Computing and Informatics, Vol. 38, 2019, 679–700, doi: 10.31577/cai_2019_3_679

LEARNING SPARQL QUERIES FROM EXPECTED RESULTS

Jedrzej Potoniec

Faculty of Computing
Poznan University of Technology
ul. Piotrowo 3
60-965 Poznan, Poland
e-mail: jpotoniec@cs.put.poznan.pl

Abstract. We present LSQ, an algorithm learning SPARQL queries from a subset of expected results. The algorithm leverages grouping, aggregates and inline values of SPARQL 1.1 in order to move most of the complex computations to a SPARQL endpoint. It operates by building and testing hypotheses expressed as SPARQL queries and uses active learning to collect a small number of learning examples from the user. We provide an open-source implementation and an on-line interface to test the algorithm. In the experimental evaluation, we use real queries posed in the past to the official DBpedia SPARQL endpoint, and we show that the algorithm is able to learn them, 82 % of them in less than a minute and asking the user just once.

Keywords: SPARQL, RDF, active learning

1 INTRODUCTION

Querying information with a complex structure is an inherently hard task for a user. The user must spend a lot of time learning a vocabulary and relations used in the data. This will become more and more important, as we develop more complex artificial intelligence systems, using vast amount of information encoded in a complex representation formalism. In this paper, we aim to remedy this issue in the context of information represented as an Resource Description Framework (RDF) graph.

Consider the following use case scenario: a user has some informal criteria to select a subset of nodes of a graph. He/she knows some of the relevant nodes in the graph, he/she also knows some of the irrelevant nodes. Moreover, he/she can

distinguish a relevant node from an irrelevant one for some price, e.g. by spending his/her time verifying if a node is a relevant one or not. He/she wants to obtain a formal query corresponding to the informal criteria to be able to query the graph.

To address this use case, we propose an algorithm for learning a SPARQL query corresponding to these criteria. The algorithm is designed in such a way that it moves most of the computational work to a SPARQL endpoint by posing quite complex queries to it. It is a reasonable decision: the RDF graph is stored there, so that is the best place to perform any optimization.

Our contribution is as follows: we present the first algorithm for constructing SPARQL queries from examples, which is *at the same time interactive and saves computational power of the client*, by moving most of the computations to the SPARQL server.

The rest of the paper is organized as follows: in Section 2 we present a short overview of the most important aspects of RDF and SPARQL. Section 3 discusses related research. The description of the algorithm, along with the required definitions, is presented in Section 4, and in Section 5 we introduce a web application implementing the presented algorithm. Section 6 presents an experimental evaluation of the algorithm in two setups:

1. using a set of real-world SPARQL queries as a gold standard;
2. using a benchmark for the task of binary classification in the structured machine learning.

We conclude in Section 7.

2 PRELIMINARIES

2.1 Resource Description Framework

Resource Description Framework (RDF) is a framework designed to represent information in the Web in a way accessible for machines [33]. The core concept of RDF is an *RDF triple*, which consists of a *subject*, a *predicate* and an *object*. A customary meaning assigned to a triple is such that the entity represented by the subject is in relation denoted by the predicate with the entity represented by the object.

A set of triples constitutes an *RDF graph*, where subjects and objects jointly form the set of *nodes* of the graph and each triple represents a directed edge from the subject of the triple to the object, labeled with the predicate.

An *RDF term* is either an IRI (Internationalized Resource Identifier), that serves as a global identifier for some entity in the universe of discourse; a blank node, that is a local identifier for some entity in the universe of discourse; a literal, that represents a concrete value such as a string of characters or a number. Usually, the non-unique name assumption is made, stating that a single entity may be referenced by multiple identifiers. The subject of a triple may be either an IRI or a blank node, the predicate must be an IRI and the object may be an arbitrary RDF term.

In formalizing RDF we follow [22] and we start by introducing three pairwise disjoint sets: the set of all IRIs $\mathbf{I}$, the set of all blank nodes $\mathbf{B}$ and the set of all literals $\mathbf{L}$. A subject of a triple is any element of the set $\mathbf{I} \cup \mathbf{B}$, the predicate of a triple is an element of the set $\mathbf{I}$, while object is any element of the set $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$. An RDF graph G, being a set of triples, is thus a subset of a Cartesian product: $G \subseteq (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$.

Throughout this work, we use a subset of the Turtle syntax [7], where each triple is written in the order subject, predicate, object and ends with a dot. We represent IRIs using the prefix notation well-known from XML, i.e., `prefix:localName`, and list the common prefixes used throughout this work in Table 1. A literal is represented in quotation marks, with its datatype after `^^`. For example, the triple `dbr:Warsaw dbo:populationTotal "1740119"^^xsd:integer.` states that an entity denoted by the IRI `dbr:Warsaw` (in full: `http://dbpedia.org/resource/Warsaw`) has a total population of 1 740 119 citizens.

Prefix	Corresponding IRI
`dbr:`	`http://dbpedia.org/resource/`
`dbo:`	`http://dbpedia.org/ontology/`
`dbp:`	`http://dbpedia.org/property/`
`dct:`	`http://purl.org/dc/terms/`
`xsd:`	`http://www.w3.org/2001/XMLSchema#`

Table 1. The common IRI prefixes used in the paper and their corresponding IRIs

One of the most prominent applications of RDF is the Web of Data (also called Linked Data)[1], a large, distributed collection of RDF graphs concerning different topics from life sciences, to media, to governmental data. In the center of the Web of Data is DBpedia[2], the result of a complex knowledge extraction process from Wikipedia [1, 3, 20].

2.2 SPARQL Query Language

Every form of information representation requires a querying language and RDF is not an exception here. SPARQL Query Language is by far the most popular query language for RDF, built around graph pattern matching, yet having a lot in common with SQL known from relational databases [14]. Below, we summarize the most important aspects of SPARQL.

A SPARQL SELECT query is of form

```
SELECT head
WHERE { pattern }
[GROUP BY variables]
```

[1] `http://lod-cloud.net/`

[2] `http://dbpedia.org`

```
[HAVING criterion]
[LIMIT limit]
```

where the square brackets denote optional parts of the query. Every time the pattern is matched against the queried RDF graph, it yields a mapping from variables of the query to the RDF terms.

A *variable* in SPARQL is prefixed with a question mark, e.g. `?var`. A *blank node* `[]` denotes an anonymous, existentially quantified variable. A *triple pattern* is an RDF triple with an arbitrary number of components replaced by variables, e.g. `dbr:Warsaw dbo:populationTotal ?population.`. A *Basic Graph Pattern* (BGP) is a set of triple patterns that are matched jointly against the queried graph.

Denote by $\mathbf{V}$ the set of all variables. Let G be an RDF graph, P a BGP and denote by $vars(P)$ the set of all variables in P. We consider a mapping, i.e., partial function $\mu\colon \mathbf{V} \mapsto \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$. We say that μ is a *solution* to P if the domain of μ is the set $vars(P)$ and there exists a function $\sigma\colon \mathbf{B} \mapsto \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$ such that $\mu(\sigma(P)) \subseteq G$. We abuse the notation and by applying μ and σ to P we understand applying it element-wise to each part of each triple pattern in P.

A BGP may be optionally extended with a filter expression `FILTER(condition)`, to the effect that if μ(`condition`) evaluates to false, then μ is rejected. An alternative of two BGPs may be realized using the UNION keyword: `bgp1 UNION bgp2`, and we say that μ is a solution for it if it is a solution to either one of the BGPs. Finally, the clause `VALUES ?var {allowed assignments}` limits the set of allowed assignments for the variable `?var` in the solutions to the RDF terms listed in the curly braces.

The multi-set of solutions μ can be represented as a table, which can be then processed according to the standard SQL semantics of GROUP BY, HAVING and LIMIT. We thus abstain from formally defining their semantics and refer the interested reader to [14].

SPARQL is frequently employed with the SPARQL Protocol, which defines means to use HTTP (Hypertext Transfer Protocol) to query an RDF graph [32]. A server capable of answering SPARQL queries posed using SPARQL Protocol is called a *SPARQL endpoint* and is identified by its URL.

3 RELATED WORK

For the past few years, researchers proposed many approaches for querying an RDF graph alternative to writing a formal query by hand. Roughly, they can be divided into a few categories: faceted browsing, natural language interfaces, visual interfaces and recommendations. In faceted browsing a user is presented with an interface dynamically generated from an RDF graph to filter the graph according to his/her needs, e.g. see [23]. Also variants tailored to a specific types of data were proposed, e.g. [21] describes an interface for geospatial data. A recent system *Sparklis* combines faceted browsing with natural language generation to enable a non-expert user to query an RDF graph [11].

Natural language interfaces are concerned with answering a query specified in a natural language, e.g. by translating it to SPARQL and then posing it to an endpoint. These systems are frequently tailored to some specific task and/or a specific RDF graph. For example, *Xser* [34] is a system for answering factual questions and *CubeQA* [15] is designed to answer statistical queries requiring aggregate functions. A recent survey [16] gives a comprehensive overview of such systems.

The visual interfaces offer a possibility of constructing a query by visual means, e.g. by navigating over a displayed part of a graph [9] or by constructing a SPARQL query from building blocks [4].

A system using recommendations still requires from a user knowledge of SPARQL, but it helps to deal with an unknown vocabulary. [13] recommends predicate names based on an inferred type of a variable in a triple pattern. [6] describes a method for recommending query terms based on a graph summary, while [5] pushes it even further by performing the recommendations on-line, by posing appropriate queries to a SPARQL endpoint.

A similar idea to ours was already proposed in [19]. The main difference are the assumptions: [19] performs all the computation on the client-side, obtaining information about resources using an approach similar to Concise Bounded Description [27] and then computing their intersections. Our approach leverages new features of SPARQL 1.1, namely grouping, aggregates and providing inline values, and thus moves most of the learning complexity directly to a SPARQL endpoint.

Methods for learning queries from a set of examples were also discussed in other contexts. [17] uses a pattern mining approach to learn a set of SPARQL queries, which then are used as binary patterns for a normal classification algorithm. In [24] the authors propose a method for unsupervised mining of data mining features, which directly correspond to SPARQL queries. [18] and [10] are methods for learning Description Logics class expressions from a set of positive and negative examples. The expressions can then be used as queries to an ontological knowledge base to retrieve individuals fulfilling the class expression, e.g. using the *DL Query* tab of *Protégé* [12].

4 LEARNING SPARQL QUERIES

4.1 Basic Concepts

Throughout this section, we use the following example: the user wants to formulate a SPARQL query which allows him/her to query DBpedia for the capitals of states of the European Union. He/she knows some of them: Warsaw, Berlin, Zagreb, Nicosia and Vilnius. He/she can also recognize whether an arbitrary DBpedia IRI refers to one of the capitals by reading a Wikipedia article corresponding to the IRI. Of course, reading consumes his/her time, so he/she wants to limit the number of articles read. He/she also knows that Oslo, a capital of Norway, which is not a European Union member, should not be presented in the results.

Generally speaking, LSQ works by formulating a hypothesis and verifying it with the user.

Definition 1 (hypothesis)**.** A *hypothesis*, denoted $H(?iri)$, consists of SPARQL triple patterns and filters. Every triple pattern in the hypothesis has a fixed predicate (i.e., the predicate is an IRI), the subject and object may be an IRI, a literal or a variable. Every filter contains only an expression of form *variable* `>=` *fixed literal* or *variable* `<=` *fixed literal*. The hypothesis may contain an unlimited number of variables, but there is a single, distinguished variable $?iri$, which must be present in at least one triple pattern. We require the undirected graph corresponding to the basic graph pattern defined by a hypothesis to be a connected graph.

An *empty hypothesis* is a hypothesis which contains no triple patterns or filters.

For example, the algorithm may generate the following hypothesis `H(?iri)`:

```
dbr:European_Union dbo:wikiPageWikiLink ?iri .
?iri dct:subject dbr:Category:Capitals_in_Europe .
dbr:Member_state_of_the_European_Union dbo:wikiPageWikiLink ?iri.
```

It may be that the user already has some SPARQL query and wants to extend it by providing examples. We allow the user to provide a BGP $U(?iri)$, such that it shares the distinguished variable $?iri$ with the hypothesis, but all the remaining variables are separate. Some of the variables from $U(?iri)$ may be also present in the head of the final query obtained from the algorithm. We observe that if the user does not know SPARQL or does not have any query to extend, it is sufficient to assume that $U(?iri) = \emptyset$.

Definition 2 (query corresponding to a hypothesis)**.** A *query corresponding to a hypothesis* $H(?iri)$ is a SPARQL SELECT query containing the variable $?iri$ in the head, optionally with other variables coming from $U(?iri)$. The WHERE clause contains the hypothesis $H(?iri)$ and the user-provided BGP $U(?iri)$.

To formulate a hypothesis, LSQ uses a set of positive examples `P` and a set of negative examples `N`. Both sets contain IRIs from the RDF graph. `P` is a subset of IRIs expected in the results of the query corresponding to the final hypothesis. `N` is a set of IRIs which are forbidden to appear in these results. They do not need to be fully specified before the algorithm is run, instead it is enough that the user provides a few IRIs in both sets, and then they are extended during the execution of the algorithm. By `n_pos` we denote for the number of IRIs in the set `P`. In our running example, `P` initially consists of `dbr:Warsaw`, `dbr:Berlin`, `dbr:Zagreb`, `dbr:Nicosia` and `dbr:Vilnius`, and thus `n_pos` $= 5$. `N` contains only `dbr:Oslo`.

The algorithm uses well-known information retrieval measures. Denote by TP the number of IRIs from the set `P` which were retrieved by the query corresponding to a hypothesis, and by FP the number of IRIs from the set `N` which were retrieved by the query corresponding to the hypothesis. Following [2], we define three measures.

Definition 3 (precision)**.** *Precision* is the fraction of the number of the positive IRIs retrieved by the query corresponding to the hypothesis over the number of both positive and negative IRIs retrieved by the query corresponding to the hypothesis

$$p = \frac{TP}{TP + FP}.$$

Definition 4 (recall)**.** *Recall* is the fraction of the number of the positive IRIs retrieved by the query corresponding to the hypothesis over the number of known positive IRIs `n_pos`

$$r = \frac{TP}{\texttt{n\_pos}}.$$

Definition 5 (F_1 measure)**.** F_1 *measure* is the harmonic mean of precision and recall

$$F_1 = \frac{2}{\frac{1}{p} + \frac{1}{r}}.$$

4.2 Overview of the Algorithm

The flowchart of LSQ is presented in Figure 1 and its pseudocode in Algorithm 1. In the beginning, the user is asked to provide a few positive and a few negative examples. The hypothesis considered by the algorithm is set to an empty hypothesis. The hypothesis is then refined, as described in Section 4.5. Then, the algorithm generates a few examples that follow the hypothesis and a few examples that contradict the hypothesis, and the user is asked to assign them to one of the two sets. If any of the examples following the hypothesis is assigned to the set `N`, it means that the hypothesis is invalid and the last refinement is retracted. If the hypothesis is good enough w.r.t. the known examples (c.f. Section 4.3), it is presented to the user along with the full result of posing its corresponding query to the SPARQL endpoint. The user then must either accept the hypothesis or add at least one new positive or negative example, e.g. by selecting an IRI from the result which should not be there, or by adding an IRI which is missing. If the hypothesis is not good enough or the user adds a new example, the algorithm goes back to generating a new refinement.

4.3 Measuring Quality of a Hypothesis

To compute with a SPARQL endpoint how good a hypothesis is, the query presented in Listing 1 is used. Such a query computes the measures described above. The variable `?s` (resp. `?t`) is mapped to all IRIs from the set `P` (resp. `N`) that follow the hypothesis, thus `?tp` corresponds to TP, `?fp` to FP and so on. If the value of F_1 measure is high enough, the hypothesis is *good enough* w.r.t. to the examples available to the algorithm, and it is presented to the user as the final answer.

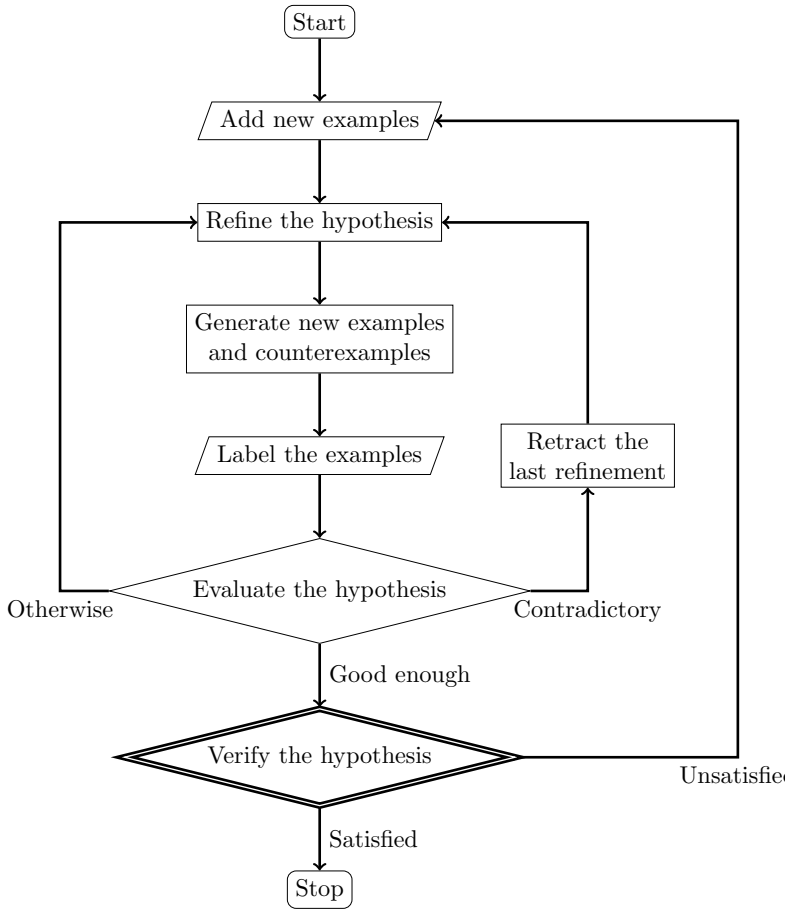

Figure 1. The flowchart of LSQ. A trapezoid denotes an input from the user, a rectangle denotes computations, a single-edged rhombus denotes a decision made by the algorithm and a double-edged rhombus denotes a decision made by the user.

4.4 Generating New Examples and Querying the User

If the hypothesis is not good enough, the algorithm should gather more evidence, to either confirm it or to reject it. To do so, the algorithm must generate a small set of examples that follow the hypothesis and another small set of examples that contradicts the hypothesis. To generate examples following the hypothesis, the query presented in Listing 2 is used. `H(?iri)` in the query ensures that an example follows the hypothesis, and `FILTER` ensures that it is a new example, i.e. one that is not present in either of the sets `P` and `N`. `LIMIT 3` is to ensure that we query the user only about a small number of new examples. Recall the example

```
h ← empty hypothesis;
fn ← 0;
while h is not good enough do
    if fn > 0 then h.pop_back();
    ready_to_query ← false;
    while not ready_to_query do
        while |h| > 0 do
            p, n ← generate new examples;
            if |p| > 0 and |n| > 0 then break;
            else h.pop_back();
        end
        cand ← ∅;
        foreach var ∈ variables(h) do
            cand ← cand ∪ refinements for variable var (cf. Section 4.5);
        end
        sort cand according to the value of precision;
        foreach c ∈ cand do
            h.push_back(c);
            p, n ← generate new examples;
            if h is good enough or (|p| > 0 and |n| > 0) then
                ready_to_query ← true;
                break;
            end
            h.pop_back();
        end
    end
    ask the user to label examples in p and n;
    fn ← number of examples from p that user labeled as negative;
end
```

Algorithm 1: The pseudo-code of the LSQ algorithm. Deciding whether the hypothesis is good enough is described in details in Section 4.3 and generating new examples in Section 4.4. Function pop_back removes the last element of the array, while push_back extends the array with its argument.

and let `H(?iri)` be `dbr:European_Union dbo:wikiPageWikiLink ?iri.` Possible new examples are `dbr:Above_mean_sea_level`, `dbr:Afroasiatic_languages`, `dbr:Andorra`, neither of them following the criteria we aim for since the hypothesis is too broad.

Generating negative examples is more difficult. Let `Hp(?iri)` be a hypothesis the hypothesis `H(?iri)` directly originated from, i.e. `Hp(?iri)` is the hypothesis `H(?iri)` without the last refinement. Consider the SPARQL query presented in Listing 3. The only common variable between `Hp(?iri)` and `H(?iri)` is `?iri`, the

```
SELECT (COUNT(DISTINCT ?s) as ?tp)
    (COUNT(DISTINCT ?t) AS ?fp)
    (?tp/(?tp+?fp) AS ?precision)
    (?tp/n_pos AS ?recall)
    (2/((1/?precision)+(1/?recall)) AS ?f1)
WHERE {{
        U(?s)
        H(?s)
        VALUES ?s { P }
    } UNION {
        U(?t)
        H(?t)
        VALUES ?t { T }
    }}
```

Listing 1. A query used to compute how good the hypothesis `H(?s)`

```
SELECT DISTINCT ?iri
WHERE {
    U(?iri)
    H(?iri)
    FILTER(?iri NOT IN (P N))
} LIMIT 3
```

Listing 2. A query used to generate new examples following the hypothesis `H(?iri)`

rest is uniquely renamed. The renaming is such that no variable except `?iri` is shared with the user-provided BGP `U(?iri)`. This query provides a set of examples that follow the hypothesis except for the last refinement. `FILTER` and `LIMIT` are used for the same purpose as before. The process of generating new examples follows the idea of active learning, where a learning algorithm selects unknown examples to

```
SELECT DISTINCT ?iri
WHERE {
    U(?iri) {
        { Hp(?iri) }
        MINUS
        { H(?iri) }
    } FILTER(?iri NOT IN (P N))
} LIMIT 3
```

Listing 3. A query used to generate new negative examples following the hypothesis `H(?iri)`, where `Hp(?iri)` is the previous hypothesis

```
SELECT ?p ?o (COUNT(DISTINCT ?s) AS ?tp)
    (COUNT(DISTINCT ?t) AS ?fp)
    (?tp/(?tp+?fp) AS ?precision)
    (?tp/n_pos AS ?recall)
    (2/((1/?precision)+(1/?recall)) AS ?f1)
WHERE {{
        U(?s)
        H(?s)
        ?s ?p ?o .
        VALUES ?s { P }
    } UNION {
        U(?t)
        H(?t)
        ?t ?p ?o .
        VALUES ?t { N }
    }}
GROUP BY ?p ?o
HAVING (?recall>=.99)
```

Listing 4. A query used to compute the set of possible refinements of the hypothesis `H(?iri)`, that have a fixed predicate and object

maximize benefit from getting the correct labels for them, i.e., to remove as much uncertainty as possible [8].

After the examples are generated, they are presented to the user. He/she must then assign each of them either to the set `P` or `N`. After the assignment is done, the algorithm continues, as described in Section 4.2.

4.5 Hypothesis Refinement

If the hypothesis is not good enough, the algorithm must refine it. First, the algorithm checks whether it is possible to generate new examples using the current hypothesis. If it is not, the most recent refinement of the hypothesis is retracted and the condition is checked again. The process continues until it becomes possible to generate new examples. In the worst case, it means emptying the hypothesis.

To refine the hypothesis, the algorithm must generate a set of possible refinements and select the best of them. First, consider a refinement consisting of a single triple pattern with a fixed predicate and object, and a subject being a variable already present in the hypothesis. For example, such a refinement could be `?iri dct:subject dbr:Category:Capitals_in_Europe`. To generate a set of such refinements the query presented in Listing 4 is used. Observe that the results are grouped by the pair `?p ?o`, so effectively what this query does is to compute the measures for a lot of hypotheses at once, each consisting of the original hypothesis H and

a refinement with the subject being a variable already present in the hypothesis (`?s` and `?t`) and fixed values for the predicate and the object. We require for the recall to reach at least 0.99, to ensure that all known positive examples are covered by a new hypothesis.

Recall the example, and let `H(?iri)` be

```
dbr:European_Union dbo:wikiPageWikiLink ?iri.
```

Consider the refinement

```
?iri dct:subject dbr:Category:Capitals_in_Europe.
```

Its recall is $r = 1$, as it matches all five elements of the set `P`, and its precision is $p = 1$ as it does not match the negative example, i.e., Oslo. On the other hand, the refinement `?iri dbo:utcOffset "+2"` will not be considered, as its recall is 0.8 (`dbr:Berlin` does not occur in such a triple).

To compute refinements with a fixed subject, but a variable object, the algorithm uses the same query, but replaces `?s ?p ?o` (resp. `?t ?p ?o`) with `?o ?p ?s` (resp. `?o ?p ?t`). Finally, to compute refinements with a fixed predicate, a new variable as the object (resp. subject) and an existing variable as the subject (resp. object), the query with `?o` replaced with a blank node `[]` in the triple patterns and without `?o` in the head and in the GROUP BY clause is used.

A more elaborate query, presented in Listing 5, is required to provide refinements with FILTER. The query operates in two steps. First, the subquery extracts all the pairs consisting of a predicate `?p` and a literal `?l` for the set `P`. Then, for every pair `?p ?l` it computes the measures for a hypothesis consisting of the original hypothesis `H(?s)`, a triple pattern `?s ?p ?xl` (`?xl` is a new variable), and a filter comparing the new variable `?xl` to a literal `?l` from the subquery. Again, we require the recall to be at least 0.99 to ensure appropriate coverage of positive examples. A sample refinement obtained this way is

```
?iri dbo:populationTotal ?var.
FILTER(?var >= "205934"^^ xsd:nonNegativeInteger).
```

When the set of the possible refinements is collected from the endpoint, the algorithm must choose the right one. The set of refinements is sorted according to the descending values of the F_1 measure and (in case of ties on F_1) the precision. For every refinement, the algorithm checks if the current hypothesis with the refinement added is good enough. If it is, the hypothesis is displayed to the user, as described in Section 4.2. Otherwise, the algorithm tries to generate new positive and negative examples (c.f. Section 4.4). If it is not possible, the refinement is retracted from the hypothesis and the refinement next in order is checked. If the examples were generated, the algorithm proceeds as described earlier. If the algorithm fails to find any suitable refinement, it terminates with a failure.

```
SELECT ?p ?l (COUNT(DISTINCT ?s) AS ?tp)
    (COUNT(DISTINCT ?t) as ?fp)
    (?tp/(?tp+?fp) AS ?precision)
    (?tp/n_pos AS ?recall)
    (2/((1/?precision)+(1/?recall)) AS ?f1)
WHERE {{
       U(?s)
       H(?s)
       ?s ?p ?xl.
       FILTER(isLiteral(?xl))
       VALUES ?s { P }
   } UNION {
       U(?t)
       H(?t)
       ?t ?p ?xl.
       FILTER(isLiteral(?xl))
       VALUES ?t { N }
   } FILTER(?xl <= ?l) {
       SELECT DISTINCT ?p ?l
       WHERE {
           U(?s)
           H(?s)
           ?s ?p ?l.
           FILTER(isLiteral(?l))
           VALUES ?s { P }
       }}}
GROUP BY ?p ?l
HAVING (?recall >= .99)
```

Listing 5. A query used to compute the set of possible refinements of the hypothesis `H(?iri)`, that use a FILTER clause

5 IMPLEMENTATION

To make it easier to reuse the algorithm, we provide a *Python* implementation of the algorithm available at: `https://semantic.cs.put.poznan.pl/ltq/src.tgz`. We also developed an on-line interface to the implementation which we coupled with *Blazegraph 2.1.1*[3] loaded with *DBpedia 2015-04* and made publicly accessible at `https://semantic.cs.put.poznan.pl/ltq/`. In the implementation, we assume a hypothesis to be good enough if the F_1 measure is at least 0.99. Screenshots of the interface are presented in Figures 2 and 3 and the detailed description is presented in [25].

[3] `https://www.blazegraph.com/`

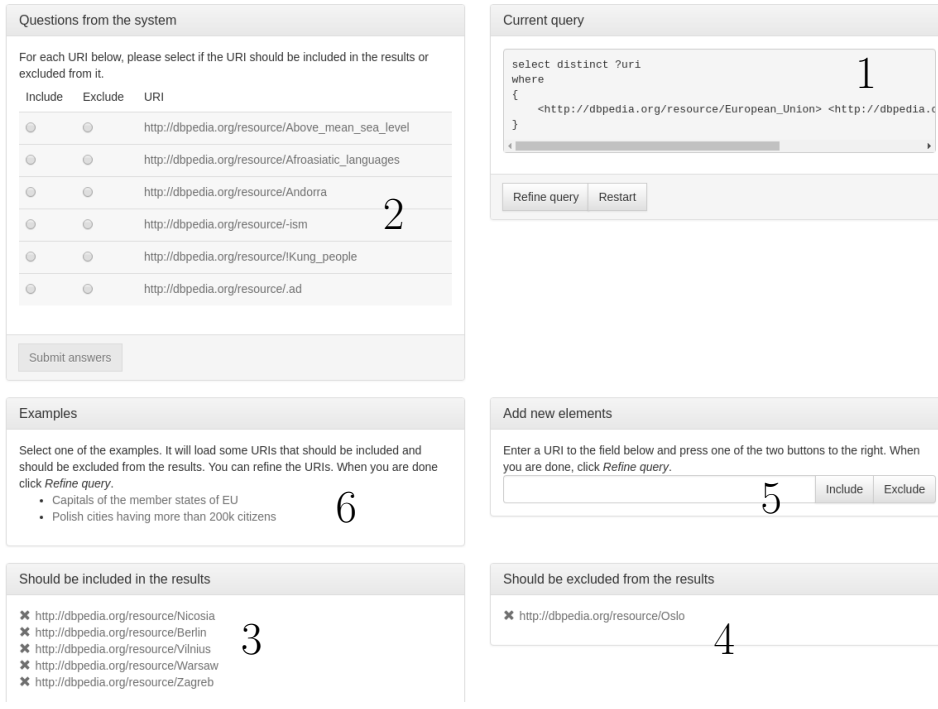

Figure 2. A screenshot of the on-line interface to the implementation of the algorithm. It presents 1) the query corresponding to a current hypothesis, 2) a set of new examples the user is supposed to assign to one of the two sets and already known 3) positive, and 4) negative examples. It is also possible to 5) add a new example by entering its IRI or 6) load one of the demo scenarios.

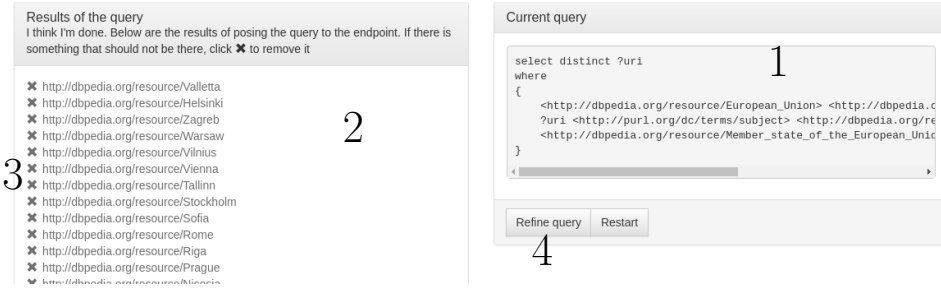

Figure 3. After a good enough hypothesis is reached, the interface displays 1) the query corresponding to the hypothesis and 2) results of posing it to the SPARQL endpoint. The user can then use 3) the `X` buttons to remove some of the unwanted results and further 4) refine the query.

6 EXPERIMENTAL EVALUATION

6.1 Using Query Logs to Simulate the Users

To validate the algorithm, we posed the following research question: is the algorithm able to cover the requirements of the real users of DBpedia? To answer the question, we collected a set of SPARQL queries from the logs of the official DBpedia SPARQL endpoint and used the queries as a gold standard. The queries and the raw results of the experiment are available with the source code. Below, we describe the details of the experiment.

6.1.1 Setup

The *Linked SPARQL Queries* dataset contains queries obtained from query logs of SPARQL endpoints for various popular Web of Data datasets [26]. Among others, it contains queries from the official DBpedia SPARQL endpoint, for the period from 30. 04. 2010 to 20. 07. 2010. From this set, we selected queries that are SELECT queries and contain only a single variable in the head or use star in the head, but contain only a single variable. This way we obtained 415 342 queries (415 145 character-wise distinct queries). We then randomly selected 50 queries, which did not fail when posed to a DBpedia SPARQL endpoint and resulted in at least 20 different IRIs. The endpoint run on *Blazegraph 2.1.1* and contained DBpedia 2015-04. For the selected 50 queries, the smallest number of different IRIs retrieved was 20, and the largest 2 060 507.

We used each of the selected queries to simulate a user. Each query was posed to the SPARQL endpoint and its results (any duplicates removed) were used as a gold standard, i.e. the set of all the IRIs the simulated user is interested in. Out of the gold standard, we randomly selected 5 IRIs, which were given to the algorithm as the initial positive examples. To provide the negative examples we randomly selected the following six IRIs: `dbr:Bydgoszcz`, `dbr:Murmur_%28record_label%29`, `dbr:Julius_Caesar`, `dbo:abstract`, `dbp:after`, `dbo:Agent`. If any of these negative examples was in the gold standard, it was removed from the set of negative examples. We used an empty user-provided BGP.

We then run the algorithm until it converged to a hypothesis that resulted in exactly the same set of IRIs as the gold standard, or for 10 good enough hypotheses generated by the algorithm, whichever came first. If a good enough hypothesis was not perfect, we randomly added up to 5 new positive examples (i.e., new IRIs which were present in the gold standard, but were not present in the results of the hypothesis) and up to 5 new negative examples (i.e., new IRIs that were present in the results of the hypothesis, but were not present in the gold standard). We also counted the number of interactions of the simulated user with the algorithm and measured the wall time.

6.1.2 Results

For 41 of the selected queries the first good enough hypothesis presented to the simulated user was perfect, for 8 the second, and for 1 only the third hypothesis was perfect. The WHERE clause of the gold standard query corresponding to this last case was `?value dbp:subdivisionType dbr:List_of_counties_in_Montana.` The first hypothesis was too broad, consisting of a single triple pattern

```
?iri dbp:areaCode "406"^^xsd:integer.
```

Apparently, this is a correct telephone area code for the state of Montana [31], but so it is for Gümüşhane Province in Turkey [30]. The second try was, on the other hand, too narrow:

```
?iri dbp:subdivisionType
    dbr:Political_divisions_of_the_United_States .
?iri dbp:subdivisionType
    dbr:List_of_counties_in_Montana .
```

It omitted four resources from the gold standard: `dbr:Browning,_Montana`, `dbr:Missoula,_Montana`, `dbr:Great_Falls,_Montana`, `dbr:Helena,_Montana`. After they were added as positive examples, the algorithm converged to the correct hypothesis.

For 38 of the queries, the simulated user was asked only once to label a set of six examples, in case of 4 queries the user was asked twice, for 3 queries thrice, for 2 queries four times, for another 2 queries five times and once six times. The user waited on average for 56 ± 22 seconds (median: 41 seconds) during the whole process for the algorithm to generate new examples or present a good enough hypothesis. In the case of queries that achieved the gold standard with the first good enough hypothesis, the average time was 50 ± 2 seconds. For 82 % of the queries the user obtained a perfect hypothesis in less than a minute and was asked to label at most 12 examples. That means the algorithm is really able to cover requirements of the users, it does not waste their time in waiting and we can answer positively to the research question.

6.2 Using the Algorithm to Solve Classification Problems

To prove that LSQ is not a DBpedia-specific algorithm, we used *SML-Bench*, a benchmarking framework for structured machine learning [29]. SML-Bench provides 9 datasets of varying complexity and size, and integrates a sizable number of learning systems. The authors of SML-Bench performed an extensive evaluation with 8 datasets and 15 configurations of the learning systems. We performed a similar experiment with LSQ and report the details below, comparing to the results reported by the authors of SML-Bench.

6.2.1 Setup

In order to perform the experiment, we had to prepare a wrapper adapting a learning problem to our interactive approach. SML-Bench provides the wrapper with a set of positive and negative learning examples, and an OWL file with the background knowledge. The wrapper then runs an instance of Blazegraph, loads the provided OWL file and initializes LSQ by randomly selecting 10 of the positive and 10 of the negative learning examples and providing them as the initial examples to LSQ. Next, LSQ is executed until it does not converge to a good enough hypothesis or until the prescribed amount of time is exceeded, whichever comes first. Every time LSQ generates a set of examples to be labeled by the user, the current hypothesis is stored and the examples are labeled as follows: if an example is present in the set of positive learning examples, it is labeled as positive, otherwise it is labeled as negative. If the maximal execution time is to be exceeded, all the stored hypotheses are reevaluated on the set of learning examples gathered so far by the algorithm and the one with the highest score is selected as the final result. We assume that the user-provided BGP is empty.

We used the SML-Bench framework to perform the experiment using the same parameters as Westphal et al. [29]: 8 datasets; 5 minutes for a single execution of the algorithm, including loading the data and the final reevaluating of the hypotheses (enforced by the framework); 10-fold cross-validation. To execute the experiment we used a workstation with *Intel Core i7-3770* CPU with 4 cores (8 threads) clocked at 3.40 GHz and equipped with 16 GB of RAM. It must be noted that this is a considerably weaker configuration than the one used by Westphal et al. and thus the limit of 5 minutes was, in fact, more strict.

6.2.2 Results

Following Westphal et al. we report the accuracy and the F_1 score in, respectively, Table 2 and Table 3. The measures were averaged over all the folds of the cross-validation. For comparison, we report the four best systems from the experiment by Westphal et al.: Aleph[4] and DLLearner [18] in three configurations. It must be noted that there was no clear winner in the experiment: different systems were the best on different learning problems.

In one case (the learning problem nctrer/1) LSQ achieved results far better than the remaining systems. In all the folds of the cross-validation LSQ generated exactly the same hypothesis consisting of a single triple pattern:

```
?s <http://dl-learner.org/ont/ActivityOutcome_NCTRER>
    "active"^^xsd:string .
```

Our suspicion is that the true label is encoded in the background knowledge, and thus the obtained result is not credible.

[4] `http://www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph.html`

For the remaining 8 datasets, we used Welch's unequal variances t-test [28] to compare the average accuracy values between the values achieved by LSQ and the highest average accuracy of the remaining systems. The null hypothesis was that both averages are equal. In two cases (pyrimidine/1 and carcinog./1) it was not possible to reject the null hypothesis (p-values, respectively, 0.234 and 0.263), in the five remaining cases the null hypothesis was rejected (p-value below 0.01).

Comparing LSQ with the best algorithm is unfavourable for LSQ, because its hypothesis space is limited to a single SPARQL BGP, and is considerably smaller those of Aleph, DLLearner (CELOE) and DLLearner (OCEL). We thus performed additional analysis comparing LSQ with DLLearner (ELTL). ELTL stands for the *EL Tree Learner* algorithm, that has the target language restricted to OWL EL, similar in its expressivity to the expressivity of SPARQL BGPs.

In the case of four learning problems (carcinog./1, hepatitis/1, lymphogr./1, mutag./42), it was not possible to reject the null hypothesis (p-values, respectively, 0.263, 1.0, 0.071, 0.753). Comparison for the learning problem pyrimidine/1 was not possible due to the missing results for ELTL, and for the learning problems mammogr./1 and prem.leag./1 we rejected the null hypotheses (p-values below 10^{-4}). In both cases we observe that the average accuracy of LSQ is higher than this of ELTL.

From the performed comparison we conclude that the performance of LSQ is at least as good as the performance of ELTL. Moreover, we observe that in some cases LSQ is on a par with the best of the remaining algorithms.

Learning Problem	LSQ	Aleph	DLLearner (CELOE)	DLLearner (OCEL)	DLLearner (ELTL)
carcinog./1	0.53 ± 0.05	0.48 ± 0.10	0.55 ± 0.02	no results	0.55 ± 0.02
hepatitis/1	0.43 ± 0.03	0.67 ± 0.05	0.47 ± 0.05	0.66 ± 0.14	0.41 ± 0.01
lymphogr./1	0.54 ± 0.03	0.83 ± 0.10	0.83 ± 0.11	0.73 ± 0.12	0.54 ± 0.03
mammogr./1	0.52 ± 0.03	0.65 ± 0.04	0.49 ± 0.02	0.82 ± 0.05	0.46 ± 0.01
mutag./42	0.31 ± 0.07	0.72 ± 0.25	0.94 ± 0.13	0.53 ± 0.29	0.30 ± 0.07
nctrer/1	1.00 ± 0.00	0.72 ± 0.14	0.59 ± 0.02	0.81 ± 0.09	0.58 ± 0.02
prem.leag./1	0.89 ± 0.09	0.95 ± 0.09	0.99 ± 0.04	0.85 ± 0.10	0.49 ± 0.02
pyrimidine/1	0.85 ± 0.20	0.95 ± 0.16	0.83 ± 0.17	0.85 ± 0.24	no results

Table 2. The average accuracy and its standard deviation over the 10-folds cross-validation for LSQ, and for the other systems, their values reported from [29]

7 CONCLUSIONS

We presented an algorithm for learning SPARQL queries using examples provided by the user. The algorithm uses active learning to minimize the required number of learning examples. It is also suitable for any RDF graph accessible through a SPARQL endpoint and does not require any preprocessing or initialization phase. By using aggregates, grouping and inline values from SPARQL 1.1, the algorithm is able to

Learning problem	Learning SPARQL Queries	Aleph	DLLearner (CELOE)	DLLearner (OCEL)	DLLearner (ELTL)
carcinog./1	0.69 ± 0.04	0.46 ± 0.12	0.71 ± 0.01	no results	0.71 ± 0.01
hepatitis/1	0.59 ± 0.01	0.38 ± 0.12	0.60 ± 0.02	0.64 ± 0.07	0.58 ± 0.01
lymphogr./1	0.70 ± 0.03	0.84 ± 0.09	0.87 ± 0.07	0.76 ± 0.10	0.70 ± 0.03
mammogr./1	0.64 ± 0.02	0.48 ± 0.08	0.64 ± 0.01	0.78 ± 0.08	0.63 ± 0.00
mutag./42	0.47 ± 0.08	0.43 ± 0.47	0.93 ± 0.14	0.29 ± 0.42	0.46 ± 0.08
nctrer/1	1.00 ± 0.00	0.71 ± 0.18	0.73 ± 0.02	0.85 ± 0.06	0.73 ± 0.02
prem.leag./1	0.86 ± 0.12	0.94 ± 0.11	0.99 ± 0.04	0.97 ± 0.06	0.66 ± 0.02
pyrimidine/1	0.81 ± 0.30	0.90 ± 0.32	0.84 ± 0.15	0.80 ± 0.13	no results

Table 3. The average F_1 score and its standard deviation over the 10-folds cross-validation for LSQ, and the other algorithms, their values reported from [29]

move most of the complex computations to the SPARQL endpoint. Contemporary RDF stores (e.g. *Blazegraph*) are very sophisticated and are able to deal with such queries without any problem. To prove that the algorithm works, we provide an online interface for testing, available at `https://semantic.cs.put.poznan.pl/ltq/`. To prove the usability of the algorithm, we performed two experiments: one using real queries from the *Linked SPARQL Queries* and DBpedia; the other using 8 datasets from the structured machine learning benchmark SML-Bench. In the first case, we showed that the algorithm is able to converge to the gold standard with only a minimal amount of interaction with the user. In the second case, the algorithms performance was similar to those algorithms with similarly restricted hypothesis space.

In the future, we would like to analyze whether using a different measure may provide even faster convergence to a correct hypothesis. We would also like to extend the algorithm with a possibility of obtaining some prior knowledge from the user.

Acknowledgements

Jedrzej Potoniec acknowledges the support of the Polish National Science Center, grant DEC-2013/11/N/ST6/03065 and the support of grant 09/91/DSMK/0659.

REFERENCES

[1] Auer, S.—Bizer, C.—Kobilarov, G.—Lehmann, J.—Cyganiak, R.—Ives, Z. G.: DBpedia: A Nucleus for a Web of Open Data. In: Aberer, K. et al. (Eds.): The Semantic Web (ISWC 2007, ASWC 2007). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4825, 2007, pp. 722–735, doi: 10.1007/978-3-540-76298-0_52.

[2] Baeza-Yates, R.—Ribeiro-Neto, B.: Modern Information Retrieval. 2nd ed. Addison-Wesley Professional, 2011.

[3] Bizer, C.—Lehmann, J.—Kobilarov, G.—Auer, S.—Becker, C.—Cyganiak, R.—Hellmann, S.: DBpedia – A Crystallization Point for the Web of Data. Journal of Web Semantics, Vol. 7, 2009, No. 3, pp. 154–165, doi: 10.1016/j.websem.2009.07.002.

[4] Bottoni, P.—Ceriani, M.: SPARQL Playground: A Block Programming Tool to Experiment with SPARQL. In: Ivanova, V., Lambrix, P., Lohmann, S., Pesquita, C. (Eds.): Proceedings of the International Workshop on Visualizations and User Interfaces for Ontologies and Linked Data (VOILA! 2015). CEUR Workshop Proceedings, Vol. 1456, 2015, pp. 103–108.

[5] Campinas, S.: Live SPARQL Auto-Completion. In: Horridge, M., Rospocher, M., van Ossenbruggen, J. (Eds.): Proceedings of the ISWC 2014 Posters and Demonstrations Track, a Track within the 13th International Semantic Web Conference (ISWC 2014). CEUR Workshop Proceedings, 2014, Vol. 1272, pp. 477–480.

[6] Campinas, S.—Perry, T. E.—Ceccarelli, D.—Delbru, R.—Tummarello, G.: Introducing RDF Graph Summary with Application to Assisted SPARQL Formulation. In: Hameurlain, Q., Min Tjoa, A., Wagner, R. (Eds.): 23rd International Workshop on Database and Expert Systems Applications (DEXA 2012). IEEE Computer Society, 2012, pp. 261–266, doi: 10.1109/dexa.2012.38.

[7] Carothers, G.—Prud'hommeaux, E.: RDF 1.1 Turtle. W3C Recommendation, W3C, February 2014, http://www.w3.org/TR/2014/REC-turtle-20140225/.

[8] Cohn, D.: Active Learning. In: Sammut, C., Webb, G. I. (Eds.): Encyclopedia of Machine Learning. Springer US, Boston, MA, 2010, pp. 10–14, doi: 10.1007/978-0-387-30164-8_6.

[9] Sana e Zainab, S.—Hasnain, A.—Saleem, M.—Mehmood, Q.—Zehra, D.—Decker, S.: FedViz: A Visual Interface for SPARQL Queries Formulation and Execution. In: Ivanova, V., Lambrix, P., Lohmann, S., Pesquita, C. (Eds.): Proceedings of the International Workshop on Visualizations and User Interfaces for Ontologies and Linked Data (VOILA! 2015). CEUR Workshop Proceedings, Vol. 1456, 2015, pp. 49–60.

[10] Fanizzi, N.—d'Amato, C.—Esposito, F.: DL-FOIL Concept Learning in Description Logics. In: Železný, F., Lavrač, N. (Eds.): Inductive Logic Programming (ILP 2008). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 5194, 2008, pp. 107–121, doi: 10.1007/978-3-540-85928-4_12.

[11] Ferré, S.: Sparklis: An Expressive Query Builder for SPARQL Endpoints with Guidance in Natural Language. Semantic Web, Vol. 8, 2017, No. 3, pp. 405–418, doi: 10.3233/sw-150208.

[12] Gennari, J. H.—Musen, M. A.—Fergerson, R. W.—Grosso, W. E.—Crubézy, M.—Eriksson, H.—Noy, N. F.—Tu, S. W.: The Evolution of Protégé: An Environment for Knowledge-Based Systems Development. International Journal of Human-Computer Studies, Vol. 58, 2003, No. 1, pp. 89–123, doi: 10.1016/s1071-5819(02)00127-1.

[13] Gombos, G.—Kiss, A.: SPARQL Query Writing with Recommendations Based on Datasets. In: Yamamoto, S. (Ed.): Human Interface and the Management of Information. Information and Knowledge Design and Evaluation (HIMI 2014). Pro-

ceedings, Part I. Springer, Cham, Lecture Notes in Computer Science, Vol. 8521, 2014, pp. 310–319, doi: 10.1007/978-3-319-07731-4_32.

[14] Harris, S.—Seaborne, A.: SPARQL 1.1 Query Language. W3C Recommendation, W3C, March 2013, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[15] Höffner, K.—Lehmann, J.: Towards Question Answering on Statistical Linked Data. In: Sack, H., Filipowska, A., Lehmann, J., Hellmann, S. (Eds.): Proceedings of the 10th International Conference on Semantic Systems (SEMANTICS 2014). ACM, 2014, pp. 61–64, doi: 10.1145/2660517.2660521.

[16] Höfner, K.—Walter, S.—Marx, E.—Usbeck, R.—Lehmann, J.—Ngonga Ngomo, A.-C.: Survey on Challenges of Question Answering in the Semantic Web. Semantic Web, Vol. 8, 2017, No. 6, pp. 895–920, doi: 10.3233/sw-160247.

[17] Ławrynowicz, A.—Potoniec, J.: Pattern Based Feature Construction in Semantic Data Mining. International Journal on Semantic Web and Information Systems (IJSWIS), Vol. 10, 2014, No. 1, pp. 27–65, doi: 10.4018/ijswis.2014010102.

[18] Lehmann, J.: DL-Learner: Learning Concepts in Description Logics. Journal of Machine Learning Research, Vol. 10, 2009, pp. 2639–2642.

[19] Lehmann, J.—Bühmann, L.: AutoSPARQL: Let Users Query Your Knowledge Base. In: Antoniou, G., Grobelnik, M. et al. (Eds.): The Semantic Web: Research and Applications (ESWC 2011). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 6643, 2011, pp. 63–79, doi: 10.1007/978-3-642-21034-1_5.

[20] Lehmann, J.—Isele, R.—Jakob, M.—Jentzsch, A.—Kontokostas, D.—Mendes, P. N.—Hellmann, S.—Morsey, M.—van Kleef, P.—Auer, S.—Bizer, C.: DBpedia – A Large-Scale, Multilingual Knowledge Base Extracted from Wikipedia. Semantic Web, Vol. 6, 2015, No. 2, pp. 167–195, doi: 10.3233/SW-140134.

[21] de Leon, A.—Wisniewki, F.—Villazón-Terrazas, B.—Corcho, O.: Map4rdf – Faceted Browser for Geospatial Datasets. PMOD Workshop, USING OPEN DATA: Policy Modeling, Citizen Empowerment, Data Journalism, 2012.

[22] Muñoz, S.—Pérez, J.—Gutierrez, C.: Minimal Deductive Systems for RDF. In: Franconi, E., Kifer, M., May, W. (Eds.): The Semantic Web: Research and Applications (ESWC 2007). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4519, 2007, pp. 53–67, doi: 10.1007/978-3-540-72667-8_6.

[23] Oren, E.—Delbru, R.—Decker, S.: Extending Faceted Navigation for RDF Data. In: Cruz, I. F. et al. (Eds.): The Semantic Web – ISWC 2006. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4273, 2006, pp. 559–572, doi: 10.1007/11926078_40.

[24] Paulheim, H.—Fürnkranz, J.: Unsupervised Generation of Data Mining Features from Linked Open Data. In: Burdescu, D. D., Akerkar, R., Badica, C. (Eds.): 2nd International Conference on Web Intelligence, Mining and Semantics (WIMS '12). ACM, 2012, Art. No. 31, doi: 10.1145/2254129.2254168.

[25] Potoniec, J.: An On-Line Learning to Query System. In: Kawamura, T., Paulheim, H. (Eds.): Proceedings of the ISWC 2016 Posters and Demonstrations Track. CEUR Workshop Proceedings, Vol. 1690, 2016, 4 pp.

[26] Saleem, M.—Ali, M. I.—Hogan, A.—Mehmood, Q.—Ngonga Ngomo, A.-C.: LSQ: The Linked SPARQL Queries Dataset. In: Arenas, M. et al. (Eds.): The

Semantic Web – ISWC 2015. Proceedings, Part II. Springer, Cham, Lecture Notes in Computer Science, Vol. 9367, 2015, pp. 261–269, doi: 10.1007/978-3-319-25010-6_15.

[27] Stickler, P.: CBD – Concise Bounded Description. W3C Member Submission, W3C, June 2005, https://www.w3.org/Submission/CBD/.

[28] Welch, B. L.: The Generalization of 'Student's' Problem When Several Different Population Variances Are Involved. Biometrika, Vol. 34, 1947, No. 1-2, pp. 28–35, doi: 10.1093/biomet/34.1-2.28.

[29] Westphal, P.—Bühmann, L.—Bin, S.—Jabeen, H.—Lehmann, J.: SML-Bench – A Benchmarking Framework for Structured Machine Learning. Semantic Web, Vol. 10, 2019, No. 2, pp. 231–245, doi: 10.3233/sw-180308.

[30] Wikipedia. Gümüşhane Province – Wikipedia, the Free Encyclopedia, 2015. [Online, accessed 22-June-2016].

[31] Wikipedia. Area Code 406 – Wikipedia, the Free Encyclopedia, 2016. Online, accessed 24-June-2016.

[32] Williams, G.—Feigenbaum, L.—Clark, K.—Torres, E.: SPARQL 1.1 Protocol. W3C Recommendation, W3C, March 2013, http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/.

[33] Wood, D.—Lanthaler, M.—Cyganiak, R.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, W3C, February 2014, http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[34] Xu, K.—Feng, Y.—Zhao, D.: Xser@QALD-4: Answering Natural Language Questions via Phrasal Semantic Parsing. In: Cappellato, L., Ferro, N., Halvey, M., Kraaij, W. (Eds.): Working Notes for CLEF 2014 Conference. CEUR Workshop Proceedings, Vol. 1180, 2014, pp. 1260–1274.

Jedrzej Potoniec is a postdoctoral researcher at Poznan University of Technology (PUT), Poland. He received his Ph.D. from PUT in 2018 for research on learning ontologies from Linked Data. His research interest concentrates on machine learning with structured data, especially with the Semantic Web data.