$ // generate query trees
2 $\mathcal{T}^- = \{T_i^- \mid \exists i.e_i^- \in E^-, T_i^- = \text{map}(G, d, e_i^-)\}$
3 $Q = [\langle T^\theta, [T_1^+, \ldots, T_n^+], \tau_{min})\rangle]$ // init working queue
4 $S = \emptyset$
5 **while** *!terminationCriteriaSatisfied()* **do**
6 $q = Q.best()$ // pick best query tree from queue
7 **foreach** $T_i^+ \in q.posExampleTrees()$ **do**
8 $T_c = \texttt{lgg}(q.tree(), T_i^+)$ // compute LGG
9 $q_n = \texttt{evaluate}(T_c, \mathcal{T}^+, \mathcal{T}^-)$ // compute score of LGG
10 **if** $q_n \notin Q$ *and* $q_n \notin S$ **then** $Q.add(q_n)$
11 $S.add(q.tree())$
12 **return** S

Algorithm 1: QTL2 Algorithm.

In a next step a priority queue Q, is initialised with an identity element (Line 3). Each element in Q consists of a query tree T, the list of positive examples $T_1^+, \ldots, T_n^+$ which are not covered by the query tree as well as a score s for the query tree T and is denoted as $\langle T, [T_1^+, \ldots, T_n^+], s\rangle$. The identity element comprises a tree T^θ with $lgg(T^\theta, T) = T$,

all positive examples trees and the minimum score according to the chosen objective function τ.

The outer loop starts with the selection of the best element q in the search queue (Line 6). Afterwards, in the inner loop the lgg is computed between the tree represented by q and each of the uncovered positive example trees (Line 8). Each of those new trees is being evaluated by the function $\mathrm{evaluate}$, which returns a new element q_n that contains, besides the tree, also a score value based on τ and all remaining uncovered positive and negative examples. If q_n was not already processed it will by added to the working queue Q. The whole process is repeated until some stopping criteria have been satisfied, e.g. Q is empty or the maximum runtime has been exceeded. The final output of the algorithm is a ranked list of query trees.

Objective Functions: We use the following two objective functions in our algorithm:

F$_\beta$-score: $\frac{(1+\beta^2)\cdot tp}{(1+\beta^2)\cdot tp+\beta^2\cdot fp+fn}$

MCC[11]: $\frac{tp\cdot tn-fp\cdot fn}{\sqrt{(tp+fp)\cdot(tp+fn)\cdot(tn+fp)\cdot(tn+fn)}}$

where tp (fp) denotes the number of true positives (false positives), tn (fn) the number of true negatives (false negatives) and β is a weight used for balancing. We decided to support MCC which is supposed to perform better on classes of very different sizes which is the case here (negative examples are dominant). The algorithm can be configured to use arbitrary objective functions.

5 Evaluation

Research Questions We analysed the following research questions:

Q1 Is the algorithm robust against noise?
Q2 Does the algorithm have a bias towards high precision and/or recall?
Q3 How many examples does the algorithm need to provide accurate results?
Q4 What are limitations of the algorithm in terms of scalability?
Q5 How does the complexity of input questions relate to the computational effort required by the algorithm?

Experimental Setup In [5] SPARQL queries from the QALD challenge[12], which is a benchmark for Question Answering over Linked Data, have been utilized for evaluation. Those SPARQL queries have been used to retrieve positive examples that extensionally describe the concept encoded by the query. Unfortunately, none of the queries that belong to the supported language of QTL2 contain triple patterns connected via variables in subject and object position. While QTL2 can obviously obtain perfect F-score on QALD, the queries are overly simple for QTL2 and do not allow us to answer all research questions. Therefore, we generated another set of SPARQL queries based on DBpedia. The procedure was as follows: We started with the resources r_i of a class C contained in DBpedia. For those resources we were searching for sequences of predicates of length n such that

[12] http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/

1. there is a path $r_i \stackrel{p_1}{\rightarrow} o_1 \ldots o_{n-1} \stackrel{p_n}{\rightarrow} o_n$ in the RDF graph for at least x resources
2. and for at least two of those resources it holds that the intermediate o_i at the same position i in the path are different from each other.

An example of length 3 would be the following SPARQL query:

```
SELECT DISTINCT ?s WHERE {
  ?s a dbo:Racecourse .
  ?s dbo:location ?o0 .
  ?o0 dbo:country ?o1 .
  ?o1 dbo:governmentType :Unitary_state }
```

It describes all racecourses that are located in a country which is a unitary state. In that case, we were looking for at least two racecourses that do not locate at the same place and moreover, the places belong to different countries.

For our benchmark we used $n = \{1, 2\}$ and $x = 30$ and did this for 50 randomly chosen classes per each n, thus resulting in 100 queries total. The whole experimental setup as well as the queries are accessible online.[13]

Based on our research questions, we have three dimensions: (1) the number of input examples both for positive and negative ones (2) the noise value (see below) and (3) the chosen objective function. The number of examples varies between 5 and 30 while the noise interval is $[0.0, 0.3]$. As objective functions we used F-score and MCC. To simulate noisy data, we replaced as many positive input examples as the noise value denotes by randomly chosen resources from DBpedia such that they do not belong to the set of all possible positive examples as well as not to the negative input examples.

For each combination of the three dimensions, we run QTL2 with the given parameters, i.e. we performed 4800 runs each of it given a maximum execution time of 60 seconds. We enabled RDFS(minimal) reasoning and loaded the DBpedia ontology into an in-memory reasoner. Note that we generated the query trees in advance, which allows us to get better insights into the runtime behaviour of QTL2 independently of the underlying triple store performance. In a next step, we took the best returned tree and evaluated it against the whole DBpedia knowledge base by using the same objective function.

As baseline we used the best edge from the root node among all trees with regards to the objective function and also evaluated it against the whole DBpedia.

Results In Figure 4, we show the behaviour of QTL2 under different values of noise when the number of examples is fixed. For different objective functions, we can observe how increasing noise reduces the value of the objective functions of the best query learned by QTL2. Moreover, we can obtain information how quickly in terms of input examples the algorithm converges to good scores. The baseline performs almost constant having an F-score of about 0.4 and usually tends to a high recall but low precision, e.g. for 30 examples the avg. recall is between 0.82 and 0.97 while precision is approx. 0.4.

Regarding runtime analysis, on average the best solution was already found in less than one second, whereas the maximum runtime was approximately 20 seconds.

[13] Experimental data: http://dl-learner.org/QTL/ESWC2016

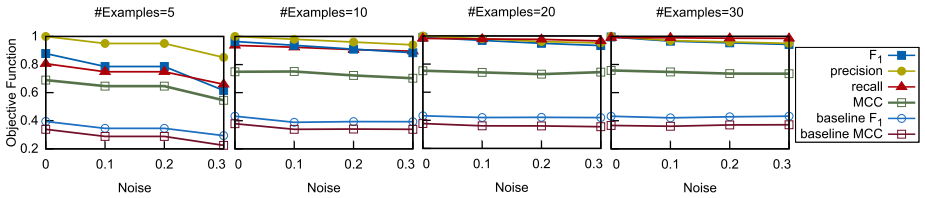


Fig. 4: Comparison of objective functions vs. noise for different number of examples.

Discussion

Q1 Yes, based on the experiments even with 30% of the input being noisy, the results look still promising.

Q2 The nature of the algorithm is that it tends to focus more on precision compared to recall. This can be influenced by configuring the objective function. QTL2 achieves low recall in cases when the target query contains very many instances and the example set only reflects one particular aspect of those instances (e.g. when aiming to find a concept such as "movie stars in top cinema productions", QTL2 prefers high precision over recall and learns more specific concepts such as "male movie stars in top cinema productions based in the United States").

Q3 The results of the experiments have shown that a small number of examples (approx. 10) is sufficient to get F_1 scores above 90% even in noisy conditions. Without noise, the F_1 score is close to 1 for 20 or more examples.

Q4 The performance of the algorithm is influenced by the size of query trees as well as the depth of the query trees. For datasets with high outdegree, the depth needs to be limited appropriately (depth 2 for DBpedia works efficiently).

Q5 The complexity of the target question in the experiments does not significantly influence the algorithm. The lgg based approach allows to learn very complex queries using few input examples as it considers the full query tree up to some depth. This is a significant advantage compared to other induction strategies, such as refinement operator based approaches, which gradually build up more complex hypothesis.

6 Related Work

Strongly related work was published in [5] and defines the least general generalisation on RDF query trees. While this work does not take into account any kind of semantic entailment and does not consider robustness against noise, we used key concepts of it as basis of our work. Another closely related approach was published in [12] and defines common subsumers on rooted RDF graphs. Compared to our work, (1) their approach does not necessarily return the least common subsumer and (2) again they do not take any semantics into account. On the other hand their approach is able to generalise even over the predicates. In [13], the problem of approximate querying of RDF databases by means of query relaxation is studied, i.e. queries that are generated and executed that are to some extend similar to the user query. [14] describes graph kernels for RDF, which

makes it easy to use state-of-the art machine learning techniques like SVMs and result in promising predictive performance.

Apart from RDF graphs, computing the least general generalisation plays also an important role in Description Logics, although it is more a non-standard inference service. Among others, the work in [15,16,17] has proposed algorithms and shown its feasibility to compute least general generalisations even in expressive Description Logics like $\mathcal{EL}$.

More generally, our approach is related to Inductive Logic Programming [18] where the goal is to learn a hypothesis, usually a clause or logic program, from examples and background knowledge. It was most widely applied for learning horn clauses, but also in the Semantic Web context based on OWL and description logics [19,20,21,22,23] with predecessors in the early 90s [24]. Those approaches use various techniques like inverse resolution, inverse entailment and commonly refinement operators. Least general generalisation, as we used here, is one of those techniques. It has favourable properties in our context, because it is very suitable for learning from a low number of examples. This is mainly due to the fact, that lgg allows to make large leaps through the search space in contrast to gradual refinement. More generally, generate-and-test procedures are often less efficient then test-incorporation as pointed out in an article about the ProGolem system [25], which has influenced the design of our system. Drawbacks of lggs usually arise when expressive target languages are used and the input data is very noisy. The latter is usually not a problem in QTL2, because examples are manually confirmed and can be revised during the learning process. As for the expressiveness of the target language, we carefully selected a fragment of SPARQL, where lggs exist and can be efficiently computed.

7 Conclusions

In this paper, we have presented the first machine learning algorithm for SPARQL queries that does not assume noise-free input and supports inference. The algorithm is based on the notion of query trees that correspond to a fragment of SPARQL. We extended this formalism to be inference-aware and provided definitions and algorithms for subsumption and least general generalisation on top of it. This was integrated in an algorithm that works efficiently by reducing computational operations to in-memory operations after an initial preprocessing phase. In an evaluation on challenging queries against a large dataset, we have shown that high precision and recall values can be obtained. We also identified limitations, e.g. the algorithm is not targeted to large numbers of input examples. This is usually not the case in its main application context in question answering. We have shown that the algorithm is robust and noise-tolerant and can, therefore, overcome the brittleness of current RDF question answering systems.

References

1. Unger, C., Bühmann, L., Lehmann, J., Ngonga Ngomo, A.C., Gerber, D., Cimiano, P.: Template-based Question Answering over RDF data. In: Proceedings of the 21st international conference on World Wide Web. (2012) 639–648

2. Bernstein, A., Kaufmann, E., Kaiser, C., Kiefer, C.: Ginseng: A guided input natural language search engine for querying ontologies. In: 2006 Jena User Conference, Bristol, UK. (2006)
3. Lopez, V., Fernández, M., Motta, E., Stieler, N.: Poweraqua: Supporting users in querying and exploring the semantic web. Semantic Web **3**(3) (2012) 249–265
4. Lehmann, J., Bizer, C., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia - a crystallization point for the web of data. Journal of Web Semantics **7**(3) (2009) 154–165
5. Lehmann, J., Bühmann, L.: AutoSPARQL: Let users query your knowledge base. In: Proceedings of ESWC 2011. (2011)
6. Angles, R., Gutierrez, C.: The expressive power of SPARQL. In Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T.W., Thirunarayan, K., eds.: ISWC 2008 Proceedings. Volume 5318 of LNCS., Springer (2008) 114–129
7. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In Cruz, I.F., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L., eds.: ISWC 2006. Volume 4273 of LNCS., Springer (2006) 30–43
8. Hayes, P.: RDF Semantics. W3C Recommendation (2004)
9. ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the owl vocabulary. J. Web Sem. **3**(2-3) (2005) 79–115
10. Muñoz, S., Pérez, J., Gutierrez, C.: Simple and efficient minimal RDFS. Journal of Web Semantics **7**(3) (September 2009) 220–234
11. Matthews, B.: Comparison of the predicted and observed secondary structure of {T4} phage lysozyme. Biochimica et Biophysica Acta (BBA) - Protein Structure **405**(2) (1975) 442 – 451
12. Colucci, S., Donini, F., Di Sciascio, E.: Common subsumbers in RDF. In Baldoni, M., Baroglio, C., Boella, G., Micalizio, R., eds.: AI*IA 2013: Advances in Artificial Intelligence. Volume 8249 of Lecture Notes in Computer Science. Springer International Publishing (2013) 348–359
13. Huang, H., Liu, C., Zhou, X.: Approximating query answering on RDF databases. World Wide Web **15**(1) (2012) 89–114
14. Lösch, U., Bloehdorn, S., Rettinger, A.: Graph kernels for RDF data. In Simperl, E., Cimiano, P., Polleres, A., Corcho, O., Presutti, V., eds.: The Semantic Web: Research and Applications. Volume 7295 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 134–148
15. Baader, F., Küsters, R.: Least common subsumer computation w.r.t. cyclic ALN-terminologies. In Franconi, E., Giacomo, G.D., MacGregor, R.M., Nutt, W., Welty, C.A., eds.: Description Logics. Volume 11 of CEUR Workshop Proceedings., CEUR-WS.org (1998)
16. Baader, F.: Computing the least common subsumer in the Description Logic EL w.r.t. terminological cycles with descriptive semantics. In de Moor, A., Lex, W., Ganter, B., eds.: ICCS. Volume 2746 of Lecture Notes in Computer Science., Springer (2003) 117–130
17. Baader, F.: A graph-theoretic generalization of the least common subsumer and the most specific concept in the Description Logic EL. In: WG. Volume 3353 of Lecture Notes in Computer Science., Springer (2004) 177–188
18. Nienhuys-Cheng, S.H., de Wolf, R., eds.: Foundations of Inductive Logic Programming. Volume 1228 of Lecture Notes in Computer Science. Springer (1997)
19. Fanizzi, N., d'Amato, C., Esposito, F.: DL-FOIL concept learning in Description Logics. In: Proceedings of the 18th International Conference on Inductive Logic Programming. Volume 5194 of LNCS., Springer (2008) 107–121
20. Lehmann, J., Hitzler, P.: Concept learning in description logics using refinement operators. Machine Learning journal **78**(1-2) (2010) 203–250
21. Lehmann, J., Haase, C.: Ideal downward refinement in the EL description logic. In: Inductive Logic Programming, 19th International Conference, ILP 2009, Leuven, Belgium. (2009)

22. Iannone, L., Palmisano, I., Fanizzi, N.: An algorithm based on counterfactuals for concept learning in the Semantic Web. Applied Intelligence **26**(2) (2007) 139–159
23. Cumby, C.M., Roth, D.: Learning with feature description logics. In: Proc. of the 12th Int. Conf. on Inductive Logic Programming. Volume 2583 of LNAI., Springer (2003) 32–47
24. Kietz, J.U., Morik, K.: A polynomial approach to the constructive induction of structural knowledge. Machine Learning **14** (1994) 193–217
25. Muggleton, S., Santos, J.C.A., Tamaddoni-Nezhad, A.: Progolem: A system based on relative minimal generalisation. In: ILP. Volume 5989 of LNCS., Springer (2009) 131–148