Learning to Query: from Concepts in Mind to SPARQL Queries

Jedrzej Potoniec

Faculty of Computing, Poznan University of Technology
ul. Piotrowo 3, 60-965 Poznan, Poland
Jedrzej.Potoniec@cs.put.poznan.pl

Abstract. We present an algorithm learning SPARQL queries reflecting a concept in the mind of a user. The algorithm leverages SPARQL 1.1 to put 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.

1 Introduction

Querying data 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. In this paper, we aim to remedy this issue in context of the data represented as an RDF (Resource Description Framework [24]) graph.

Consider the following use case scenario: a user has a concept in mind, expressed in her own terms. She knows some of the relevant nodes in the graph, she also knows some of the irrelevant nodes. Moreover, she can distinguish a relevant node from an irrelevant one for some price, e.g. by spending her time checking it. She wants to obtain a formal query corresponding to the concept in her mind, to be able to query the graph.

To address this use case, in Section 3 we propose an algorithm for learning a SPARQL [11] query by discovering this concept. 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 it is the best place to perform any optimization. A few years ago it would not work, but recent increase in computational power and development of RDF stores made it feasible.

Thought this work, we use the following prefixes: `dbr:` for `http://dbpedia.org/resource/`, `dbo:` for `http://dbpedia.org/ontology/`, `dbp:` for `http://dbpedia.org/property/`, `dct:` for `http://purl.org/dc/terms/`, `xsd:` for `http://www.w3.org/2001/XMLSchema#`.

2 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 her needs, e.g. see [18]. Also variants tailored to a specific types of data were proposed, e.g. [17] describes an interface for geospatial data. A recent system *Sparklis* [8] combines faceted browsing with natural language generation to enable a non-expert user to query an RDF graph.

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* [25] is a system for answering factual questions and *CubeQA* [12] is designed to answer statistical queries requiring aggregate functions. A very recent survey [13] 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 [26] or by constructing a SPARQL query from building blocks [3].

A system using recommendations still requires from a user knowledge of SPARQL, but it helps to deal with an unknown vocabulary. [10] recommends predicate names based on an inferred type of a variable in a triple pattern. [5] describes a method for recommending query terms based on a graph summary, while [4] 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 [16]. The main difference are the assumptions: [16] performs all the computation on the client-side, obtaining information about resources using an approach similar to Concise Bounded Description [21] and then computing their intersections. Our approach leverages new features of SPARQL 1.1 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. [14] 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 [19] the authors propose a method for unsupervised mining of data mining features, which directly correspond to SPARQL queries. [15] and [7] 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 a *DL Query* tab of *Protégé* [9].

3 Learning SPARQL queries

3.1 Basic concepts

Throughout this section, we use the following example: the user wants to formulate a SPARQL query which allows her to query *DBpedia* [1] for the capitals of states of the European Union. She knows some of them: Warsaw, Berlin, Zagreb, Nicosia and Vilnius. She can also recognize if an arbitrary DBpedia URI refeers to one of the capitals by reading a Wikipedia article corresponding to the URI. Of course, reading consumes her time, so she wants to limit the number of articles read. She also know that Oslo, being a capital of Norway, which is not an European Union member, should not be present in the results.

In a very broad terms, the algorithm works by formulating a hypothesis and verifying it with the user. A *hypothesis* consists of SPARQL triple patterns and filters. Every triple pattern in the hypothesis has a fixed predicate (i.e. the predicate is an URI), the subject and object may be an URI, a literal or a variable. Every filter contains only an expression of form *variable* `>=` *fixed literal* or *variable* `<=` *fixed literal*. An *empty hypothesis* is a hypothesis, which contains no triple patterns or filters. We require that an undirected graph corresponding to a basic graph pattern defined by a hypothesis is a connected graph. A *query corresponding to a hypothesis* is a SELECT SPARQL query with a single variable in the head. We ask only for unique bindings, i.e. the variable is preceded with DISTINCT modifier. The WHERE clause contains only the hypothesis and there are no other clauses in the query. We denote a hypothesis by `H(?uri)`, where `?uri` is the variable in the head of the corresponding query. For example, the algorithm may generate the following hypothesis `H(?uri)`:

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

To formulate a hypothesis, the algorithm uses a set of positive examples `P` and a set of negative examples `N`. Both of these sets contain URIs from the RDF graph. `P` is a subset of URIs expected in the results of the final hypothesis. `N` is a set of URIs 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 URIs in both sets, and then they are extended during the execution of the algorithm. `n_pos` stands for the number of URIs 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 a number of URIs from the set `P` which were retrieved by a hypothesis, and by FP a number of URIs from the set `N` which were retrieved by the hypothesis. Following [2], we define three measures. *Precision* is a fraction of positive URIs retrieved by the hypothesis over both positive and negative URIs retrieved by the hypothesis, i.e. $p = \frac{TP}{TP+FP}$. *Recall* is a fraction of positive URIs retrieved by

the hypothesis over the number of known positive URIs `n_pos`, i.e. $r = \frac{TP}{\texttt{n\_pos}}$. Finally, F_1 *measure* is a harmonic mean of precision and recall, i.e.

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

3.2 Overview of the algorithm

In the beginning of the algorithm, 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 the Section 3.5. 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 3.3), it is presented to the user along with the full result of posing it 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 a URI from the result which should not be there, or by adding a URI 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.

3.3 Measuring quality of a hypothesis

To compute with a SPARQL endpoint how good a hypothesis is, the following query is used:

```
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 {
    {
        H(?s)
        VALUES ?s { P }
    } UNION {
        H(?t)
        VALUES ?t { T }
    }}
```

Such a query computes the measures described above. The variable `?s` (resp. `?t`) is bound to all URIs 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 a final answer.

3.4 Generating new examples and querying the user

If the hypothesis is not good enough, the algorithm should gather more evidence, to either confirm or 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 following query is used:

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

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

Generating negative examples is more difficult. Let `Hp(?uri)` be a hypothesis the hypothesis `H(?uri)` directly originated from, i.e. `Hp(?uri)` is the hypothesis `H(?uri)` without the last refinement. Consider the following SPARQL query:

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

The only common variable between `Hp(?uri)` and `H(?uri)` is `?uri`, the rest is uniquely renamed. 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 maximize benefit from getting correct labels for them, e.g. to remove as much uncertainty as possible [6].

After the examples are generated, they are presented to the user. 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 3.2.

3.5 Hypothesis refinement

If the hypothesis is not good enough, the algorithm must refine it. First, the algorithm checks if it is possible to generate new examples using the current

hypothesis. If 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 `?uri dct:subject dbr:Category:Capitals_in_Europe`. To generate a set of such refinements the following query is used.

```
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 {{
        H(?s)
        ?s ?p ?o .
        VALUES ?s { P }
    } UNION {
        H(?t)
        ?t ?p ?o .
        VALUES ?t { N }
    }}
GROUP BY ?p ?o
HAVING (?recall>=.99)
```

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 a subject being a variable already present in the hypothesis (`?s` and `?t`) and fixed values for a predicate and an 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(?uri)` be `dbr:European_Union dbo:wikiPageWikiLink ?uri.` A refinement `?uri dct:subject dbr:Category:Capitals_in_Europe .` has recall $r = 1$, as it returns all five elements of the set `P`, and it also has precision $p = 1$ as it does not return the negative example, i.e. Oslo. On the other hand, a refinement `?uri 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 an object (resp. subject) and an existing variable as a subject (resp. object), the algorithm replaces `?o` with a blank node `[]` in the triple patterns and removes `?o` from the head and from the GROUP BY clause.

It requires a more elaborate query to provide refinements with FILTER. Consider the following query:

```
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 {{
        H(?s)
        ?s ?p ?xl. FILTER(isLiteral(?xl))
        VALUES ?s { P }
    } UNION {
        H(?t)
        ?t ?p ?xl. FILTER(isLiteral(?xl))
        VALUES ?t { N }
    } FILTER(?xl <= ?l) {
        SELECT DISTINCT ?p ?l
        WHERE {
            H(?s)
            ?s ?p ?l. FILTER(isLiteral(?l))
            VALUES ?s { P }
        }}}
GROUP BY ?p ?l
HAVING (?recall >= .99)
```

The query operates in two steps. First, the subquery extracts all 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. An example of a refinement obtained this way is `?uri dbo:populationTotal ?var1337. filter(?var1337 >= "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 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 3.2. Otherwise, the algorithm tries to generate new positive and negative examples (c.f. Section 3.4). If it is not possible, the refinement is retracted from the hypothesis and the next refinement 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.

4 Implementation

To make it easier to reuse the algorithm, we provide a *Python* implementation of the algorithm available in a *Git* repository: `https://bitbucket.org/`

`jpotoniec/kretr/`. We also developed an on-line interface to the implementation which we coupled with *Blazegraph 2.1.1*[1] 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 1 and 2.

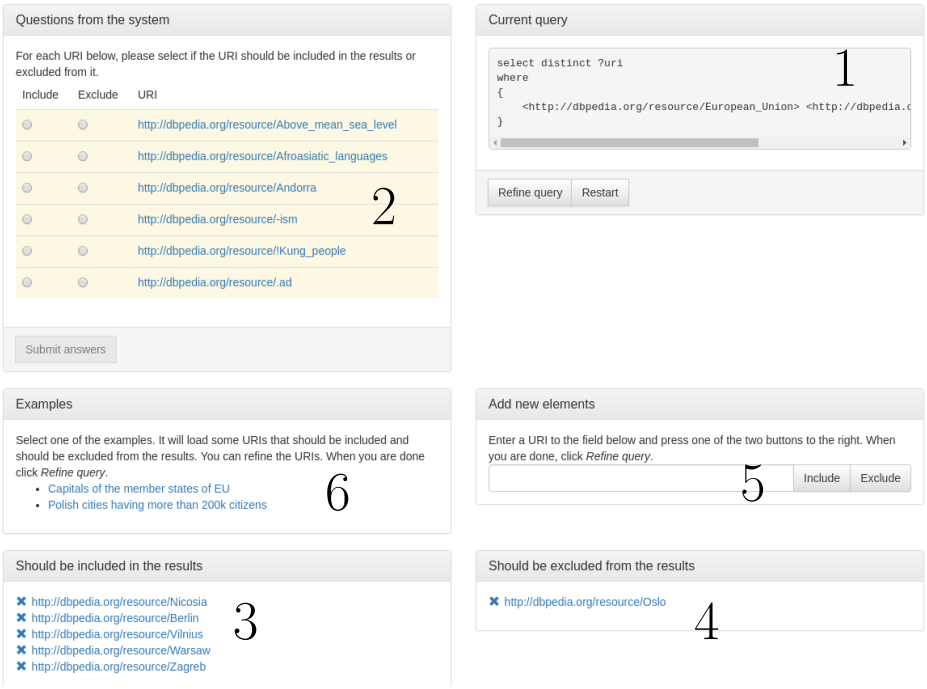

Fig. 1. A screenshot of the on-line interface to the implementation of the algorithm. It presents (1) the query corresponding to a current hypothesis, (2) a set of new examples the user is supposed to assign to one of the two sets and already known (3) positive, and (4) negative examples. It is also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

5 Experimental evaluation

To validate the algorithm, we posed the following research question: is the algorithm able to cover requirements of 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

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

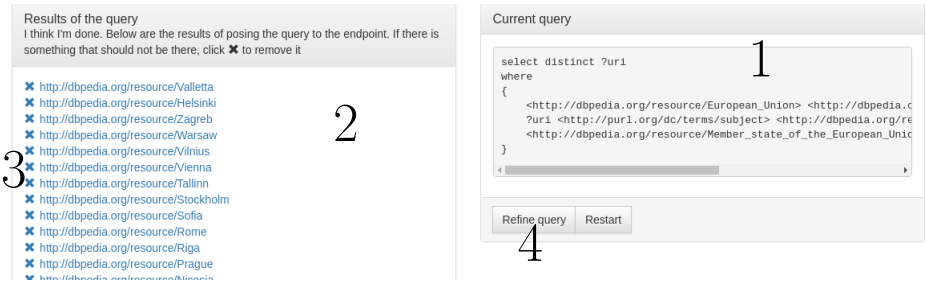

Fig. 2. After a good enough hypothesis is reached, the interface displays (1) the query corresponding to the hypothesis and (2) results of posing it to the SPARQL endpoint. The user can then use (3) the `X` buttons to remove some of the unwanted results and further (4) refine the query.

and the raw results of the experiment are available in the repository with the source code (see Section 4). Below, we describe the details of the experiment.

5.1 Setup

The *Linked SPARQL Queries* dataset contains queries obtained from query logs of SPARQL endpoints for various popular Open Linked Data datasets [20]. 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 415342 queries (415145 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 URIs. The endpoint run on *Blazegraph 2.1.1* and contained *DBpedia* 2015-04. For the selected 50 queries, the smallest number of different URIs was 20, and the largest 2060507.

For each of the selected queries, we used it to simulate a user. The query was posed to the SPARQL endpoint and its results (any duplicates removed) were used as a gold standard, i.e. a set of all URIs the simulated user is interested in. Out of the gold standard, we randomly selected 5 URIs, which were given to the algorithm as initial positive examples. To provide negative examples we randomly selected the following six URIs: `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 then run the algorithm until it converged to a hypothesis that resulted in exactly the same set of URIs as the gold standard, or for 10 good enough hypotheses generated by the algorithm, whatever came first. If a good enough hypothesis was not perfect, we randomly added up to 5 new positive examples (i.e. new URIs 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 URIs that were present in the results of the hypothesis, but were not present in the gold standard). We also counted number of interactions of the simulated user with the algorithm and measured wall time.

5.2 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 only `?uri dbp:areaCode "406"^^xsd:integer.` Apparently, this is a correct telephone area code for the state of Montana [23], but so it is for Gümüşhane Province in Turkey [22]. The second try was, on the other hand, too narrow:

```
?uri dbp:subdivisionType dbr:Political_divisions_of_the_United_States .
?uri 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 a 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 case of queries which reached 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 Conclusions

We presented an algorithm for learning SPARQL queries reflecting a concept in mind of a user using examples provided by the user. The algorithm uses active learning to minimize required number of examples. It is also suitable for any RDF graph accessible through a SPARQL endpoint and does not require any preprocessing or initialization phase. SPARQL 1.1 gave us a set of powerful tools, which enabled us to move most of the complex computations in the algorithm 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 on-line interface for testing,

available at `https://semantic.cs.put.poznan.pl/ltq/`. We also performed an experiment using real queries obtained from the *Linked SPARQL Queries*. We showed that the algorithm is able to converge to a gold standard with only a minimal amount of interaction with the user.

In the future, we would like to analyze if using different measures may provide even faster convergence to a correct hypothesis. We would also like to extend the algorithm with possibility of obtaining some prior knowledge from the user. Finally, this algorithm is targeted for an end-user, who is not necessarily a Semantic Web expert. Coupling the algorithm with a tool for visual presentation of SPARQL queries and displaying *Wikipedia* links instead of *DBpedia* URIs would enable us to perform a crowdsourcing experiment to verify if the algorithm meets the requirements of non-expert users.

Acknowledgement. Jedrzej Potoniec acknowledges support of Polish National Science Center, grant DEC-2013/11/N/ST6/03065.

References

1. Auer, S., Bizer, C., et al.: Dbpedia: A nucleus for a web of open data. In: Aberer, K., Choi, K., et al. (eds.) The Semantic Web. LNCS, vol. 4825, pp. 722–735. Springer (2007)
2. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Pearson Education Limited (2011)
3. Bottoni, P., Ceriani, M.: SPARQL playground: A block programming tool to experiment with SPARQL. In: Ivanova, V., Lambrix, P., et al. (eds.) Proc. of the International Workshop on Visualizations and User Interfaces for Ontologies and Linked Data. CEUR Workshop Proc., vol. 1456, p. 103
4. Campinas, S.: Live SPARQL auto-completion. In: Horridge, M., Rospocher, M., van Ossenbruggen, J. (eds.) Proc. of the ISWC 2014 Posters & Demonstrations Track. CEUR Workshop Proceedings, vol. 1272, pp. 477–480 (2014)
5. Campinas, S., Perry, T., et al.: Introducing RDF graph summary with application to assisted SPARQL formulation. In: Hameurlain, A., Tjoa, A.M., Wagner, R. (eds.) 23rd Int. Workshop on Database and Expert Systems Applications, DEXA 2012. pp. 261–266. IEEE Computer Society (2012)
6. Cohn, D.: Active learning. In: Sammut, C., Webb, G.I. (eds.) Encyclopedia of Machine Learning. pp. 10–14. Springer US, Boston, MA (2010)
7. Fanizzi, N., d'Amato, C., Esposito, F.: DL-FOIL concept learning in description logics. In: Železný, F., Lavrac, N. (eds.) Inductive Logic Programming. LNCS, vol. 5194, pp. 107–121. Springer (2008)
8. Ferré, S.: SPARKLIS: a SPARQL endpoint explorer for expressive question answering. In: Horridge, M., Rospocher, M., van Ossenbruggen, J. (eds.) Proc. of the ISWC 2014 Posters & Demonstrations Track. CEUR Workshop Proc., vol. 1272, pp. 45–48 (2014)
9. Gennari, J.H., Musen, M.A., et al.: The evolution of protégé: an environment for knowledge-based systems development. Int. J. Hum.-Comput. Stud. 58(1), 89–123 (2003)

10. Gombos, G., Kiss, A.: SPARQL query writing with recommendations based on datasets. In: Yamamoto, S. (ed.) Human Interface and the Management of Information. LNCS, vol. 8521, pp. 310–319. Springer (2014)
11. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C recommendation, W3C (Mar 2013), http://www.w3.org/TR/2013/REC-sparql11-query-20130321/
12. Höffner, K., Lehmann, J.: Towards question answering on statistical linked data. In: Sack, H., Filipowska, A., et al. (eds.) Proc. of the 10th Int. Conf. on Semantic Systems, SEMANTICS 2014. pp. 61–64. ACM (2014)
13. Höffner, K., Walter, S., Marx, E., Usbeck, R., Lehmann, J., Ngomo, A.C.N.: Survey on challenges of question answering in the semantic web. Semantic Web Journal (accepted for publication) (2016)
14. Lawrynowicz, A., Potoniec, J.: Pattern based feature construction in semantic data mining. Int. J. Semantic Web Inf. Syst. 10(1), 27–65 (2014), http://dx.doi.org/10.4018/ijswis.2014010102
15. Lehmann, J.: Dl-learner: Learning concepts in description logics. Journal of Machine Learning Research 10, 2639–2642 (2009)
16. 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. LNCS, vol. 6643, pp. 63–79. Springer (2011)
17. Leon, A.d., Wisniewki, F., Villazón-Terrazas, B., Corcho, O.: Map4rdf-faceted browser for geospatial datasets. In: USING OPEN DATA: policy modeling, citizen empowerment, data journalism (2012)
18. Oren, E., Delbru, R., Decker, S.: Extending faceted navigation for RDF data. In: Cruz, I.F., Decker, S., et al. (eds.) The Semantic Web - ISWC 2006. LNCS, vol. 4273, pp. 559–572. Springer (2006)
19. Paulheim, H., Fürnkranz, J.: Unsupervised generation of data mining features from linked open data. In: Burdescu, D.D., Akerkar, R., Badica, C. (eds.) 2nd Int. Conf. on Web Intelligence, Mining and Semantics. pp. 31:1–31:12. ACM (2012)
20. Saleem, M., Ali, M.I., other: LSQ: the linked SPARQL queries dataset. In: Arenas, M., Corcho, Ó., et al. (eds.) The Semantic Web - ISWC 2015. LNCS, vol. 9367, pp. 261–269. Springer (2015)
21. Stickler, P.: CBD – concise bounded description. W3C member submission, W3C (Jun 2005), https://www.w3.org/Submission/CBD/
22. Wikipedia: Gümüşhane province — wikipedia, the free encyclopedia (2015), https://en.wikipedia.org/w/index.php?title=G%C3%BCm%C3%BC%C5%9Fhane_Province&oldid=672439114, [Online; accessed 22-June-2016]
23. Wikipedia: Area code 406 — wikipedia, the free encyclopedia (2016), https://en.wikipedia.org/w/index.php?title=Area_code_406&oldid=714668171, [Online; accessed 24-June-2016]
24. Wood, D., Lanthaler, M., Cyganiak, R.: RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C (Feb 2014), http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/
25. Xu, K., Feng, Y., Zhao, D.: Answering natural language questions via phrasal semantic parsing. In: Cappellato, L., Ferro, N., et al. (eds.) Working Notes for CLEF 2014 Conference. CEUR Workshop Proc., vol. 1180, pp. 1260–1274 (2014)
26. e Zainab, S.S., Saleem, M., Mehmood, Q., et al.: Fedviz: A visual interface for SPARQL queries formulation and execution. In: Ivanova, V., Lambrix, P., et al. (eds.) Proc. of the International Workshop on Visualizations and User Interfaces for Ontologies and Linked Data. CEUR Workshop Proc., vol. 1456, p. 49 (2015)