Learning of OWL Class Descriptions on Very Large Knowledge Bases

Sebastian Hellmann and Jens Lehmann and Sören Auer
Department of Computer Science, Universität Leipzig, Germany

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, and others are emerging and are 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 descriptions from objects in knowledge bases by using Machine Learning techniques. We describe in detail how we leverage existing techniques to achieve scalability on large knowledge bases available as SPARQL endpoints or Linked Data. Our 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.

Introduction

The vision of the Semantic Web is to make use of semantic representations on the largest possible scale - the Web. We currently experience that Semantic Web technologies are gaining momentum and large knowledge bases such as DBpedia (Auer m. fl., 2007), OpenCyc (Lenat, 1995), GovTrack (Tauberer, 2008) and others are freely available. These knowledge bases are based on semantic knowledge representation standards like RDF and OWL. They contain hundred thousands of properties as well as classes and an even larger number of facts and relationships. These knowledge bases and many more (ESWWiki, 2008) are available as Linked Data (Berners-Lee, 2006; Bizer, Cyganiak, & Heath, 2007) or SPARQL endpoints (Clark, Feigenbaum, & Torres, 2008).

Due to their sheer size, users of these knowledge bases, however, are 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 previous example, can be understood as a class construction problem: We are searching for a class description 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 (Lehmann & Hitzler, 2007a, 2007b). 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. In this paper, we present an approach for leveraging Machine Learning algorithms for learning of ontology class descriptions in large knowledge bases, in particular those available as SPARQL(Clark m. fl., 2008) endpoints or Linked Data. 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.

Overall, we make the following contributions:

- development of a flexible method for extracting relevant parts of very large and possibly interlinked knowledge bases for a given learning task,
- thorough implementation, integration, and evaluation of these methods in the DL-Learner framework (Lehmann, 2007)
- presentation of several application scenarios and examples employing some of the largest knowledge bases available on the Web.

In this article, we will first cover preliminaries, namely a quick introduction in Description Logics, OWL, and the learning problem we consider. We then briefly describe the underlying learning algorithm and present in detail how it can be applied on very large knowledge sources. We describe and evaluate our approach in several application scenarios in the following sections and, finally, conclude with some related work and an outlook on future work.

OWL, Description Logics

Description logics are a family of knowledge representation (KR) formalisms. They emerged from earlier KR formalisms like semantic networks and frames. Their origin lies in the work of Brachmann on structured inheritance networks (Brachman, 1978). Since then, description logics have

enjoyed increasing popularity. They can essentially be understood as fragments of first-order predicate logic. They have less expressive power, but usually decidable inference problems and a user-friendly variable free syntax.

Description logics represent knowledge in terms of *objects*, *concepts*, and *roles*. Concepts formally describe notions in an application domain, e.g. we could define the concept of being a father as "a man having a child" ($\texttt{Father} \equiv \texttt{Man} \sqcap \exists \texttt{hasChild}.\top$ in DL notation). Objects are members of concepts in the application domain and roles are binary relations between objects. Objects correspond to constants, concepts to unary predicates, and roles to binary predicates in first-order logic.

In description logic systems information is stored in a *knowledge base*. It is divided in two parts: *TBox* and *ABox*. The ABox contains *assertions* about objects. It relates objects to concepts and roles. The TBox describes the *terminology* by relating concepts and roles. (For some expressive description logics this clear separation does not exist.)

As mentioned before, DLs are a family of KR formalisms. We will introduce the $\mathcal{ALC}$ description logic as a prototypical example. It should be noted that $\mathcal{ALC}$ is a proper fragment of OWL (Horrocks, Patel-Schneider, & Harmelen, 2003) and is considered to be a prototypical description logic for research investigations.

$\mathcal{ALC}$ stands for *attributive language with complement*. It allows to construct complex concepts from simpler ones using various language constructs. The next definition shows how such concepts can be built.

Definition 1 (syntax of $\mathcal{ALC}$ concepts) Let N_R be a set of *role names* and N_C be a set of concept names ($N_R \cap N_C = \emptyset$). The elements of N_C are also called *atomic concepts*. The set of $\mathcal{ALC}$ concepts is inductively defined as follows:

1. Each atomic concept is a concept.

2. If C and D are $\mathcal{ALC}$ concepts and $r \in N_R$ a role, then the following are also $\mathcal{ALC}$ concepts:

- $\top$ (top), $\bot$ (bottom)
- $C \sqcup D$ (disjunction), $C \sqcap D$ (conjunction), $\neg C$ (negation)
- $\forall r.C$ (value/universal restriction), $\exists r.C$ (existential restriction)

The semantics of $\mathcal{ALC}$ concepts is defined by means of interpretations. See the following definition and Table 1 listing all $\mathcal{ALC}$ concept constructors. The corresponding OWL terminology is also listed (according to (Bechhofer m. fl., 2004))

Definition 2 (interpretation) An *interpretation* $\mathcal{I}$ consists of a non-empty *interpretation domain* $\Delta^{\mathcal{I}}$ and an *interpretation function* $\cdot^{\mathcal{I}}$, which assigns to each $A \in N_C$ a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to each $r \in N_R$ a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

In the most general case, *terminological axioms* are of the form $C \sqsubseteq D$ or $C \equiv D$, where C and D are concepts. The former axioms are called *inclusions* and the latter *equivalences*. An equivalence whose left hand side is an atomic concept is a *concept definition*. We can define the semantics of terminological axioms in a straightforward way. An interpretation $\mathcal{I}$ satisfies an inclusion $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ and it satisfies the equivalence $C \equiv D$ if $C^{\mathcal{I}} = D^{\mathcal{I}}$. $\mathcal{I}$ satisfies a set of terminological axioms if it satisfies all axioms in the set. An interpretation, which satisfies a (set of) terminological axiom(s) is called a *model* of this (set of) axiom(s). Two (sets of) axioms are *equivalent* if they have the same models. A finite set $\mathcal{T}$ of terminological axioms is called a *(general) TBox*. Let N_I be the set of object names (disjoint with N_R and N_C). An *assertion* has the form $C(a)$ (*concept assertion*) or $r(a, b)$ (*role assertion*), where a, b are object names, C is a concept, and r is a role. An *ABox* $\mathcal{A}$ is a finite set of assertions.

Objects are also called individuals. To allow interpreting ABoxes we extend the definition of an interpretation. Additionally to mapping concepts to subsets of our domain and roles to binary relations, an interpretation has to assign to each individual name $a \in N_I$ an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. An interpretation $\mathcal{I}$ is a model of an ABox $\mathcal{A}$ (written $\mathcal{I} \models \mathcal{A}$) if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ for all $C(a) \in \mathcal{A}$ and $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$ for all $r(a, b) \in \mathcal{A}$. An interpretation $\mathcal{I}$ is a model of a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ (written $\mathcal{I} \models \mathcal{K}$) iff it is a model of $\mathcal{T}$ and $\mathcal{A}$.

A concept is in *negation normal form* if negation only occurs in front of concept names. The *length* of a concept is defined in a straightforward way, namely as the sum of the numbers of concept names, role names, quantifier, and connective symbols occurring in the concept. The *depth* of a concept is the maximal number of nested concept constructors. The *role depth* of a concept is the maximal number of nested roles. For brevity we sometimes omit brackets. In this case, constructors involving quantifiers have higher priority, e.g. $\exists r.\top \sqcap A$ means $(\exists r.\top) \sqcap A$.

As we have described, a knowledge base can be used to represent the information we have about an application domain. Besides this *explicit* knowledge, we can also deduce *implicit* knowledge from a knowledge base. It is the aim of *inference algorithms* to extract such implicit knowledge. There are some standard reasoning tasks in description logics, which we will briefly describe.

In *terminological reasoning* we reason about concepts. The standard problems are *satisfiability* and *subsumption*. Intuitively, satisfiability determines if a concept can be satisfied, i.e. it is free of contradictions. Subsumption of two concepts detects whether one of the concepts is more general than the other.

Definition 3 (satisfiability) Let C be a concept and $\mathcal{T}$ a TBox. C is *satisfiable* iff there is an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$. C is *satisfiable with respect to* $\mathcal{T}$ iff there is a model $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}} \neq \emptyset$.

Definition 4 (subsumption, equivalence) Let C, D be concepts and $\mathcal{T}$ a TBox. *C is subsumed by D*, denoted by $C \sqsubseteq D$, iff for any interpretation $\mathcal{I}$ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. *$C$ is subsumed by D with respect to $\mathcal{T}$*, denoted by $C \sqsubseteq_{\mathcal{T}} D$, iff for any model $\mathcal{I}$ of $\mathcal{T}$ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

C is equivalent to D (with respect to $\mathcal{T}$), denoted by $C \equiv D$ ($C \equiv_{\mathcal{T}} D$), iff $C \sqsubseteq D$ ($C \sqsubseteq_{\mathcal{T}} D$) and $D \sqsubseteq C$ ($D \sqsubseteq_{\mathcal{T}} C$).

OWL	DL	DL syntax	semantics
named class	atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
object property	abstract role	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Thing	top concept	$\top$	$\Delta^{\mathcal{I}}$
Nothing	bottom concept	$\bot$	$\emptyset$
intersectionOf	conjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
unionOf	disjunction	$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
complementOf	negation	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
someValuesFrom	exists restriction	$\exists r.C$	$(\exists r.C)^{\mathcal{I}} = \{a \mid \exists b.(a,b) \in r^{\mathcal{I}} \text{ and } b \in C^{\mathcal{I}}\}$
allValuesFrom	universal restriction	$\forall r.C$	$(\forall r.C)^{\mathcal{I}} = \left\{a \mid \forall b.(a,b) \in r^{\mathcal{I}} \text{ implies } b \in C^{\mathcal{I}}\right\}$

Table 1
$\mathcal{ALC}$ syntax and semantics along with corresponding OWL constructs

C is strictly subsumed by D (with respect to $\mathcal{T}$), denoted by $C \sqsubset D$ ($C \sqsubset_{\mathcal{T}} D$), iff $C \sqsubseteq D$ ($C \sqsubseteq_{\mathcal{T}} D$) and not $C \equiv D$ ($C \equiv_{\mathcal{T}} D$).

Subsumption allows to build a hierarchy of atomic concepts, commonly called the *subsumption hierarchy*. Analogously, for more expressive description logics *role hierarchies* can be inferred.

In *assertional reasoning* we reason about objects. The *instance check problem* is to find out whether an object is an instance of a concept, i.e. belongs to it. *Retrieval* is the problem of finding all instances of a given concept.

Definition 5 (instance check) Let $\mathcal{A}$ be an ABox, $\mathcal{T}$ a TBox, $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a knowledge base, C a concept, and $a \in N_I$ an object. *a is an instance of C with respect to $\mathcal{A}$* , denoted by $\mathcal{A} \models C(a)$, iff in any model $\mathcal{I}$ of $\mathcal{A}$ we have $a^{\mathcal{I}} \in C^{\mathcal{I}}$. *$a$ is an instance of C with respect to $\mathcal{K}$*, denoted by $\mathcal{K} \models C(a)$, iff in any model $\mathcal{I}$ of $\mathcal{K}$ we have $a^{\mathcal{I}} \in C^{\mathcal{I}}$.

To denote that a is not an instance of C with respect to $\mathcal{A}$ ($\mathcal{K}$) we write $\mathcal{A} \not\models C(a)$ ($\mathcal{K} \not\models C(a)$).

We use the same notation for sets S of assertions of the form $C(a)$, e.g. $\mathcal{K} \models S$ means that every element in S follows from $\mathcal{K}$.

Definition 6 (retrieval) Let $\mathcal{A}$ be an ABox, $\mathcal{T}$ a TBox, $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a knowledge base, C a concept. The *retrieval $R_{\mathcal{A}}(C)$ of a concept C with respect to $\mathcal{A}$* is the set of all instances of C: $R_{\mathcal{A}}(C) = \{a \mid a \in N_I \text{ and } \mathcal{A} \models C(a)\}$. Similarly the *retrieval $R_{\mathcal{A}}(C)$ of a concept C with respect to $\mathcal{K}$* is: $R_{\mathcal{K}}(C) = \{a \mid a \in N_I \text{ and } \mathcal{K} \models C(a)\}$

Correspondence of OWL and Description Logics

As we move forward in the course of this article, from theoretical foundations and algorithms to practical use cases and real-world applications a shift in terminology is necessary. Decades of research in Description Logics have entered design decisions for OWL, which even results in the fact that a DL knowledge base is "nowadays often called ontology" (as noted by Baader, Ganter, Sertkaya, och Sattler (2007, p. 3)). As we progress, we will use OWL terminology, where appropriate, but keep some notations in Description Logics, especially where the advantages of representation is obvious (e.g.. complex class descriptions). Cf. Table 1 for the most commonly used expressions. For a complete mapping from OWL to Description Logics we refer the interested reader to Horrocks och Patel-Schneider (2003). OWL is based on the description language $\mathcal{SHOIN}$ and OWL 2 will probably be based on $\mathcal{SROIQ}$ (Horrocks, Kutz, & Sattler, 2006).

The Learning Problem in OWL and Description Logics

In this section, we will briefly describe the learning problem in Description Logics. The process of learning in logics, i.e. finding logical explanations for given data, is also called *inductive reasoning*. In a very general setting this means that we have a logical formulation of background knowledge and some observations. We are then looking for ways to extend the background knowledge such that we can explain the observations, i.e. they can be deduced from the modified knowledge.

For learning in Description Logics we can give a more specific description of the learning problem. The background knowledge is a knowledge base $\mathcal{K}$. The goal is to find a definition for a concept we want to call `Target`. Hence the *examples* are of the form `Target`(a) where a is an *example instance*. We are then looking for a concept definition of the form `Target` $\equiv C$ such that we can extend our knowledge base by this definition. Let $\mathcal{K}' = \mathcal{K} \cup \{\texttt{Target} \equiv C\}$ be this extended knowledge base. Then we want that the positive examples follow from it, i.e. $\mathcal{K}' \models E^+$, and the negative examples should not to follow, i.e. $\mathcal{K}' \not\models E^-$. Please note that the description language of the background knowledge can be more expressive than the language of the concept C we want to learn.

When we speak about concepts as possible problem solutions it is useful to introduce some shortcuts for the two main criteria: covering all positive examples and not covering negative examples.

Definition 7 (complete, consistent, correct) Let C be a concept, $\mathcal{K}$ the background knowledge base, `Target` the target concept, $\mathcal{K}' = \mathcal{K} \cup \{\texttt{Target} \equiv C\}$ the extended knowledge base, and E^+ and E^- the positive and negative examples.

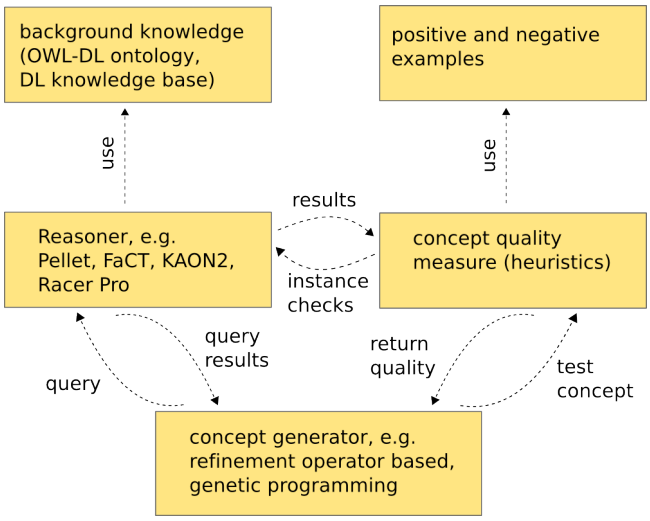

Figure 1. Generate and test approach used in DL-Learner.

C is *complete* with respect to E^+ if for any $e \in E^+$ we have $\mathcal{K}' \models e$. C is *consistent* with respect to E^- if for any $e \in E^-$ we have $\mathcal{K}' \not\models e$. C is *correct* with respect to E^+ and E^- if C is complete with respect to E^+ and consistent with respect to E^-.

Figure 1 gives a brief overview of how the learning problem can be solved by means of a generate and test approach common in Machine Learning. Several concepts are tested during a learning process, each of which is evaluated using an OWL reasoner. The reasoner performs instance checks on the given concept and the examples. Smart algorithms will take the results of those tests into account to suggest further promising concepts. A brief description of the concrete algorithm employed here can be found below.

We implemented the algorithm within the open source framework DL-Learner(Lehmann, 2007), which employs several Machine Learning algorithms for learning complex class descriptions from objects. It uses a modular system, which allows to define different types of components: knowledge sources (e.g. OWL files), reasoners (e.g. DIG interface based (Bechhofer, Mller, & Crowther, 2003)), learning problems, and learning algorithms. DL-Learner is easily extensible by defining additional components. The component, which will be presented in this paper, is the SPARQL and Linked Data knowledge source component.

Learning Algorithm Description

In this section, we will describe the workings of the algorithm for learning complex classes on large knowledge bases.

Before, we referred to Figure 1 for a general overview on how the learning problem in Description Logics can be solved. In Lehmann och Hitzler (2007b), we reported about a concrete algorithm solving the task, which is inspired by Inductive Logic Programming techniques (ILP) (Nienhuys-Cheng & Wolf, 1997). We will give a brief overview of the algorithm in this section to give the reader an intuition about class description learning methods (although the algorithm itself is not a scientific contribution made in this article).

The goal of learning is to find a correct concept with respect to the examples. This can be seen as a search process in the space of concepts. A natural idea is to impose an ordering on this search space and use operators to traverse it. This strategy is well-known in ILP, where refinement operators are widely used to find hypotheses. Intuitively, downward (upward) refinement operators construct specializations (generalizations) of hypotheses.

Definition 8 (refinement operator) A *quasi-ordering* is a reflexive and transitive relation. In a quasi-ordered space $(S, \preceq)$ a *downward (upward) refinement operator* ρ is a mapping from S to 2^S, such that for any $C \in S$ we have that $C' \in \rho(C)$ implies $C' \preceq C$ ($C \preceq C'$). C' is called a *specialization* (*generalization*) of C.

This idea can be used for searching in the space of concepts. As ordering we can use subsumption. (Note that the subsumption relation $\sqsubseteq$ is a quasi-ordering.) If a concept C subsumes a concept D ($D \sqsubseteq C$), then C will cover all examples, which are covered by D. This makes subsumption a suitable order for searching in concepts as it allows to prune parts of the search space without losing possible solutions.

The approach we used is a top-down refinement operator based algorithm. This means that the first concept, which will be tested is the most general concept ($\top$), which is then mapped to a set of more special concepts by means of a downward refinement operator. Naturally, the refinement operator can be applied to the obtained concepts again, thereby spanning up a search tree. The search tree can be pruned when we reach an incomplete concept, i.e. a concept which does not cover all the positive examples. This can be done, because the downward refinement operator guarantees that all refinements of this concept will also not cover all positive examples and therefore cannot be solutions of the learning problem. One example for a path in a search tree spanned up by a downward refinement operator is as follows:

$$\top \rightsquigarrow \texttt{Person} \rightsquigarrow \texttt{Person} \sqcap \exists\, \texttt{participatesIn.Event}$$
$$\rightsquigarrow \texttt{Person} \sqcap \exists\, \texttt{participatesIn.Conference}$$

The heart of such a learning strategy is to define a suitable refinement operator. The refinement operator in the considered algorithm can be found in Lehmann och Hitzler (2007b) and is build on solid theoretical foundations (Lehmann & Hitzler, 2007a). It has been shown to be the best achievable operator with respect to a set of properties (not further described here), which are used to assess the performance of refinement operators. The used refinement operator can reach any OWL class description, i.e. we are guaranteed to find a solution in finite time if one exists.

While the refinement operator defines the search tree, a heuristic decides on which node to apply the refinement operator. Heuristics can take several criteria into account, e.g. accuracy of class description on positive and negative examples, accuracy gain compared to parent node, length of class

description, computational resources needed to apply refinement operator. The heuristic we use combines those criteria. Since the focus of this paper is the fragment selection process, we refrain from a formal description. Going back to Figure 1, the refinement operator is used as concept generator, whereas the heuristics is used to evaluate concept quality, which the learning algorithm uses to decide which concept to try next.

Selection of a Suitable Knowledge Fragment

As detailed in the previous section, the used refinement operator is well designed according to the possible properties a refinement operator for DL can have. The used heuristic for traversing the search space is also highly efficient. Nevertheless, both heavily depend on an OWL reasoner for standard reasoning tasks such as subsumption, instance checks and retrieval. The most commonly used reasoners such as Pellet and Fact++ do not, although highly optimized and efficient, have the ability to scale up to large knowledge bases. Thus, it becomes impossible to use the presented learning algorithm as soon as the target knowledge base reaches a certain size and complexity, with two major problems being initialization time (i.e. to load the data into the reasoner) and the time to answer queries involving complex classes (such as retrieval). However, in order to solve the learning problem it is not necessary to consider the complete knowledge base, but only a fragment that holds enough information to produce good results, while at the same time is small enough to allow efficient reasoning.

Desired Fragment We are looking for a sufficiently small fragment F of an ontology O ($F \subset O$), which contains the example instances E and all relevant information to solve a given learning problem LP. If we can successfully apply the learning algorithm on the fragment yielding the concept C, which satisfies the learning problem, then C should also satisfy the learning problem in the large knowledge base O.

The following example shall briefly illustrate, what can be achieved by our fragment selection approach, before we will explain, in the next sections, in detail, how such a fragment is selected and which parameters are used.

Example 1 (Manual example from Semantic Bible) Here and also in later experiments, we choose the Semantic Bible ontology(Boisen, 2006), because it is a medium sized ontology, contains rich background knowledge and is still manageable by a reasoner as a whole. This enables us to directly compare the results of learning on the fragment to results obtained on the whole knowledge base. We manually choose Archelaus and HerodAntipas, two brothers from the New Testament as positive examples, while we choose God, Jesus, Michael and Gabriel (the archangels) as negative examples. The learning algorithm was then executed twice, once in normal mode, where the whole ontology was loaded into the OWL reasoner (Pellet) and once where first a fragment was selected by our extraction method[1], which was then loaded into Pellet (see Figure 2 for an overview). The 20 best learned classes from both runs (like $\exists$siblingOf.Man or $\exists$siblingOf.$\exists$spouseOf.Human) are with some exceptions identical and, even more important, all 20 classes learned from the fragment yield 100% accuracy on the whole ontology. Table 2 provides details on the Semantic Bible ontology and solving the learning problem on it as a whole or on a fragment. This example is only meant to illustrate the used methods and algorithms. For a full quantitative evaluation the reader is referred to the later sections of this article.

Semantic Bible	Normal	Fragment
No. of classes	49	27
No. of instances	724	60
No. of object properties	29	20
No. of data properties	9	0
No. of subclass axioms	51	25
Time needed for extraction	-	4.2s
Reasoner instantiation time	3.6s	1.3s
No. of reasoner queries	1480	313
Avg. time per query	120ms	2ms
Reasoning time	178.0s	0.8s
Learning time without reasoning	0.4s	0.1s
Total time	182.0s	6.4s

Table 2
Manual example to give a first glance at the presented method. Note that not only are reasoner queries faster on average, but also the number of queries needed is significantly smaller (due to the smaller search space.)

What properties should the fragment have?

In the previous section, we clearly stated what a desired fragment is. It allows fast reasoning and the learned classes achieve (approximately) the same accuracy, when validated versus the original knowledge base. We now take a closer look at what should be included in the fragment for the learning algorithm to work efficiently while still achieving good results. The first obvious inclusions are the example instances themselves. Secondly, all classes of the example instances and all related instances (via an object property) are necessary. Note that the property between instances will always be included implicitly, when we add related instances to the fragment. Up to now, the fragment consists of the combined Concise Bound Descriptions (CBD (Stickler, 2004)) of the example instances. The information contained is clearly not yet sufficient to learn complex classes. The most complex class definitions derivable when using only CBDs are of the form $\mathtt{C} \sqcup \mathtt{R}$ or $\mathtt{C} \sqcap \mathtt{R}$, where $\mathtt{C}$ is any conjunction or intersection of classes and $\mathtt{R}$ is a conjunction or intersection of unqualified property restrictions of the form $\exists\, \mathtt{property}.\top$.

[1] The ontology was loaded into a local Joseki triple store and queried with SPARQL.

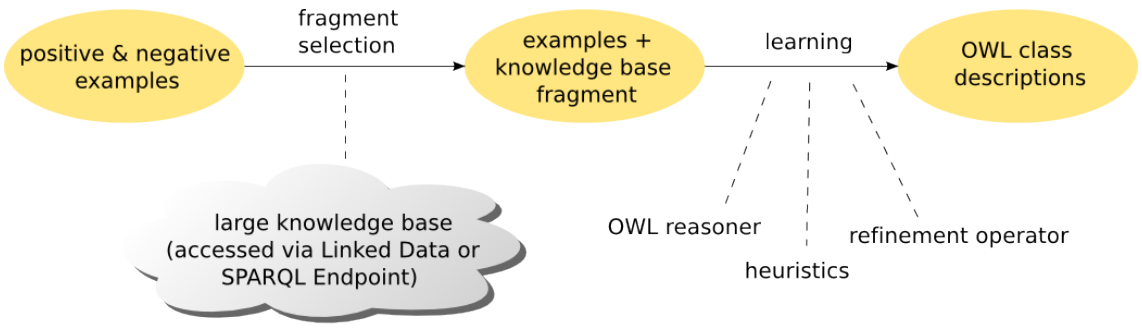

Figure 2. Process illustration: In a first step, a fragment is selected based on instances from a knowledge source and in a second step the learning process is started on this fragment and the given examples.

While this is of course often not sufficient, it still represents the smallest sensible fragment, where it is possible to learn classes at all with the trade-off scale shifted away from high learning accuracy towards efficient light-weight reasoning. Beyond this point, selection and extraction of information becomes more complicated. Our aim here is solely to show that it is possible to produce class descriptions, which are useful for the original knowledge base. One of the major influences on the validity of learning results stands in direct relation to the possible deductive inferences on the fragment.

Since reasoning in Description Logics is monotonic, the inferences obtained on a fragment of an ontology are also valid for the ontology as a whole (soundness). However, not all possible inferences might be obtainable on the fragment and reasoning thus can be viewed as being "incomplete", e.g. an instance check $C(a)$ answered negatively on the fragment (the reasoner cannot deduce that a is instance of C) might be answered positively on the whole ontology.

As a consequence, the learning problem might be solved incorrectly, because the learning algorithm implicitly assumes that the underlying reasoning methods are complete. So, if for a class description C the resulting answer set of a retrieval will contain all positive example individuals and none of the negatives, it will present C as a solution. Due to the issues explained above, however, the previously not covered negative example individuals might now be an instance of C when the whole ontology is considered. Thus a correctly learned class definition might turn out to be inconsistent (cf. Def 7). We tackle this problem by trying to avoid such cases through selection of an ontology fragment containing all relevant information as described in detail below. Furthermore, in most application scenarios the learned class descriptions (and/or its implications) are reviewed by a human expert. Because reasoning on large knowledge bases remains impossible, it is hard to give exact measures of the extend to which the negative example coverage problem occurs on very large knowledge bases. However, we will later perform benchmarks on the medium sized Semantic bible ontology and can draw conclusions from those observations.

Extension of CBDs

In the following, we will give a list containing which information can be additionally extracted to learn more complex classes than with CBDs. We assume that the CBDs of all example individuals are already included in the fragment. On this basis, the following list shows in detail, which information can additionally be included to learn more complex class descriptions:

1. *Direct Classes* Retrieving direct classes for all instances in the fragment, that do not yet have any types, will allow to learn qualified property restrictions of the form $\exists$ `property`.`C`.
2. *Increased Property Depth* A further extension of the CDBs by instances, which are related to an instance, which is again related to an example instance via an object property etc., will enable to learn classes with nested property restrictions of the form $\exists$ `propertyA`.$\exists$ `propertyB`.$\top$. This extension can be continued such that it is possible to learn even deeper nested property restrictions.
3. *Hierarchy* Retrieving all superclasses of all existing classes in the fragment and the corresponding hierarchy, will improve the efficiency of the learning algorithm, because it 1) optimizes the search tree with the help of the subsumption hierarchy and 2) enables the usage of those classes in learned descriptions.
4. *Class Definitions and Axioms* Extracting information for all classes in the fragment like definitions via `owl:equivalentClass` or disjointness, etc., will permit the learning algorithm to make use of this valuable background knowledge, e.g. knowing whether classes are disjoint speeds up the reasoning and learning process. Other axioms are of course necessary to draw conclusions. In general, extracting class related axioms reduces the above mentioned negative example coverage problem.

The items above directly influence how complex learned classes can be. We continue this list and present in detail, which information influences reasoning on the fragment.

5. *Complex Descriptions* All the points in the list mentioned above improve the ability of the reasoner to deduce whether an object is instance of a complex class description, which directly relates to the ability to learn those class descriptions.
6. *Explicit Property Information* Retrieving characteristics of object properties, such as `owl:SymmetricProperty`, domain/range, and the property hierarchy allows more inferences as the fragment is handed to the OWL reasoner.

7. *Inferred Property Information* Because reasoning is normally deactivated in SPARQL endpoints or Linked Data sources, reasoning on the fragment could be improved by also including instances that are related to the example instances via a symmetric, inverse or transitive property. Nevertheless to include such properties, which only become "visible" after inference, extensive and costly discovery methods need to be used. Because we follow the general aim to improve performance we accepted this trade-off in favor of a faster extraction procedure.

8. *Complete Class Definitions* There also is the possibility that classes which are contained in the fragment might occur somewhere in the ontology on the right hand side of a class definition (e.g. $SomeClass = AnotherClass \sqcup ClassInFragment$). As in the item above, the cost to find such information can become quite large. To completely extract all such information all class axioms would need to be evaluated. As above, such information requires an intensive search, which is why we refrained from including it, although it might become a parameter in future releases. .

Another requirement for the fragment is that it should be in correct OWL-DL, so that it can be processed by OWL Reasoners.

Extraction Methods

Although the extraction algorithm, we are about to present, was developed to fit the needs of the class learning algorithm, it can basically be applied in any context, where a set of individuals needs to be analyzed with respect to given background knowledge (a circumstance often required in Machine Learning). The size of the fragment can be controlled in a flexible way to regulate the trade-off between complete reasoning and performance. Especially the Linked Data paradigm gives rise to questions concerning reasoning and performance, which cannot merely be answered by optimizing existing reasoning algorithms and using more powerful hardware. Linked Data connects facts across knowledge bases. Due to limited computational resources, we have to decide how far links into other knowledge bases or within the knowledge base itself should be followed and how we retrieve relevant data. In the course of this section, we will describe the extraction algorithm independently from the actual knowledge source, because it is not bound to a certain formalism and works for several variations such as Linked Data or SPARQL endpoints. The actual data provisioning is merely a technical question of implementation. After this section though, we will describe our implementation for SPARQL endpoints, which contains optimizations of the method.

The algorithm traverses the RDF graph of the original knowledge base recursively starting from the example instances. The parameters of the algorithm allow to control the size of the fragment, so that each point in the above mentioned list (information necessary to learn more complex classes) can be included or excluded. Additionally, filters are used to gain even more flexibility during the extraction of the fragment. The filters are applied to the lowest possible level in the data acquisition and thus we will start with describing the acquisition interface.

Definition 9 (Tuple acquisition interface) A tuple acquisition function of the form $acquire_{KB}$ (resource , predicateFilter, objectFilter, literals) takes as input a resource, a list of unwanted namespaces or URIs for predicates, a list of prohibited namespaces or URIs for objects and a boolean flag *literals*, indicating whether datatype properties should be retrieved. According to its implementation it will retrieve all triples from the knowledge base KB, whose subject is *resource*. The triples (s,p,o) will then be filtered, so that all triples are removed, which contain a namespace or URI from the predicateFilter list as a predicate (same accounts for objectFilter). If *literals* is false, all triples with datatype properties and literals will be removed. It returns a set of tuples of the form (p,o), where (p,o) are the resources (properties and objects) of the remaining triples. We will simply use *acquire*(resource) when the context is clear.

The filters provide the possibility to create a fine-grained selection of the extracted information. They are especially useful for multi-domain knowledge bases such as DBpedia, where retrieving information unfiltered will lead to an unnecessary large fragment. In our case, we avoid retrieving information, that is not important to the learning process. In some cases, we do not want to use datatype properties, so they can be omitted by the literal parameter shown above. The predicate filter removes properties that are not important (e.g. when working with DBpedia we can use this to filter properties pointing to web pages and pictures). The same is true for the object filter, i.e. it filters uninteresting objects in triples.

The configuration of filter criteria is in most cases optional and is clearly content-driven. While the parameters of the extraction algorithm steer the structural selection of knowledge, filters work at a lower abstraction level. The configuration depends on the particularities of the knowledge source and the intended task and can be optimized for the application. The choice can, on the one hand, add another edge to performance and, on the other hand, allow a content-aware filtering. If the knowledge base makes use of different structural hierarchies such as DBpedia, which uses YAGO classes (Suchanek, Kasneci, & Weikum, 2007) and also the SKOS vocabulary(Miles & Brickley, 2005) combined with its own categories, one of the hierarchies can be selected by excluding the other. Adding the SKOS namespace (*http://www.w3.org/2004/02/skos/core*) to the predicate and object filter list will guarantee that the fragment will be free of SKOS vocabulary. A Social Semantic Web application for example might be especially interested in FOAF and thus would filter other information.

After having defined the filters for the respective knowledge source, a recursive algorithm (see Algorithm 1) extracts relevant knowledge for each of the instances in the example set using *acquire(instance)*. The objects of the retrieved tuples (p,o) are evaluated and manipulated and used to further extract knowledge (*acquire(o)*) until a given recursion depth is reached. The process is illustrated in Figure 3.

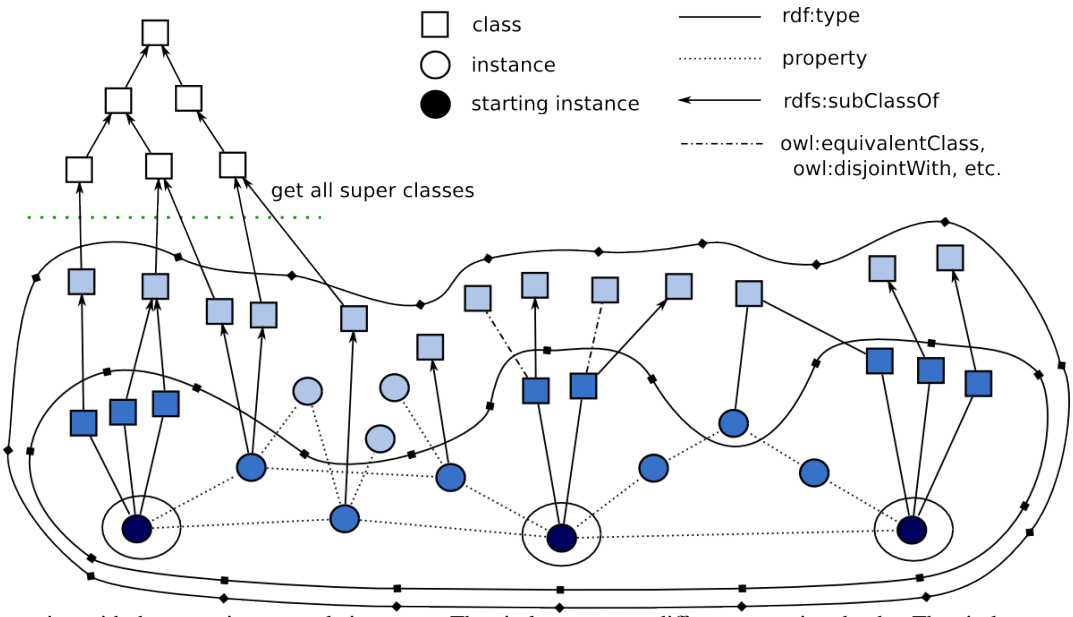

Figure 3. Extraction with three starting example instances. The circles represent different recursion depths. The circles around the starting instances signify recursion depth 0. The larger inner circle represents the fragment with recursion depth 1 and the largest outer circle with recursion depth 2.

The algorithm remembers valuable information that is later used to convert the fragment to OWL DL, which we will describe later.

Parameters In the following, we will relate the influences of the algorithm parameters to the list in the previous section.

The parameter *recursion depth* has the greatest influence on the number of triples extracted and included in the fragment. If set to 0 the fragment will only consist of the example instances. A recursion depth of 1 means, that only the directly related instances and classes are extracted, which results in the combined CBDs of all example instances. A recursion factor of 2 extracts all direct classes of the example instances, their direct super classes and all directly related instances and their direct classes and directly related instances. This will enable the algorithm to learn nested property restrictions (2. *Increased Property Depth*), includes some hierarchy information (3. *Hierarchy*), allows qualified property restrictions for unnested properties (1. *Direct Classes*) and includes definitions of classes directly connected to the starting individuals (4. *Class Definitions and Axioms*).

We avoid following cycles, which often occur when encountering inverse properties, `owl:sameAs`, `owl:equivalentClass` etc., by storing all resources already visited. If the object is a blank node, we will not decrease the recursion counter until no further blank nodes are retrieved.

If we use all existing instances of the original knowledge base as starting seeds with a sufficient recursion depth, the algorithm will extract the whole knowledge base with the exception of unconnected resources, which in most cases barely contain useful information.

To cover other points on the list above, the algorithm retrieves additional information in a post-processing step, which can be switched on and off independently.

Close after recursion For each instance in the fragment that does not yet have any classes assigned to it, classes are retrieved and added to the fragment (cf list 1. *Direct Classes*).

Get all super classes For all classes in the fragment, all super classes are retrieved and the hierarchy is extracted (cf. list 3. *Hierarchy*). Additionally all class definitions are included (cf. list 4. *Class Definitions and Axioms*).

Get all property information For all object properties, types, Domain, Range and the property hierarchy will be retrieved (cf. list 6. *Explicit Property Information*).

Depending on the expected complexity of class descriptions (in particular their property depth) and the density of the background knowledge, a recursion depth of 1 or 2 (with all post-processing steps enabled, otherwise 2 or 3) represents a good balance between the amount of useful information and the possibility to reason efficiently.

The retrieved triples can be further manipulated by means of user defined rules. For example, vocabularies that resemble OWL class hierarchies but use different identifiers (such as SKOS) can be mapped to OWL class hierarchies. We also used this technique to embed tags or other structurally important individuals in a class hierarchy in order to enable learning class descriptions. Additional information can be easily inserted in this step of the extraction. The function `manipulate` does not only allow for manipulation, but can also be used to retrieve and include information from other knowledge bases. Even a new extraction can be started based on the current resource.

Algorithm 1: Knowledge Extraction Algorithm

```
1  Function: extract
   Input: recursion counter, resource, predicateFilter,
          objectFilter, literals
   Output: set S of triples
2
3  if recursion counter equals 0 then
4      return ∅
5  S = empty set of triples;
6  // for acquire see Definition 9
7  resultSet = acquire(resource, predicateFilter,
   objectFilter, literals);
8  newResultSet = ∅ ;
9  foreach tuple (p,o) from resultSet do
10     newResultSet = newResultSet ∪
       manipulate(typeOfResource,p,o);
11     // the function manipulate allows the alteration
12     // based on the semantic information of the retrieved
13     // URIs and evaluates the type of the newly
14     // retrieved resources
15 create triples of the form (resource,p,o) from the
   newResultSet ;
16 add triples of the form (resource,p,o) to S ;
17 foreach tuple (p,o) from the newResultSet do
18     if o is a blank node then
19         S = S ∪ extract(recursion counter, o,
           predicateFilter, objectFilter, literals);
20     else
21         S = S ∪ extract(recursion counter -1 ,o,
           predicateFilter, objectFilter, literals);
22 return S
```

OWL DL Conversion of the Fragment

The extracted knowledge has to be altered to adhere to OWL DL for processing, which means explicitly typing classes, properties and instances. Since the knowledge base might not provide (correct) typing information for all individuals, we infer typing information for newly retrieved resources. We follow Bechhofer och Volz (2004), who mention an approach, that is based on the idea that if the type of a triple's subject is known, we can infer the type of the object by analyzing the predicate. Since we always start from instances, we possess additional information and therefore are able to extend the rules mentioned in Bechhofer och Volz (2004, pp. 673-674). Given a triple (s, p, o) we can draw the following conclusions:

- If s is an instance and p is `rdf:type` then o is a class.
- If s is an instance, p is not `rdf:type`, and o not a literal then o is an instance.
- If s is a class then o is a class, unless the knowledge source is in OWL Full, in which case we can configure DL-Learner to either ignore such statements or map `rdf:type` (between classes) to `rdfs:subClassOf`. All properties are then ignored except those in the OWL vocabulary having `owl:Class` as range.
- p is an object property if o is a resource and p is a datatype property if o is a literal.

With the help of these observations, we can type all collected resources iteratively, since we know that the starting resources are instances. Thus, we presented a consistent way to convert the knowledge fragment to OWL DL based on the information collected during the extraction process. Due to the comparatively small size, deductive reasoning can now be applied efficiently, allowing the application of machine learning techniques.

SPARQL implementation of Tuple Acquisition

In this section, we will briefly explain how the tuple acquisition interface is implemented for SPARQL endpoints efficiently. The basic pattern is of the form {<resource> ?p ?o} according to the function *acquire(resource)*, which returns a tuple (p,o). The remaining parameters are appended using the FILTER keyword as in the example below. To disburden the SPARQL endpoint, caching is used to remember SPARQL query results which were already retrieved. The extraction algorithm's performance for non-local endpoints is mainly determined by the latency for retrieving SPARQL results via HTTP.

Example 2 (Example SPARQL query on DBpedia) In this example we show how we filter out triples using SKOS and DBpedia categories, but leave YAGO classes. Furthermore, links to websites and literals are filtered out.

```
SELECT ?p ?o WHERE {
  <http://dbpedia.org/resource/Angela_Merkel> ?p ?o.
  FILTER (
    !regex(str(?p),
        'http://dbpedia.org/property/website')
    && !regex(str(?p),
        'http://www.w3.org/2004/02/skos/core')
    && !regex(str(?o),
        'http://dbpedia.org/resource/Category')
    && !isLiteral(?o) ). }
```

More optimizations include nested queries according to recursion depth in such a way that it is only necessary to execute one query per example instance. When retrieving the class hierarchy (*Get all superclasses*) already extracted subclass and other class axioms are remembered and not queried a second time. Because blank nodes in SPARQL result sets do often not relate to the internal blank nodes of knowledge bases (they are iteratively numbered for each result set according to the specification), we use a backtracking technique and assign internal blank node ids.

The implementation of other tuple acquisitors is far simpler. Especially Linked Data can be extracted by just a HTTP request, while the filters are applied after the request. The great advantage of the Linked Data tuple acquisitor is that it allows for cross-boundary acquisition of tuples from different knowledge bases without further configuration and thus enables cross knowledge base accumulation of knowledge.

Usage Scenarios

Instance Data Analysis

The learning algorithm can be used to analyze instance data. With more data on the web, the number of possible applications will increase. We briefly describe two scenarios using GovTrack (Tauberer, 2008) and MusicBrainz (Swartz, 2002).

Last.fm [2] is the worlds largest social music platform. For a given username, we can get information about the last songs a user listened to as RDF[3]. The songs contain ZitGist[4] `owl:sameAs` links, which again refer to MusicBrainz. MusicBrainz is a very large open source music metadata base with plenty of informations about musicians. We want to obtain a description of the last artists a user has listened to. We pick the MusicBrainz URIs of those artists as positive examples and randomly selected artists as negative ones. To improve the learning process, we converted the MusicBrainz tag cloud into a class hierarchy on the fly by adding a property mapping entry, executed in the *manipulate* function (see Algorithm 1). With the positive examples "Genesis", "Children on Stun", "Robbie Williams", and "Dusty Springfield", and as negative ones "Madonna", "Cher", and "Dreadzone" we learned the description $\texttt{UK-Artist} \sqcup (\texttt{Rock-Genre} \sqcap \exists \texttt{bioEvent.Death})$. This gives the user feedback (when expressed in natural language) and allows the system to suggest similar songs, e.g. UK-Rock in this case. As there is a variety of existing media players with MusicBrainz support[5], a learning application could be integrated as plugin into those and employ the Semantic Web to provide descriptions of a users favorite artists, songs, etc.

A similar example for instance analysis can be given for GovTrack, a data set about the US congress containing more than 10 million facts. Amongst other uses, we can apply the presented techniques to learn about the interests and working areas of politicians. To do so, we chose a US senator and queried the GovTrack SPARQL endpoint to return all bills, which were sponsored by him or her. We used this as positive examples and applied DL-Learner. As before with the MusicBrainz tags we performed an enrichment step by converting the subject strings of the bills (financial matter, education) to concepts. We queried the Cyc Foundation browser, which uses OpenCyc(Cycorp, 2008) as background knowledge, to find suitable concepts and integrated them in a hierarchy. As a result, we could see which topics a senator is most interested in and who are cosponsors in bills sponsored by a senator. In this case, the advantage of DL-Learner is to reduce the often considerable amount of information about a senator to a concise approximate description.

Improving Data Quality

For large knowledge bases, in particular those developed by an Internet community, it is often difficult to maintain a proper classification scheme. A typical example are the DBpedia classification schemata. There have been various attempts to create a classification hierarchy for DBpedia using e.g. the Wikipedia category system as input. Even with good extraction techniques, human errors cannot be completely eliminated and thus articles are assigned to wrong categories or to superfluously many categories. Class learning can be useful in this scenario to learn a complex class C as a possible definition of an existing class A and then verifying whether the instances of C coincide with those of A. Also class descriptions can be used to spot data inconsistencies in instance data and to make suggestions for missing instances. In Example 3 we show how we can successfully apply the algorithm on DBpedia in different ways to either improve the class schemata, spot inconsistencies in existing Categories or make suggestions to Wikipedia editors. Note that a detailed evaluation of the used methods can be found in the next section. Here we just evaluated the possibilities for future applications.

Example 3 (Re-Learning Wikipedia Categories) We choose 4 Wikipedia categories (Best Actor Academy Award winners, Prime Ministers of the UK, Fluorescent Dyes, Islands of Tonga), which are included in the DBpedia dataset. These categories as well as the belonging individuals are currently manually maintained by the Wikipedia community, who would benefit greatly from a list of suggestions for missing instances or missing infobox properties. To provide such suggestions a fully automated process is required, when re-learning these categories. While the choice of positive example instances is trivial (all instances assigned to the categories via `skos:subject`), the selection of negative examples is not. If the instances are from a completely different domain or randomly chosen, the correct class descriptions are likely to be quite simple. The negative examples were thus obtained by retrieving instances that share the same YAGO classes as the instances in the category. We then randomly selected from this set, such that the number of positive and negative examples were equal. The learning process was then started. The assignment of articles to categories in Wikipedia is done manually by Wikipedia editors and are therefore inconsistent (some categories seem to be confused with tags). The category of British Prime Ministers, for example, also includes instances like *Anthony Eden hat* (a typical hat form worn by Anthony Eden) or *Supermac* (a comic strip about Harold Macmillan). We therefore allowed 20% noise in the accuracy when learning on the fragment. The learned class descriptions were used to classify the positive examples in two groups: correctly assigned to the category and incorrectly assigned. We then manually checked these two sets as a Wikipedia editor would do and compared the classification with the information contained in the Wikipedia article.

The results, which are shown in Table 3, give a first glance at how useful the generated sets can be for Wikipedia authors. A retrieval of learned concepts (see below for explanation) on DBpedia can further find missing instance.

[2] *http://www.last.fm/*

[3] via http://dbtune.org/last-fm/$username (description at *http://dbtune.org/last-fm*)

[4] http://www.zitgist.com/

[5] see *http://en.wikipedia.org/wiki/MusicBrainz*

Wikipedia Categories	Prime Min.	Best Actors	Dyes	Tonga
Total number	71	75	56	50
Correct	53(1)	66(0)	34(0)	50(0)
Incorrect	18(0)	9(3)	22(7)	0(0)
Accuracy	98%	96%	88%	100%

Table 3
The table shows a probe of the automatic re-learning method for classes. Sets were evaluated manually, falsely classified individuals in brackets

Since the above described process is fully automated (automatic example choice and concept selection), it can be used to conduct data mining automatically. The retrieved lists could support Wikipedia users, when editing lists and make suggestions about missing entries. Also the automatically discovered inconsistencies in DBpedia could contribute to future releases of DBpedia itself.

SPARQL based Retrieval To validate the results in the described scenario above, we assume that we can retrieve all instances of a learned concept. Usually this is a typical reasoner task. However, as mentioned before, it would be too time-consuming to load the complete knowledge base in a reasoner and pose a retrieval query for the learned concept. A way to solve this problem is to use one or more SPARQL queries to obtain an approximation of the retrieval. We can draw on other work in this area here. The open source project SMART (Battista, Villanueva-Rosales, Palenychka, & Dumontier, 2007) implemented a mapping, which they call DL2SPARQL, to query large knowledge bases. It can be easily tested via their online demonstrator[6]. Other work in the area of efficient approximate inferences for Description Logics is also applicable.

Usage for Navigation

Large knowledge bases are very difficult to navigate and explore for end users, in particular in cases with large TBoxes (schema) *and* large ABoxes (instance data). When users search for interesting knowledge with respect to a certain task, they are often able to find interesting objects by searching, browsing or remembering certain objects. However, users usually will not be able to use the full complexity of a knowledge base for posing sophisticated queries corresponding to their enquiries. In these situations class learning can help to suggest high level concepts, thereby allowing the user to gain new insights and explore other relevant objects, which are otherwise hard to find. As an example we choose the DBpedia SPARQL endpoint again, as it is a multi-domain ontology, which could typically be used for research on a certain topic. A user may browse the knowledge through a user interface, which implicitly or explicitly detects some articles, which are relevant for the current enquiry and others which are not. These can be fed into the DL-Learner system (possibly asynchronously called via AJAX in a web application scenario) as positive and negative examples. An example is given below:

Example 4 With the help of class navigation we try to relate certain ancient Greek mathematicians to mathematicians throughout history, that have similarities. Interesting articles are: Pythagoras, Philolaus, Archytas (positive examples)
Uninteresting articles: Socrates, Plato, Zeno of Elea (negative examples)
In this first run(a) we deduce the class *yago:Mathematician* retrieving more than 2000 instances from DBpedia. Those retrieved instances can further be ranked according to certain keywords or rules. We add one of those instances (Democritus) to the negative example set and learn the class description $Theorist \sqcup (Mathematician \sqcap Physicist)$ in the next run (b), with which we retrieve slightly above 1000 instances from DBpedia. By adding *Aristoxenus* to the negative examples, the algorithm now (c) presents the class description (among other similar alternatives, which we omitted here) $Believer \sqcup (Mathematician \sqcap Physicist)$. The number of resulting instances from DBpedia shrank to the human manageable size of 159. This list reveals a categorical similarity between the now 8 chosen examples and the instances that belong to the same learned class, containing *Archimedes, Aristotle, Blaise Pascal, Carl Friedrich Gauss, Christian Doppler, Galileo Galilei, Gottfried Leibniz, Isaac Newton, Leonhard Euler, Thales*, just to mention a few famous persons from this list (we might add, that the real value are the not so famous and obvious instances on this list, which are generally harder to identify in a large set of data.).

The obtained class descriptions mentioned in the example can be converted into natural language and shown as navigation links to the user. Hence, a user interface can present related objects to a user and also tell why they are related.

Telling the difference

As we have seen in Auer och Lehmann (2007), DBpedia can provide answers to questions such as 'What Have Innsbruck and Leipzig in Common?'. With the algorithm we can now provide answers to even more sophisticated questions in a minimal use case scenario. We can ask the difference between two instances using them as positive and negative examples for the learning algorithm, thus enabling a user to gain a quick insight without tedious manual searching. The following example shows, how quick and precise answers can be retrieved. Most of the classes are not directly related to the instances and would normally require reasoning methods to be retrieved. The time for extraction, reasoning and learning was slightly over one second for each example.

Example 5 (Hillary Clinton vs Angela Merkel)
We queried DBpedia and used a filter that only leaves YAGO classes; we switched both instances for each learning problem:

Classes that Angela Merkel belongs to, but Hillary Clinton does not (a):

[6] *http://134.117.108.147:8181/smart/query.jsf*

GermanWomenInPolitics, Communicator, Negotiator, Representative, HeadOfState, Chancellor, ChancellorsOfGermany, CurrentNationalLeaders, Head, CurrentFemaleHeadsOfGovernment, FemaleHeadsOfGovernment, LeadersOfPoliticalParties, GermanChristianDemocratPoliticians, PeopleFromHamburg, Scientist, GermanScientists

Classes that Hillary Clinton belongs to, but Angela Merkel does not(b):
Achiever, FirstLady, FirstLadiesOfTheUnitedStates, Professional, Educator, Academician, AmericanLegalAcademics, Lawyer, ArkansasLawyers, AmericanWomenInPolitics, Advocate, Democrat, NewYorkDemocrats, Contestant, Opposition, CongressionalOpponentsOfTheIraqWar, Intellectual, Scholar, Alumnus, WellesleyCollegeAlumni

Evaluation

The evaluation is split into two parts. In the first part, we evaluated the performance of the SPARQL retrieval component and the learning algorithm. The results are depicted in Figure 4. We randomly selected ten YAGO classes in DBpedia and retrieved instances that belonged to the class as positive examples and then selected the same number of negative examples from a super class. We performed an extraction with varying recursion depth, which is the most important factor influencing performance, and recorded the following values: number of triples extracted (left figure), time needed for extraction (right figure, lower line of each color), and total time needed for extraction and learning (right figure, upper line of each color). Please note that a recursion depth of e.g. two includes all instances at distance smaller or equal two plus the complete class hierarchy spawned by these instances. The optional parameter *Get all superclasses* and *Close after recursion* were enabled during the post-processing. Each point in the figure is an average over 10 runs and was obtained using a Virtuoso DBpedia mirror on our local network running on a 2.4 GHz dual core machine with 4 GB memory.

We can see that the curves for the time of extraction and learning in the right figure is equally or less steep than the increase in number of extracted triples in the left figure. The time for the learning process increases with more examples used, not only because of the increased time needed for reasoning but also due to the fact that the learned class descriptions tend to get more complex for a higher number of examples. Overall, we achieved typical total learning times on a very large and dense (more than 8 properties associated to an instance on average) DBpedia knowledge base of a couple of seconds. Performance could be improved further by merging several SPARQL queries into more complex ones such that the triple store can make use of further internal optimization routines.

In the second part of our evaluation we measured the validity of learned class descriptions on the fragment, when compared to the whole ontology. As mentioned before, we choose the Semantic Bible ontology (Boisen, 2006) as target, because it is a medium sized ontology, contains complex background knowledge and is still manageable by a reasoner as a whole. It consists of 49 classes, 724 instances, 29 object properties and 9 data properties (4350 axioms total) and is in OWL-DL (but not OWL-Lite). Also worth mentioning is the large size of object and data property axioms (Domain: 35, Range: 35, Inverse: 17, Symmetric: 6, Subproperty: 12, Functional: 4). To objectively compare the fragment selection approach with the normal approach we randomly selected 100 different sets of learning problems with 10 instances each (5 positive example instances and 5 negative example instances)[7] and conducted the experiments with the same learning algorithm configuration and the same underlying reasoner (Pellet). In the first 4 experiments (S_10s, N_10s, S_100s, N_100s) the learning algorithm was stopped after a fixed time period (10 seconds and 100 seconds) and the best learned concept so far was validated versus the whole ontology. In the remaining 4 experiments (S_1000, N_1000, S_10000, N_10000) the algorithm was stopped after a fixed number of concept test (cf. Figure 1, generate and test approach) independently of time needed. The fragment was extracted with the following parameters: recursion depth 2, close after recursion enabled, get all superclasses enabled, get explicit property information, no filters, literals allowed. The result can be viewed in Table 4.

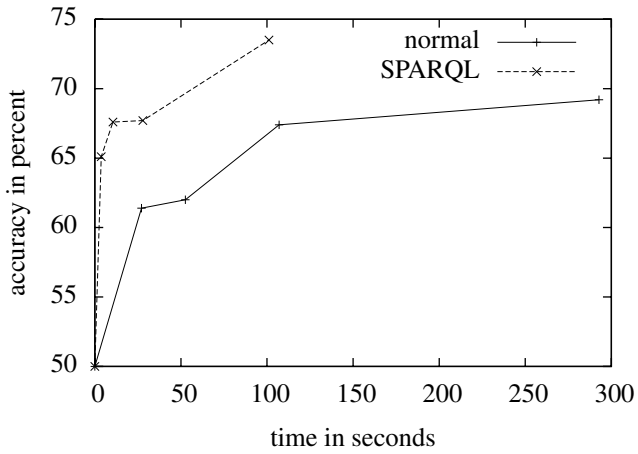

Figure 5. Time vs. Accuracy for learning on the Semantic Bible ontology. The two lines are for using only a fragment of the ontology or using the complete ontology.

The setup of the experiment is meant to answer two questions. First, we wanted to know how large the actual error is, if the fragmentized approach returns a learned class and analyze if we correctly predicted the type of error that can occur and second, we wanted to compare speed performance (fragment vs. whole).

Because the learning algorithm uses top-down refinement and ignores all class descriptions that do not cover all positive examples, the accuracy for positive examples only is always

[7] Random selection is different from real life problems. However, it is sufficient to gain some insights w.r.t. scalability.

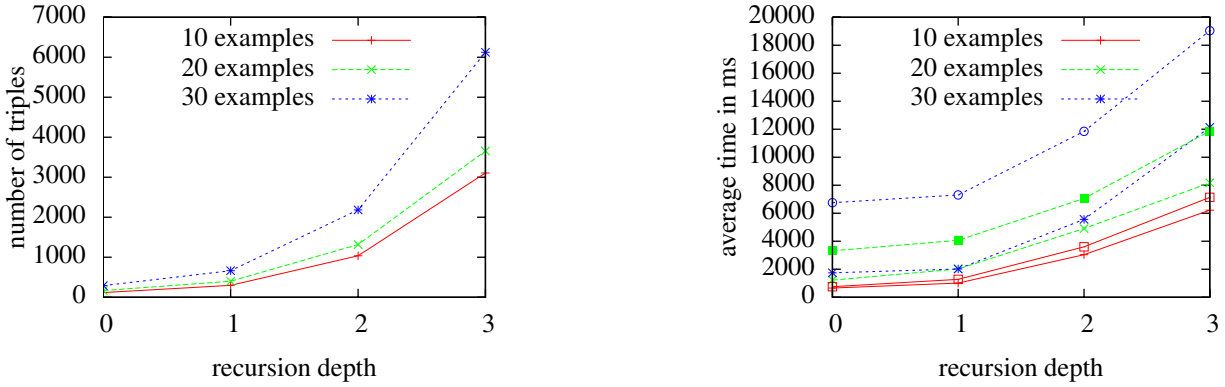

Figure 4. Left: extracted triples depending on recursion depth and number of examples. Right: extraction time needed depending on recursion depth and number of examples - for each color the lower line is the time needed for extraction while the upper line is the total time (including learning).

Semantic Bible	S_10s	N_10s	S_100s	N_100s	S_1000	N_1000	S_10000	N_10000
acc fragment(%)	67.8 (±15.5)	61.4 (±12.0)	73.7 (±13.7)	67.4 (±14.6)	65.3 (±14.7)	62.0 (±13.0)	67.9 (±15.5)	69.2 (±15.0)
acc whole (%)	67.6 (±15.4)	61.4 (±12.0)	73.5 (±13.7)	67.4 (±14.6)	65.1 (±14.6)	62.0 (±13.0)	67.7 (±15.4)	69.2 (±15.0)
acc pos (%)	100.0 (±.0)	100.0 (±.0)	100.0 (±.0)	100.0 (±.0)	100.0 (±.0)	100.0 (±.0)	100.0 (±.0)	100.0 (±.0)
acc neg (%)	35.2 (±30.9)	22.8 (±24.0)	47.0 (±27.4)	34.8 (±29.2)	30.2 (±29.2)	24.0 (±26.1)	35.4 (±30.9)	38.4 (±30.0)
extraction time	1.2s (±.4s)	.0s (±.0s)	1.3 (±.7)	.0s (±.0s)	1.1s (±.3s)	.0s (±.0s)	1.2s (±.4s)	.0s (±.0s)
reasoner init time	.1s (±.1s)	.2s (±.0s)	.0 (±.1)	.3s (±.0s)	.0s (±.0s)	.3s (±.2s)	.1s (±.0s)	.3s (±.0s)
learning time	10.5s (±.7s)	27.1s (±7.1s)	102.3 (±4.7)	107.0s (±16.9s)	3.7s (±2.3s)	52.6s (±56.9s)	27.9s (±14.1s)	292.8s (±92.7s)
axiom number	726 (±221)	4350	726 (±221)	4350	726 (±221)	4350	726 (±221)	4350
desc. length	3.8 (±3.0)	2.2 (±1.6)	5.5 (±3.8)	3.6 (±2.7)	3.2 (±2.5)	2.4 (±1.8)	3.6 (±2.9)	4.1 (±2.8)
desc. depth	2.1 (±1.1)	1.6 (±.7)	2.8 (±1.5)	2.2 (±1.2)	1.9 (±1.0)	1.6 (±.8)	2.2 (±1.2)	2.3 (±1.2)

Table 4
The table shows the statistics for the fragment selection (S) approach compared to the "normal"(N) usage of the learning algorithm. We tested fixed runtime (10 seconds and 100 seconds) and fixed number of concept tests (1000 and 10000). All values are averaged over the same 100 example sets , standard deviation in brackets. A 2.4 GHz dual core machine with 4 GB memory was used and the fragment was retrieved via SPARQL from a local Joseki endpoint.

stable at 100%. This is also true for the fragment because of monotonicity of Description Logics. The small error of 0.2% occurred, as predicted, when previously not covered negatives were covered in the whole ontology. We manually checked the data and found that a part of the learned class description ($Object \sqcup \exists locationOf.\top$) contained an inverse functional property with only an inbound edge to the example instance, which is not covered on purpose by our extraction method (cf. list 7. Inferred Property Information).

The low overall accuracy of the class descriptions (only 60% to 70%) is due to the schematic similarity between random sampled individuals, which made it impossible to induce sensible class descriptions. For about 10% of the learning problems all 8 experiments did not return a better class description than $\top$ with accuracy of (50%). The high description depth, length and runtime are also a measure that the sampled learning problems were not trivial in general and are difficult to solve.

The speed gain of the fragmentized approach is obvious and can be seen in Figure 5. We would like to note again that we choose the Semantic Bible ontology for evaluation. The real target of the fragment selection approach are even larger knowledge bases, which currently only support minimal reasoning mechanisms, if any. The experiments showed an increase in speed by roughly the factor 10 without losing quality. Even more so the highest accuracy (73.5%) in the set time frame was achieved by the reasoning over the fragment. The high description depth (2.8) and length (5.5) also reveals that it is possible to construct complex class descriptions with the information contained in the fragment. Since the extraction method is more syntactical than semantical in nature, it is likely to scale well for larger knowledge bases in terms of extraction time (as also shown in the previous experiment.)

Related Work

Related work can essentially be divided in ABox contraction and summary techniques on the one hand and learning in Description Logics and OWL on the other hand. Regarding the first area, Fokoue, Kershenbaum, Ma, Schonberg, och Srinivas (2006) for example present an approach how to compute a possibly much smaller summary of an ABox obeying equivalent reasoning properties. Such ap-

proaches are suitable for clean and homogeneous ontologies with small TBoxes and large ABoxes, while our approach is targeted at impure, heterogeneous multi-domain ontologies with both components TBoxes and ABoxes being large. Another application, that is concerned with reasoning on large ABoxes is instanceStore(Horrocks, Li, Turi, & Bechhofer, 2004), which as of now only works on role-free knowledge bases. A project aiming to enable massive distributed incomplete reasoning is LarKC(Fensel m. fl., 2008), which has started recently.

Another related approach is described in Seidenberg och Rector (2006), where fragments of the GALEN ontology are extracted to enable efficient reasoning. The major difference compared to our approach is that we focused on providing a fragment extraction algorithm suitable for learning class descriptions. We start from instances instead of classes and do not need to extract subclasses of obtained classes. Our approach was implemented with support for SPARQL and Linked Data for querying knowledge bases. Furthermore, we do not require the OWL ontology to be normalized and can handle complex class descriptions as fillers of property restrictions. Similarities between both approaches is the idea of a (recursion) depth limit and the extraction of class and property hierarchies.

Regarding learning in Description Logics, the authors of Badea och Nienhuys-Cheng (2000), for example, design a refinement operator for $\mathcal{ALER}$ to obtain a top-down learning algorithm for this language. Other approaches to concept learning were presented in Iannone, Palmisano, och Fanizzi (2007), where concept descriptions are learned based on approximated MSC's (most specific concepts) of the starting instances, which are then merged or refined. A problem of this approach compared to our work is that the proposed algorithm tends to produce very long concept descriptions, which, although they achieve accurate results, can most of the time not be comprehended easily by humans any more, as it is the case in our examples.

Conclusions and Future Work

The focus of our work was to increase the scalability of OWL learning algorithms through intelligent pre-processing. We successfully showed how machine learning techniques can be applied to very large knowledge bases. The creation of background knowledge is a tedious and time consuming process and we proposed a solution to ease this burden. By shifting the necessity of 'inventing' new concept descriptions to the simplicity of selecting instances, we open the field of ontology creation to a broader audience, which might add further momentum to the Semantic Web. We presented methods, which can in the future provide semi-automatic tool support for the enrichment of background knowledge and also add a new dimension to navigation and semantic search. The given examples allow a first glance at the usefulness of the presented algorithms and what sort of results can be achieved.

We plan to establish a DBpedia Navigator Web interface, which will allow users to navigate through sets of instances based on learned class descriptions. Although being developed for DBpedia, the navigator will also be usable with other SPARQL endpoints and make heavy use of the methods presented herein.

During our experiments, we experienced technical and engineering hurdles such as non-standard behavior, lack of interlinking and semantically rich structures or simply inaccessibility. Hence, working with very large knowledge bases is still challenging from both - engineering and research - perspectives.

References

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., & Ives, Z. G. (2007). DBpedia: A nucleus for a web of open data. I *ISWC/ASWC (2007)* (s. 722-735). Springer.

Auer, S., & Lehmann, J. (2007). What have Innsbruck and Leipzig in common? extracting semantics from wiki content. I *Proceedings of the eswc (2007)* (ss. 503–517). Springer.

Baader, F., Ganter, B., Sertkaya, B., & Sattler, U. (2007). Completing Description Logic knowledge bases using Formal Concept Analysis. I M. M. Veloso (red.), *Ijcai* (s. 230-235).

Badea, L., & Nienhuys-Cheng, S.-H. (2000). A refinement operator for description logics. *LNCS (1866)*, 40–58.

Battista, A. D. L., Villanueva-Rosales, N., Palenychka, M., & Dumontier, M. (2007). SMART: A web-based, ontology-driven, semantic web query answering application. I *Semantic web challenge at the iswc 2007*.

Bechhofer, S., Harmelen, F. van, Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., m. fl. (2004, Feb). *OWL web ontology language reference.* W3C Recommendation. (Available at: http://www.w3.org/TR/owl-ref, last access on Dez 2008)

Bechhofer, S., Mller, R., & Crowther, P. (2003). The DIG description logic interface. I D. Calvanese, G. D. Giacomo, & E. Franconi (red:er), *Description logics* (vol. 81). CEUR-WS.org.

Bechhofer, S., & Volz, R. (2004). Patching syntax in OWL ontologies. I S. A. McIlraith, D. Plexousakis, & F. van Harmelen (red:er), *International semantic web conference* (vol. 3298, s. 668-682). Springer.

Berners-Lee, T. (2006). *Linked Data.* (Retrieved August 15, 2008, from *http://www.w3.org/DesignIssues/LinkedData.html*)

Bizer, C., Cyganiak, R., & Heath, T. (2007). *How to publish Linked Data on the web.* (Retrieved August 15, 2008, from *http://sites.wiwiss.fu-berlin.de/suhl/bizer/pub/LinkedDataTutorial/*)

Boisen, S. (2006). *Semantic Bible: New Testament Names: a semantic knowledge base.* (Retrieved August 15, 2008, from *http://www.semanticbible.com/ntn/ntn-overview.html*)

Brachman, R. J. (1978). *A structural paradigm for representing knowledge* (teknisk rapport nr. BBN Report 3605). Cambridge, MA: Bolt, Beraneck and Newman, Inc.

Clark, K. G., Feigenbaum, L., & Torres, E. (2008, January 15). *SPARQL Protocol for RDF* (W3C Recommendation). W3C.

Cycorp. (2008). *Opencyc is the open source version of the cyc technology.* (Retrieved August 15, 2008, from *http://opencyc.org/*)

ESWWiki. (2008). *Currently Alive SPARQL Endpoints.* (Retrieved August 15, 2008, from *http://esw.w3.org/topic/SparqlEndpoints*)

Fensel, D., Harmelen, F. van, Andersson, B., Brennan, P., Cunningham, H., Valle, E. D., m. fl. (2008, 8). *Towards LarKC: a Platform for Web-scale Reasoning.* IEEE Computer Society Press Los Alamitos, CA, USA. (to appear)

Fokoue, A., Kershenbaum, A., Ma, L., Schonberg, E., & Srinivas, K. (2006). The summary ABox: Cutting ontologies down to size. I *Iswc* (s. 343-356).

Horrocks, I., Kutz, O., & Sattler, U. (2006). The even more irresistible SROIQ. I P. Doherty, J. Mylopoulos, & C. A. Welty (red:er), *Proceedings, tenth international conference on principles of knowledge representation and reasoning, lake district of the united kingdom, june 2-5, 2006* (ss. 57–67). AAAI Press.

Horrocks, I., Li, L., Turi, D., & Bechhofer, S. (2004). The instance store: Dl reasoning with large numbers of individuals. I V. Haarslev & R. Möller (red:er), *Description logics* (vol. 104). CEUR-WS.org.

Horrocks, I., & Patel-Schneider, P. F.(2003). Reducing OWL entailment to description logic satisfiability. I D. Fensel, K. Sycara, & J. Mylopoulos (red:er), *Proc. of the 2nd international semantic web conference (iswc 2003)* (ss. 17–29). Springer.

Horrocks, I., Patel-Schneider, P. F., & Harmelen, F. van. (2003). From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, *1*(1), 7–26.

Iannone, L., Palmisano, I., & Fanizzi, N. (2007). An algorithm based on counterfactuals for concept learning in the semantic web. *Appl. Intell.*, *26*(2), 139-159.

Lehmann, J. (2007). *DL-Learner project page.* (open source and available at *http://dl-learner.org*)

Lehmann, J., & Hitzler, P. (2007a). Foundations of refinement operators for description logics. I *17th int. conf. on inductive logic programming (ilp).*

Lehmann, J., & Hitzler, P. (2007b). A refinement operator based learning algorithm for the ALC description logic. I *17th int. conf. on inductive logic programming (ilp).*

Lenat, D. (1995). CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, *38*(11), 33–38.

Miles, A., & Brickley, D. (2005, November). *Skos core vocabulary specification.* World Wide Web Consortium, Working Draft WD-swbp-skos-core-spec-20051102. (Retrieved August 15, 2008, from *http://www.w3.org/TR/2005/WD-swbp-skos-core-spec-20051102/*)

Nienhuys-Cheng, S.-H., & Wolf, R. de (red:er). (1997). *Foundations of inductive logic programming*. Springer.

Seidenberg, J., & Rector, A. L.(2006). Web ontology segmentation: analysis, classification and use. I L. Carr, D. D. Roure, A. Iyengar, C. A. Goble, & M. Dahlin (red:er), *Www* (ss. 13–22). ACM.

Stickler, P. (2004). *CBD - concise bounded description.* (Retrieved August 15, 2008, from *http://www.w3.org/Submission/CBD/*)

Suchanek, F. M., Kasneci, G., & Weikum, G. (2007). Yago: a core of semantic knowledge. I *Www '07: Proceedings of the 16th international conference on world wide web* (ss. 697–706). New York, NY, USA: ACM Press.

Swartz, A. (2002). Musicbrainz: A semantic web service. *IEEE Intelligent Systems*, *17*(1), 76-77.

Tauberer, J. (2008). *Govtrack.us - a civic project to track congress.* (Retrieved August 15, 2008, from *http://www.govtrack.us*)