Improving the Performance of the DL-Learner SPARQL Component for Semantic Web Applications

Didier Cherix, Sebastian Hellmann Jens Lehmann

Universität Leipzig, IFI/BIS/AKSW, D-04109 Leipzig, Germany
{lastname}@informatik.uni-leipzig.de, http://aksw.org

Abstract. The vision of the Semantic Web is to make use of semantic representations on the largest possible scale - the Web. Large knowledge bases such as DBpedia, OpenCyc, GovTrack are emerging and freely available as Linked Data and SPARQL endpoints. Exploring and analysing such knowledge bases is a significant hurdle for Semantic Web research and practice. As one possible direction for tackling this problem, we present an approach for obtaining complex class expressions from objects in knowledge bases by using Machine Learning techniques. We describe in detail how they leverage existing techniques to achieve scalability on large knowledge bases available as SPARQL endpoints or Linked Data. The algorithms are made available in the open source DL-Learner project and we present several real-life scenarios in which they can be used by Semantic Web applications. Because of the wide usage of the method in several well-known tools, we optimized and benchmarked the existing algorithms and show that we achieve an approximately 3-fold increase in speed, in addition to a more robust implementation.

1 Introduction

The vision of the Semantic Web aims to make use of semantic representations on the largest possible scale - the Web. Large knowledge bases such as DBpedia, OpenCyc, GovTrack are emerging and freely available as Linked Data and SPARQL endpoints. Due to their sheer size, however, users of large Semantic Web knowledge bases are often facing the problem, that they can hardly know which identifiers are used and are available for the construction of queries. Furthermore, domain experts might not be able to express their queries in a structured form at all, but they often have a very precise imagination what kind of results they would like to retrieve. A historian, for example, searching in DBpedia for ancient Greek law philosophers influenced by Plato can easily name some examples and if presented a selection of prospective results he will be able to quickly identify false results.

However, he might not be able to efficiently construct a formal query adhering to the large DBpedia knowledge base a priori. The construction of queries asking for objects of a certain kind contained in an ontology, such as in the

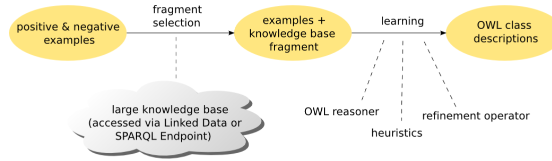

Fig. 1: Process illustration: In a first step, a fragment is selected based on instances from a knowledge source and in a second step the learning process is started on this fragment and the given examples. [1]

previous example, can be understood as a class construction problem: We are searching for a class expression which subsumes exactly those objects adhering to our informal query (e.g. ancient Greek law philosophers influenced by Plato). Recently, several methods have been proposed for constructing ontology classes by means of Machine Learning techniques from positive and negative examples [1]. These techniques are tailored for small and medium size knowledge bases, while they cannot be directly applied to large knowledge bases (such as the initially mentioned ones) due to their dependency on reasoning methods. The scalability of the algorithms is ensured by reasoning only over ”interesting parts” of a knowledge base for a given task. As a result users of large knowledge bases are empowered to construct queries by iteratively providing positive and negative examples to be contained in the prospective result set.

In this paper, we will motivate the usefulness of the method first presented in [1] by describing the algorithm and how it is employed in several well-known applications(HANNE, ORE, DL-Learner, Tiger Corpus Navigator, etc.). Then we will introduce the changes we have made to improve performance of the data acquisition method. Although the basic problem was introduced several years ago[2], we will discuss lessons learned and actual problems in the last section.

2 Summary of the Existing SPARQL Component

In this paper, we focus on the knowledge source component, which is responsible for downloading a “relevant” fragment of the available knowledge base via SPARQL. At the beginning, the component receives a list of positive and negative examples needed for supervised machine learning. As the separation into positive and negative examples is irrelevant for data acquisition, a list of all (seed) examples is created by merging. We also assume that the given examples must be instances of an OWL class. Based on a given recursion depth parameter, the old component retrieved all information of the seed examples naively, by traversing the graph with SPARQL as shown in Figure 2 and without making any distinction between A- and T-Box (i.e. treating OWL axioms as triples) . Reasoning on the resulting RDF fragment is sound, because of the monotonicity of DL, but naturally incomplete, since we restricted the available knowledge. Nevertheless, Hellmann et. al. [1] have shown that results are comparable to learning with the complete data. Actually, better results can be achieved in the same time and

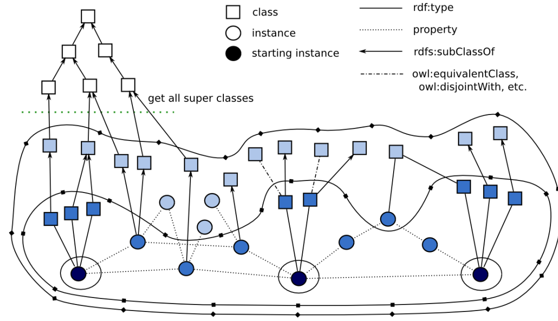

Fig. 2: Extraction with three starting example instances. The circles represent different recursion depths. The circles around the starting instances signify recursion depth 0. The larger inner circle represents the fragment with recursion depth 1 and the largest outer circle with recursion depth 2. [1]

with less memory, as unnecessary data is not loaded into the reasoner. Next, we briefly introduce the applications that use a SPARQL component.

HANNE (**H**olistic **A**pplication for **N**avigational K**N**owledge **E**ngineering) [3] enables users and domain experts to navigate through knowledge bases by selecting examples. From these examples, formal OWL class expressions are learned on the fly by the approach presented in this article. When saved by users, these class expressions form an expressive OWL ontology, which can be exploited in numerous ways: as navigation suggestions for users, as a hierarchy for browsing, and as input for a team of ontology editors

The Tiger Corpus Navigator was the predecessor of HANNE and can profit from the improving [4].

Another interesting use case for the proposed fragment selection approach is the debugging and maintenance of large scale ontologies. For this reason, the fragmentation approach has been integrated in the **ORE (ontology repair and enrichment) tool**. ORE [5] uses the approach to scale to larger knowledge bases such as DBpedia and OpenCyc. ORE can detect inconsistencies and unsatisfiable classes within large knowledge bases by continuously loading fragments of increasing size. Furthermore, it can also learn new definitions of classes as described previously.

AutoSPARQL [6] is a question answering support system, which uses fragment extraction for an active learning algorithm. The algorithm allows to refine results obtained from a question answering system.

DL-Learner is a framework to learn concepts in description logics [2,7], the source code for the component is available through this project.

3 The new version of the SPARQL Component

The three tasks of the SPARQL component that are illustrated in Figure 1 are now realized in four separate steps. 1.) The T-Box of the ontology is loaded

and indexed (T-Box index). 2.) All outgoing properties of seed examples and related objects and literals are retrieved. 3.) All asserted classes are retrieved via SPARQL. 4.) The T-Box index is used to infer the class hierarchy.

Step 1: Indexing the T-Box. The previous component traversed the T-Box syntactically based on the retrieved triples during fragment extraction. Our improved method first retrieves all T-Box axioms from the SPARQL endpoint and can, optionally, also directly load an ontology file if available, which we did in out experiments. The ontology is loaded into a reasoner and queried once to materialize the transitive closure of the subclass hierarchy. This subclass hierarchy is then stored in an index, which allows retrieval of superclasses in constant time O(1).

Step 2: A-Box queries. Given a certain recursion depth, the algorithm now traverses the A-Box part of the RDF graph based on the SPARQL template given in Listing 1.1. EX1,..,EXn are the seed examples. After each recursion step, EX1,...,EXn are replaced by the new objects (?o) which have not yet been queried. The SPARQL 1.1 "IN" feature is used.[1] As a last optional step, manual filtering is used to remove data irrelevant for a learning processs.

Listing 1.1: SPARQL query for the A-Box

```
1 CONSTRUCT {?s ?p ?o} { ?s ?p ?o }
2 #list of all seed examples
3 FILTER (?s IN (<EX1>, <EX2>,...,<EXn>))
4 FILTER (!(?p=rdf:type))
5 #other filters, here excluding one specific property and all properties
      of the foaf namespace
6 FILTER ( ?p!=<http://dbpedia.org/property/wikiPageUsesTemplate>  $$ !
      regex(str(?p), '^http://xmlns.com/foaf/0.1/') && ...)
```

Step 3: Typing retrieved instances. Based on the A-Box data retrieved in Step 2, we are able to query all types of the found instances (again using the SPARQL "IN" construct) and include them in the fragment as well. To be able to correctly type all resources, we implemented the rules introduced by Bechhofer and Volz [8]. The SPARQL query is given here:

Listing 1.2: SPARQL query for the T-Box

```
1 CONSTRUCT { ?ex a ?class . } {?ex a ?class . }
2 #list of all objects
3 FILTER (?s IN (<EX1>, <EX2>,...,<EXn>))
4 #other filters, here excluding all classes of the yago namespace
5 FILTER ( !regex(str(?class),  '^http://dbpedia.org/class/yago/')) . }
```

Step 4: T-Box Index. To complete the fragment, we iterate over all retrieved and included classes and query the T-Box index for all subclassof axioms and include them in the fragment.

4 Perfomance Evaluation

The new component is compared to the old one regarding two aspects: execution time and correctness. For this, we designed an experiment using DBpedia

[1] http://www.w3.org/TR/sparql11-query/#func-in

Iteration	new component Min	Total	Avg	old component Min	Total	Avg
1	205	1,642,678	6545	3359	5,999,248	23901
2	136	744,953	4756	2295	3,707,947	19337
3	132	984,861	3924	1703	2,956,095	16817
	158	1,124,165	5,075	2452	4,221,097	20019

Table 1: Performance evaluation showing runtimes in ms.(rounded)

and DL-Learner. Specifically, we used DL-Learner to generate definitions for the DBpedia classes. The experiment is a learning problem with positive and negative examples. Figure 1 explains how DL-Learner is working. We created one learning task for each class in the DBpedia ontology[2]. Starting with the DBpedia T-Box description, we extracted all its classes. For each of class, we retrieved all instances until a limit of 10000 is reached. Then, we randomly choose 30 instances and use those as positive examples. For the negatives examples we use one sister class. A sister class is a class that has the same superclass as the one we are processing, but naturally not the class itself. For each sister class we randomly picked 30 examples from its instances. In the case where the class to learn or its sister have less than 30 instances, all instances are used as positive and negative examples, respectively. As endpoint we use `http://live.dbpedia.org/sparql`. The recursion depth for each learning problem (a DBpedia class is one learning problem) is set to 1. We excluded the following properties from the fragment: `dbo:wikiPageUsesTemplate`, `dbo:wikiPageExternalLink`, `dbo:wordnet_type`, `<http://www.w3.org/2002/07/owl#sameAs>`
and classes having those prefixes:
`http://dbpedia.org/class/yago/` or `http://dbpedia.org/resource/Category:`.

The experiments are repeated three times to reduce the effect of the network and caching from SPARQL-endpoint. To measure the needed time we use the JAMon framework[3]. We measure the execution time of the SPARQL-component for each class to learn. After each experiment (when all classes are learned), we compute the average time for this process. Results are shown in Table 1.

The experiments ran on a 64 bit 2.26 GHZ dual core processor and 8GB RAM. For the old component we use the cache database. The cache has be cleared between each class, because if a class is learned after his superclass to ensure a fair experiment.

5 Discussion and Conclusion

Table 1 shows that the new implementation improves performance significantly. The old component has a maximum query time of 171245 ms, the new one of 68800 ms. As explained above, a cache is not implemented (and possible not

[2] `http://downloads.dbpedia.org/3.6/dbpedia_3.6.owl`
[3] `http://jamonapi.sourceforge.net/`

necessary) in the new component. Still, the runtime improves by factor 4 on average.

In the distribution of the F-measure, we see that the new component has a little bit better results as the old one. One possible explanation for this is that the queries of the new component can be executed faster and, thus, the endpoint returns more results, which improves the approximations in the machine learning processes. Another explanation and probably the bigger effect is that DL-Learner with the new component returns an answer for much more classes compared to the previous component. With the new, DL-Learner found definitions for 248 classes, whereas it did only for 158 previously.

Reasons that for DL-Learner not returning definitions are Java exceptions, which are usually due to errors in the retrieved data. The new component fired 61 exceptions until the experiment. But only three of those caused an interruption of the algorithm and cause the algorithm to return no answer. All of those are `com.hp.hpl.jena.shared.BadUriException`s resulting from malformed output from the SPARQL endpoint. The others exceptions are all parse exceptions due to errors in the XML output of the endpoint. Those exceptions cause a loss of some retrieved triples when they occur, but the algorithm can still recover and terminate. The behavior is more stable than and able to recover from more types of errors than the previous component.

Overall, we presented an improved fragment extraction component for machine learning over large SPARQL endpoints, which is more efficient and more stable than previously published methods.

References

1. Hellmann, S., Lehmann, J., Auer, S.: Learning of OWL class descriptions on very large knowledge bases. IJSWIS **5**(2) (2009) 25–48
2. Lehmann, J.: DL-Learner: learning concepts in description logics. The Journal of Machine Learning Research **10** (2009) 2639–2642
3. Hellmann, S., Unbehauen, J., Lehmann, J.: Hanne - a holistic application for navigational knowledge engineering. In: Posters and Demos of ISWC. (2010)
4. Hellmann, S., Unbehauen, J., Chiarcos, C., Ngonga Ngomo, A.C.: The tiger corpus navigator. In: Proceedings of the Ninth International Workshop on Treebanks and Linguistic Theories (TLT9). NEALT Proceeding Series (2010)
5. Lehmann, J., Bühmann, L.: Ore - a tool for repairing and enriching knowledge bases. In: ISWC, Springer (2010)
6. Lehmann, J., Bühmann, L.: Autosparql: Let users query your knowledge base. In: ESWC, Springer (2011)
7. Lehmann, J., Hitzler, P.: Concept learning in description logics using refinement operators. Machine Learning journal **78**(1-2) (2010) 203–250
8. Bechhofer, S., Volz, R.: Patching syntax in OWL ontologies. In: ISWC. Volume 3298 of LNCS., Springer (2004) 668–682