LOD2 Deliverable D3.3.1: Release of Knowledge Base Enrichment Algorithms

Lorenz Bühmann, Jens Lehmann

Abstract: The prototype deliverable consists of a DL-Learner software release and an accompanying deliverable report. The software is open source and can be downloaded at http://dl-learner.org. It implement several knowledge base enrichment algorithms developed or extended within LOD2. The deliverable describes those algorithms in more detail. It first gives an overview of enrichment in context of Linked Data and OWL. Afterwards, it presents the algorithms and a brief evaluation on DBpedia. Finally, the software itself is described and important pointers are given.

Collaborative Project

LOD2 - Creating Knowledge out of Interlinked Data

Project Number: 257943 Start Date of Project: 01/09/2010 Duration: 48 months

Deliverable 3.3.1
Release of Knowledge Base Enrichment Algorithms

Dissemination Level	Public
Due Date of Deliverable	Month 12, 31/08/2011
Actual Submission Date	31/08/2011
Work Package	WP3, Knowledge Base Creation, Enrichment and Repair
Task	Task T3.3
Type	Report
Approval Status	Approved
Version	1.0
Number of Pages	29
Filename	deliverable-3.3.1.pdf

Abstract: This is a deliverably accompanying a software release on knowledge base enrichment algorithms.

Project funded by the European Commission within the Seventh Framework Programme (2007 – 2013)

History

Version	Date	Reason	Revised by
0.1	2011-07-25	Initial version	Jens Lehmann
0.2	2010-08-02	Class Enrichment	Jens Lehmann
0.3	2010-08-12	Initial Property Enrichment	Lorenz Bühmann
0.4	2010-08-18	Initial Evaluation	Lorenz Bühmann
0.5	2010-08-23	Introduction	Jens Lehmann
0.6	2010-08-26	Enrichment	Jens Lehmann
0.7	2010-08-26	Evaluation	Lorenz Bühmann
0.8	2010-08-27	Property Enrichment	Jens Lehmann
0.9	2010-08-28	Conclusion	Jens Lehmann
1.0	2010-08-30	Final version	Jens Lehmann

Author list

Organisation	Name	Contact Information
ULEI	Lorenz Bühmann	buehmann@informatik.uni-leipzig.de
ULEI	Jens Lehmann	lehmann@informatik.uni-leipzig.de

Executive summary

The prototype deliverable consists of a DL-Learner[1] software release and an accompanying deliverable report. The software is open source and can be downloaded at `http://sf.net/projects/dl-learner/files/DL-Learner/`. It implement several knowledge base enrichment algorithms developed or extended within LOD2. The deliverable describes those algorithms in more detail. It first gives an overview of enrichment in context of Linked Data and OWL. Afterwards, it presents the algorithms and a brief evaluation on DBpedia. Finally, the software itself is described and important pointers are given.

[1] `http://dl-learner.org`

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

Note: If you are already familiar with schema enrichment and just want to test the prototype, you can go straight to Chapter 6.

The Semantic Web has recently seen a rise in the availability and usage of knowledge bases, as can be observed within the Linking Open Data Initiative, the TONES and Protégé ontology repositories, or the Watson search engine. Despite this growth, there is still a lack of knowledge bases that consist of sophisticated schema information and instance data adhering to this schema. Several knowledge bases, e.g. in the life sciences, only consist of schema information, while others are, to a large extent, a collection of facts without a clear structure, e.g. information extracted from data bases or texts. The combination of sophisticated schema and instance data would allow powerful reasoning, consistency checking, and improved querying possibilities. Schema enrichment, as described in this deliverable, allows to create sophisticated schema base based on existing data (sometimes referred to as "grass roots" approach or "after the fact" schema creation).

Example 1.1
As an example, consider a knowledge base containing a class `Capital` *and instances of this class, e.g. London, Paris, Washington, Canberra etc. A machine learning algorithm could, then, suggest that the class* `Capital` *may be equivalent to one of the following OWL class expressions in Manchester OWL syntax*[1]*:*

```
City and isCapitalOf at least one GeopoliticalRegion
City and isCapitalOf at least one Country
```

[1]For details on Manchester OWL syntax (e.g. used in Protégé, OntoWiki) see `http://www.w3.org/TR/owl2-manchester-syntax/`.

Both suggestions could be plausible: The first one is more general and includes cities that are capitals of states, whereas the latter one is stricter and limits the instances to capitals of countries. A knowledge engineer can decide which one is more appropriate, i.e. a semi-automatic approach is used, and the machine learning algorithm should guide her by pointing out which one fits the existing instances better. Assuming the knowledge engineer decides for the latter, an algorithm can show her whether there are instances of the class `Capital` which are neither instances of `City` nor related via the property `isCapitalOf` to an instance of `Country`.[2] The knowledge engineer can then continue to look at those instances and assign them to a different class as well as provide more complete information; thus improving the quality of the knowledge base. After adding the definition of `Capital`, an OWL reasoner can compute further instances of the class which have not been explicitly assigned before.

We implemented our enrichment methods in the DL-Learner framework [18] based on our previous efforts in [13, 28, 17, 27, 26, 19, 24, 23, 22]. The latest release, which implements the algorithms described in this deliverable is available at `www.w3.org/TR/owl2-manchester-syntax/`. Whereas previous releases of DL-Learner focused only on learning equivalence and subclass relationships, this prototype can perform schema enrichment for most OWL2 axioms. This is described in more detail in Chapters 3 and 4. The usage of the prototype itself is described in Chapter 6. At this stage, we provide a command line interface for performing schema enrichment. Later in the LOD2 project, the algorithms will be integrated in the ORE user interface to make them accessible for a more general audience.

Overall, the presented software prototype implements the following features:

- support for suggesting the following axioms to enrich a knowledge base:
  - equivalent classes (via supervised learning)
  - superclasses (via supervised learning)
  - disjoint classes
  - transitivity of properties
  - property hierarchies
  - symmetry of properties
  - functional properties

[2] This is not an inconsistency under the standard OWL open world assumption, but rather a hint towards a potential modelling error.

- scalable algorithms for large RDF knowledge bases
- support for several available algorithms
- easily configurable algorithms

In the remainder of the introduction, we briefly described the term schema enrichment, give an overview of existing approaches and present the outline of the deliverable.

1.1 Knowledge Base Enrichment Overview

The term *enrichment* in this deliverable refers to the (semi-)automatic extension of a knowledge base schema. It describes the process of increasing the expressiveness and semantic richness of a knowledge base. Usually, this is achieved by adding or refining terminological axioms.

Enrichment methods can typically be applied in a *grass-roots* approach to knowledge base creation. In such an approach, the whole ontological structure is not created upfront, but evolves with the data in a knowledge base. Ideally, this enables a more agile development of knowledge bases. In particular, in the context of the Web of Linked Data such an approach appears to be an interesting alternative to more traditional ontology engineering methods. Amongst others, Tim Berners-Lee advocates to get "raw data now"[3] and worry about the more complex issues later.

Knowledge base enrichment can be seen as a sub-discipline of ontology learning. Ontology learning is more general in that it can rely on external sources, e.g. written text, to create an ontology. The term knowledge base enrichment is typically used when already existing data in the knowledge base itself is analysed to improve its schema.

Enrichment methods span several research areas like knowledge representation and reasoning, machine learning, statistics, natural language processing, formal concept analysis and game playing. Ontology enrichment usually involves applying heuristics or machine learning techniques to find axioms, which can be added to an existing ontology. Naturally, different techniques have been applied depending on the specific type of axiom.

One of the most complex tasks in ontology enrichment is to find *definitions* of classes. This is strongly related to Inductive Logic Programming (ILP) [31] and more specifically supervised learning in description logics. Research in those fields has many applications apart from being applied to enrich ontologies. For instance, it is used in the life sciences to detect whether

[3] `http://www.ted.com/talks/tim_berners_lee_on_the_next_web.html`

drugs are likely to be efficient for particular diseases. Work on learning in description logics goes back to e.g. [9, 10], which used so-called *least common subsumers* to solve the learning problem (a modified variant of the problem defined in this article). Later, [6] invented a refinement operator for $\mathcal{ALER}$ and proposed to solve the problem by using a top-down approach. [11, 15, 16] combine both techniques and implement them in the YINYANG tool. However, those algorithms tend to produce very long and hard-to-understand class expressions. The algorithms implemented in DL-Learner [26, 27, 17, 28] overcome this problem and investigate the learning problem and the use of top down refinement in detail. DL-FOIL [12] is a similar approach, which is based on a mixture of upward and downward refinement of class expressions. They use alternative measures in their evaluation, which take the open world assumption into account, which was not done in ILP previously. Most recently, CELOE [20] implements appropriate heuristics and adaptations for learning definitions in ontologies. The focus in this deliverable is efficiency and practical application of learning methods. CELOE was partially developed within the LOD2 project and is one of the foundations of our schema enrichment prototype (see Chapter 3).

A different approach to learning the definition of a named class is to compute the so called *most specific concept* (msc) for all instances of the class. The most specific concept of an individual is the most specific class expression, such that the individual is instance of the expression. One can then compute the *least common subsumer* (lcs) [5] of those expressions to obtain a description of the named class. However, in expressive description logics, an msc does not need to exist and the lcs is simply the disjunction of all expressions. For light-weight logics, such as $\mathcal{EL}$, the approach appears to be promising.

Other approaches, e.g. [29] focus on learning in hybrid knowledge bases combining ontologies and *rules*. Ontology evolution [30] has been discussed in this context. Usually, hybrid approaches are a generalisation of concept learning methods, which enable powerful rules at the cost of efficiency (because of the larger search space). Similar as in knowledge representation, the tradeoff between expressiveness of the target language and efficiency of learning algorithms is a critical choice in symbolic machine learning.

Another enrichment task is *knowlege base completion*. The goal of such a task is to make the knowledge base complete in a particular well-defined sense. For instance, a goal could be to ensure that all subclass relationships between named classes can be inferred. The line of work starting in [33] and further pursued in e.g. [4] investigates the use of *formal concept analysis* for completing knowledge bases. It is promising, although it may not be able to handle noise as well as a machine learning technique. A Protégé plugin [34]

is available. [37] proposes to improve knowledge bases through relational exploration and implemented it in the *RELExO framework*[4]. It focuses on simple relationships and the knowledge engineer is asked a series of questions. The knowledge engineer either must positively answer the question or provide a counterexample.

[38] focuses on learning *disjointness* between classes in an ontology to allow for more powerful reasoning and consistency checking. To achieve this, it can use the ontology itself, but also texts, e.g. Wikipedia articles corresponding to a concept. The article includes an extensive study, which shows that proper modelling disjointness is actually a difficult task, which can be simplified via this ontology enrichment method.

Another type of ontology enrichment is schema mapping. This task has been widely studied and will not be discussed in depth here. Instead, we refer to [8] for a survey on ontology mapping. Schema mapping is not integrated in the presented prototype.

There are further more light-weight ontology enrichment methods. For instance, *taxonomies* can be learned from simple tag structures via heuristics. Similarly, "properties of properties" can be derived via simple statistical analysis. This includes the detection whether a particular property might be symmetric, function, reflexive, inverse functional etc. Similarly, domains and ranges of properties can be determined from existing data. Enriching the schema with domain and range axioms allows to find cases, where properties are misused via OWL reasoning. We created and implemented several such light weight approaches in the deliverable prototype.

1.2 Outline

The deliverable is structured as follows: We introduce necessary notions for understanding the basics of the used machine learning algorithms in Chapter 2. We then continue with specific schema enrichment approaches for complex classes in Chapter 3 and for all other axioms in Chapter 4. To be able to separate the process of generating enrichments from the process of manual supervision by a knowledge engineer, we need to store the suggestions. We do this via an ontology, which is described in Chapter 5. The usage of DL-Learner release for enrichment is described in Chapter 6. It allows the reader to apply the algorithms on arbitrary SPARQL endpoints. We then continue by giving some preliminary evaluation results for applying the algorithms on DBpedia in Chapter 7. Finally, we conclude and give

[4] `http://code.google.com/p/relexo/`

Type/Aim	**References**
Taxonomies	[39]
Definitions	often done via ILP approaches such as [26, 27, 28, 20, 12, 11, 15, 16, 6], genetic approaches [17] have also been used
Super Class Axioms	[20]
Rules in Ontologies	[29, 30]
Disjointness	[38]
Properties of properties	usually via heuristics
Alignment	challenges: [35], recent survey: [8]
Completion	formal concept analysis and relational exploration [4] [37, 34]

Table 1.1: Work in ontology enrichment grouped by type or aim of learned structures.

important pointers to the DL-Learner software release, which is described in this deliverable.

Chapter 2

Preliminaries

Herein, we describe foundations of the machine learning approaches in this deliverable. We assume familiarity with OWL, description logics and SPARQL.

It should be noted that those preliminaries are mainly required to understand the CELOE algorithm, which we will describe in the sequel. They are, however, not necessary to use the DL-Learner enrichment prototype.

2.1 Learning Problem

The process of learning in logics, i.e. trying to find high-level explanations for given data, is also called *inductive reasoning* as opposed to *inference* or *deductive reasoning*. The main difference is that in deductive reasoning it is formally shown whether a statement follows from a knowledge base, whereas in inductive learning new statements are invented. Learning problems, which are similar to the one we will analyse, have been investigated in *Inductive Logic Programming* [31] and, in fact, the method presented here can be used to solve a variety of machine learning tasks apart from ontology enrichment.

In the ontology learning problem we consider, we want to learn a formal description of a class A, which has (inferred or asserted) instances in the considered ontology. In the case that A is already described by a class expression C via axioms of the form $A \sqsubseteq C$ or $A \equiv C$, those can be either refined, i.e. specialised/generalised, or relearned from scratch by the learning algorithm. To define the class learning problem, we need the notion of a *retrieval* reasoner operation $R_{\mathcal{K}}(C)$. $R_{\mathcal{K}}(C)$ returns the set of all instances of C in a knowledge base $\mathcal{K}$. If $\mathcal{K}$ is clear from the context, the subscript can be ommitted.

Definition 2.1 (class learning problem)
Let an existing named class A in a knowledge base $\mathcal{K}$ be given. The *class*

learning problem is to find an expression C such that $R_{\mathcal{K}}(C) = R_{\mathcal{K}}(A)$.

Clearly, the learned expression C is a description of (the instances of) A. Such an expression is a candidate for adding an axiom of the form $A \equiv C$ or $A \sqsubseteq C$ to the knowledge base $\mathcal{K}$. If a solution of the learning problem exists, then the used base learning algorithm (as presented in the following subsection) is complete, i.e. guaranteed to find a correct solution if one exists in the target language and there are no time and memory constraints (see [27, 28] for the proof). In most cases, we will not find a solution to the learning problem, but rather an approximation. This is natural, since a knowledge base may contain false class assignments or some objects in the knowledge base are described at different levels of detail. For instance, in Example 1.1, the city "Apia" might be typed as "Capital" in a knowledge base, but not related to the country "Samoa". However, if most of the other cities are related to countries via a role `isCapitalOf`, then the learning algorithm may still suggest `City and isCapitalOf at least one Country` since this describes the majority of capitals in the knowledge base well. If the knowledge engineer agrees with such a definition, then a tool can assist him in completing missing information about some capitals.

By Occam's razor [7] simple solutions of the learning problem are to be preferred over more complex ones, because they are more readable. This is even more important in the ontology engineering context, where it is essential to suggest simple expressions to the knowledge engineer. We measure simplicity as the *length* of an expression, which is defined in a straightforward way, namely as the sum of the numbers of concept, role, quantifier, and connective symbols occurring in the expression. The algorithm is biased towards shorter expressions. Also note that, for simplicity the definition of the learning problem itself does enforce coverage, but not prediction, i.e. correct classification of objects which are added to the knowledge base in the future. Concepts with high coverage and poor prediction are said to *overfit* the data. However, due to the strong bias towards short expressions this problem occurs empirically rare in description logics [28].

2.2 Refinement Operators

The solution of the learning problem stated above can be cast as a search for a correct concept definition in an ordered space $(\Sigma, \preceq)$. In such a setting, one can define suitable operators to traverse the search space. Refinement operators can be formally defined as:

Definition 2.2 (refinement operator)
Given a quasi-ordered[1] search space $(\Sigma, \preceq)$

- a *downward refinement operator* is a mapping $\rho : \Sigma \to 2^{\Sigma}$ such that
$$\forall \alpha \in \Sigma \quad \rho(\alpha) \subseteq \{\beta \in \Sigma \mid \beta \preceq \alpha\}$$
- an *upward refinement operator* is a mapping $\delta : \Sigma \to 2^{\Sigma}$ such that
$$\forall \alpha \in \Sigma \quad \delta(\alpha) \subseteq \{\beta \in \Sigma \mid \alpha \preceq \beta\}$$

□

Definition 2.3 (properties of DL refinement operators)
A refinement operator ρ is

- *(locally) finite* iff $\rho(C)$ is finite for all concepts C.
- *redundant* iff there exists a refinement chain from a concept C to a concept D, which does not go through some concept E and a refinement chain from C to a concept equal to D, which does go through E.
- *proper* iff for all concepts C and D, $D \in \rho(C)$ implies $C \not\equiv D$.

A downward refinement operator ρ is called

- *complete* iff for all concepts C, D with $C \sqsubset D$ we can reach a concept E with $E \equiv C$ from D by ρ.
- *weakly complete* iff for all concepts $C \sqsubset \top$ we can reach a concept E with $E \equiv C$ from $\top$ by ρ.

The corresponding notions for upward refinement operators are defined dually. □

In the following, we will consider a space of concept definitions ordered by the subsumption relationship $\sqsubseteq$ which induces a quasi-order on the space of all the possible concept descriptions [6, 11]. In particular, given the space of concept definitions in the reference DL language, say $(\mathcal{L}, \sqsubseteq)$, ordered by subsumption, there is an infinite number of generalizations and specializations. Usually one tries to devise operators that can traverse efficiently throughout the space in pursuit of one of the correct definitions (w.r.t. the examples that have been provided).

[1] A quasi-ordering is a reflexive and transitive relation.

Definition 2.4 (refinement chain)
A *refinement chain* of an $\mathcal{L}$ refinement operator ρ of length n from a concept C to a concept D is a finite sequence $C_0, C_1, \ldots, C_n$ of concepts, such that $C = C_0, C_1 \in \rho(C_0), C_2 \in \rho(C_1), \ldots, C_n \in \rho(C_{n-1}), D = C_n$. This refinement chain *goes through* E iff there is an i $(1 \leq i \leq n)$ such that $E = C_i$. We say that D can be reached from C by ρ if there exists a refinement chain from C to D. $\rho^*(C)$ denotes the set of all concepts, which can be reached from C by ρ. $\rho^m(C)$ denotes the set of all concepts, which can be reached from C by a refinement chain of ρ of length m. □

Instead of $D \in \rho(C)$, we will often write $C \rightsquigarrow_\rho D$. If the used operator is clear from the context, it is usually omitted, i.e. we write $C \rightsquigarrow D$.

Chapter 3

Complex Class Expression Enrichment

We will first describe class enrichment, i.e. learning semantic descriptions of OWL classes. As previously mentioned, we employ the CELOE algorithm for this, which has been partially developed within LOD2 by the authors. Full details can be found in [20], but we also describe it here to make the deliverable self-contained.

3.1 CELOE Refinement Operator

Designing a refinement operator ρ involves the need to make decisions on which properties are most useful in practice regarding the underlying learning algorithm. Considering the properties *completeness*, *weak completeness*, *properness*, *finiteness*, and *non-redundancy* an extensive analysis in [26] has shown that the most feasible property combination for our setting is {*weakly complete, complete, proper*}, which we will justify briefly. Only for less expressive description logics like $\mathcal{EL}$, ideal, i.e. complete, proper and final, operators exist [25]. (Weak) Completeness is considered a very important property, since an incomplete operator may fail to converge at all and thus may not return a solution even if one exists. Reasonable, weakly complete operators are often complete. Consider, for example, the situation where a weakly complete operator ρ allows to refine a concept C to $C \sqcap D$ with some $D \in \rho(\top)$. Then it turns out that this operator is already complete (which is not hard to show).

Concerning finiteness, having an infinite operator is less critical from a practical perspective since this issue can be handled algorithmically. So it is preferable not imposing finiteness, which allows to develop a proper operator

This is the consequence of a result in [28] which shows that finiteness and properness cannot be combined while preserving completeness. As for non-redundancy, this appears to be very difficult to achieve for more complex operators. Consider, for example, the concept $A_1 \sqcap A_2$ which can be reached from $\top$ via the chain $\top \rightsquigarrow A_1 \rightsquigarrow A_1 \sqcap A_2$, For non-redundancy, the operator would need to make sure that this concept cannot be reached via the chain $\top \rightsquigarrow A_2 \rightsquigarrow A_2 \sqcap A_1$. While there are methods to handle this in such simple cases via normal forms, it becomes more complex for arbitrarily deeply nested structures, where even applying the same replacement leads to redundancy. In the following example, A_1 is replaced by $A_1 \sqcap A_2$ twice in different order in each chain:

$$\begin{aligned}\top \rightsquigarrow &\forall r_1.A_1 \sqcup \forall r_2.A_1 \rightsquigarrow \forall r_1.A_1 \sqcup \forall r_2.(A_1 \sqcap A_2)\\ &\rightsquigarrow \forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.(A_1 \sqcap A_2)\\ \top \rightsquigarrow &\forall r_1.A_1 \sqcup \forall r_2.A_1 \rightsquigarrow \forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.A_1\\ &\rightsquigarrow \forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.(A_1 \sqcap A_2)\end{aligned}$$

To avoid this, an operator would need to regulate when A_1 can be replaced by $A_1 \sqcap A_2$, which appears not to be achievable by syntactic replacement rules. Alternatively, a computationally inexpensive redundancy check can be used, which seems to be sufficiently useful in practice.

We now define the refinement operator ρ: For each $A \in N_C$, we define (sh stands for subsumption hierarchy):

$$sh_{\downarrow}(A) = \{A' \in N_C \mid A' \sqsubset A, \text{ there is no } A'' \in N_C \text{ with } A' \sqsubset_{\mathcal{T}} A'' \sqsubset_{\mathcal{T}} A\}$$

$sh_{\downarrow}(\top)$ is defined analogously for $\top$ instead of A. $sh_{\uparrow}(A)$ is defined analogously for going upward in the subsumption hierarchy. We do the same for roles, i.e. :

$$sh_{\downarrow}(r) = \{r' \mid r' \in N_R, r' \sqsubset r, \text{ there is no } r'' \in N_R \text{ with } r' \sqsubset_{\mathcal{T}} r'' \sqsubset_{\mathcal{T}} r\}$$

domain(r) denotes the domain of a role r and *range*(r) the range of a role r. A range axiom links a role to a concept. It asserts that the role fillers must be instances of a given concept. Domain axioms restrict the first argument of role assertions to a concept. We define:

$$\begin{aligned}ad(r) = \quad &\text{an } A \text{ with } A \in \{\top\} \cup N_C \text{ and } \mathit{domain}(r) \sqsubseteq A\\ &\text{and there does not exist an } A' \text{ with } \mathit{domain}(r) \sqsubseteq A' \sqsubset A\end{aligned}$$

$ar(r)$ is defined analogously using *range* instead of *domain*. ad stands for atomic domain and ar stands for atomic range. We assign exactly one atomic

concept as domain/range of a role. Since using atomic concepts as domain and range is very common, *domain* and *ad* as well as *range* and *ar* will usually coincide. The set app_B of applicable properties with respect to an atomic concept B is defined as:

$$app_B = \{r | r \in N_R, ad(r) = A, A \sqcap B \not\equiv \bot\}$$

To give an example, for the concept `Person`, we have that the role `hasChild` with ad(`hasChild`) = `Person` is applicable, but the role `hasAtom` with ad(`hasAtom`) = `ChemicalCompound` is not applicable (assuming `Person` and `ChemicalCompound` are disjoint). We will use this to restrict the search space by ruling out unsatisfiable concepts. The index B describes the context in which the operator is applied, e.g. $\top \rightsquigarrow$ `Person` is a refinement step of ρ. However, $\exists$`hasAtom`.$\top \rightsquigarrow \exists$`hasAtom`.`Person` is not a refinement step of ρ assuming ar(`hasAtom`) and `Person` are disjoint. The set of most general applicable roles mgr_B with respect to a concept B is defined as:

$$mgr_B = \{r \mid r \in app_B, \text{ there is no } r' \text{ with } r \sqsubset r', r' \in app_B\}$$

M_B with $B \in \{\top\} \cup N_C$ is defined as the union of the following sets:

- $\{A \mid A \in N_C, A \sqcap B \not\equiv \bot, A \sqcap B \not\equiv B, \text{ there is no } A' \in N_C \text{ with } A \sqsubset A'\}$
- $\{\neg A \mid A \in N_C, \neg A \sqcap B \not\equiv \bot, \neg A \sqcap B \not\equiv B, \text{ there is no } A' \in N_C \text{ with } A' \sqsubset A\}$
- $\{\exists r.\top \mid r \in mgr_B\}$
- $\{\forall r.\top \mid r \in mgr_B\}$

The operator ρ is defined in Figure 3.1. Note that ρ delegates to an operator ρ_B with $B = \top$ initially. B is set to the atomic range of roles contained in the input concept when the operator recursively traverses the structure of the concept. The index B in the operator (and the set M above) is used to rule out concepts which are disjoint with B.

Example 3.1 (ρ refinements)
Since the operator is not easy to understand at first glance, we provide some examples. Let the following knowledge base be given:

$\mathcal{K} = \{$*Man* $\sqsubset$ *Person*; *Woman* $\sqsubset$ *Person*; *SUV* $\sqsubset$ *Car*; *Limo* $\sqsubset$ *Car*;
Person $\sqcap$ *Car* $\equiv \bot$; *domain*(*hasOwner*) = *Car*; *range*(*hasOwner*) = *Person*$\}$

$$\rho(C) = \begin{cases} \{\bot\} \cup \rho_\top(C) & \text{if } C = \top \\ \rho_\top(C) & \text{otherwise} \end{cases}$$

$$\rho_B(C) = \begin{cases} \emptyset & \text{if } C = \bot \\ \{C_1 \sqcup \cdots \sqcup C_n \mid C_i \in M_B \ (1 \leq i \leq n)\} & \text{if } C = \top \\ \{A' \mid A' \in sh_\downarrow(A)\} & \text{if } C = A \ (A \in N_C) \\ \quad \cup \{A \sqcap D \mid D \in \rho_B(\top)\} & \\ \{\neg A' \mid A' \in sh_\uparrow(A)\} & \text{if } C = \neg A \ (A \in N_C) \\ \quad \cup \{\neg A \sqcap D \mid D \in \rho_B(\top)\} & \\ \{\exists r.E \mid A = ar(r), E \in \rho_A(D)\} & \text{if } C = \exists r.D \\ \quad \cup \{\exists r.D \sqcap E \mid E \in \rho_B(\top)\} & \\ \quad \cup \{\exists s.D \mid s \in sh_\downarrow(r)\} & \\ \{\forall r.E \mid A = ar(r), E \in \rho_A(D)\} & \text{if } C = \forall r.D \\ \quad \cup \{\forall r.D \sqcap E \mid E \in \rho_B(\top)\} & \\ \quad \cup \{\forall r.\bot \mid & \\ \qquad D = A \in N_C \text{ and } sh_\downarrow(A) = \emptyset\} & \\ \quad \cup \{\forall s.D \mid s \in sh_\downarrow(r)\} & \\ \{C_1 \sqcap \cdots \sqcap C_{i-1} \sqcap D \sqcap C_{i+1} \sqcap \cdots \sqcap C_n \mid & \text{if } C = C_1 \sqcap \cdots \sqcap C_n \\ \qquad D \in \rho_B(C_i), 1 \leq i \leq n\} & (n \geq 2) \\ \{C_1 \sqcup \cdots \sqcup C_{i-1} \sqcup D \sqcup C_{i+1} \sqcup \cdots \sqcup C_n \mid & \text{if } C = C_1 \sqcup \cdots \sqcup C_n \\ \qquad D \in \rho_B(C_i), 1 \leq i \leq n\} & (n \geq 2) \\ \quad \cup \{(C_1 \sqcup \cdots \sqcup C_n) \sqcap D \mid & \\ \qquad D \in \rho_B(\top)\} & \end{cases}$$

Figure 3.1: Definition of the refinement operator ρ, a simplified version of the operator used in CELOE.

Then the following refinements of $\top$ *exist:*

$$\rho(\top) = \{\mathtt{Car}, \mathtt{Person}, \neg\mathtt{Limo}, \neg\mathtt{SUV}, \neg\mathtt{Woman}, \neg\mathtt{Man}, \\ \exists\mathtt{hasOwner}.\top, \forall\mathtt{hasOwner}.\top, \mathtt{Car} \sqcup \mathtt{Car}, \mathtt{Car} \sqcup \mathtt{Person}, \dots\}$$

This illustrates how the set $M_\top$ *is constructed. Note that refinements like* $\mathtt{Car} \sqcup \mathtt{Car}$ *are incorporated in order to reach e.g.* $\mathtt{SUV} \sqcup \mathtt{Limo}$ *later in a possible refinement chain. The concept* $\mathtt{Car} \sqcap \exists\mathtt{hasOwner}.\mathtt{Person}$ *has the following refinements:*

$$\begin{aligned}\rho(\mathtt{Car} \sqcap \exists\mathtt{hasOwner}.\mathtt{Person}) = \{&\mathtt{Car} \sqcap \exists\mathtt{hasOwner}.\mathtt{Man}, \\ &\mathtt{Car} \sqcap \exists\mathtt{hasOwner}.\mathtt{Woman}, \\ &\mathtt{SUV} \sqcap \exists\mathtt{hasOwner}.\mathtt{Person}, \\ &\mathtt{Limo} \sqcap \exists\mathtt{hasOwner}.\mathtt{Person}, \dots\}\end{aligned}$$

Note the traversal of the subsumption hierarchy, e.g. $\mathtt{Car}$ *is replaced by* $\mathtt{SUV}$*.*

Proposition 3.2 (Downward Refinement of ρ)
ρ is an $\mathcal{ALC}$ downward refinement operator.

A distinguishing feature of ρ compared to other DL refinement operators [6, 11], is that it makes use of the subsumption and role hierarchy, e.g. for concepts $A_2 \sqsubset A_1$, we reach A_2 via $\top \rightsquigarrow A_1 \rightsquigarrow A_2$. This way, we can stop the search if A_1 is already too weak and, thus, make better use of TBox knowledge. The operator also uses domain and range of roles to reduce the search space. This is similar to mode declarations in Aleph, Progol, and other ILP programs. However, in DL knowledge bases and OWL ontologies, domain and range are usually explicitly given, so there is no need to define them manually. Overall, the operator supports more structures than those in [6, 11] and tries to intelligently incorporate background knowledge. In [28] further extensions of the operator are described, which increase its expressivity such that it can handle most OWL class expressions. Note that ρ is infinite. The reason is that the set M_B is infinite and we put no bound on the number of elements in the disjunctions, which are refinements of the top concept. Furthermore, the operator requires reasoner requests for calculating M_B. However, the number of requests is fixed, so – assuming the results of those requests are cached – the reasoner is only needed in an initial phase, i.e. during the first calls to the refinement operator. This means that, apart from this initial phase, the refinement operator performs only syntactic rewriting rules.

3.2 CELOE Algorithm

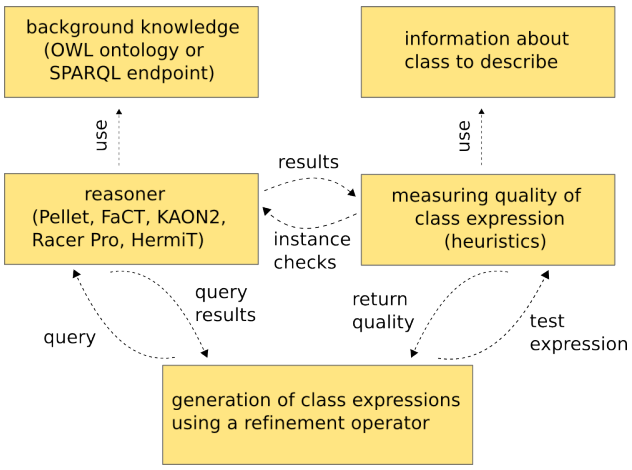

Figure 3.2: Outline of the general learning approach in CELOE: One part of the algorithm is the generation of promising class expressions taking the available background knowledge into account. Another part is a heuristic measure of how close an expression is to being a solution of the learning problem. Figure adapted from [13].

Figure 3.2 gives an overview of our algorithm *CELOE* (standing for "class expression learning for ontology engineering"), which follows the common "generate and test" approach in ILP. Learning is seen as a search process and several class expressions are generated and tested against a background knowledge base. Each of those class expressions is evaluated using a heuristic [20]. A challenging part of a learning algorithm is to decide which expressions to test. Such a decision should take the computed heuristic values and the structure of the background knowledge into account. For CELOE, we use the approach described in [27, 28] as base, where this problem has been analysed, implemented, and evaluated. It is based on an extension of the refinement operator introduced in Section 3.1. We refrain from describing all details of this operator in order to keep the deliverable succinct.

The approach we used is illustrated in Figure 3.3. This means that the first class expression, which will be tested is the most general expression (usually $\top$), which is then mapped to a set of more specific expressions by means of the introduced downward refinement operator. The refinement operator can be applied to the obtained expressions again, thereby spanning

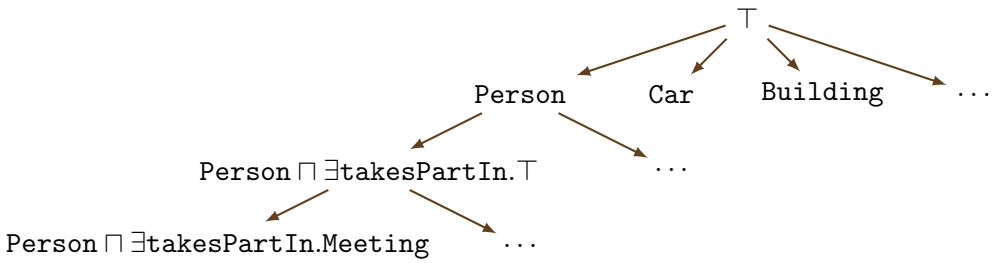

Figure 3.3: Illustration of a search tree in a top down refinement approach.

a *search tree*. The search tree can be pruned when an expression does not cover sufficiently many instances of the class A we want to describe. One example for a path in a search tree spanned up by a downward refinement operator is the following ($\rightsquigarrow$ denotes a refinement step):

$$\top \rightsquigarrow \texttt{Person} \rightsquigarrow \texttt{Person} \sqcap \texttt{takesPartinIn}.\top \rightsquigarrow \texttt{Person} \sqcap \texttt{takesPartIn.Meeting}$$

The heart of such a learning strategy is to define a suitable refinement operator and an appropriate search heuristics for deciding which nodes in the search tree should be expanded. Details about the chosen heuristic can be found in [20]. The goal of the heuristic is to adapt the learning algorithm to the ontology engineering scenario: For example, the algorithm was modified to introduce a strong bias towards short class expressions. This means that the algorithm is less likely to produce long class expressions, but is almost guaranteed to find any suitable short expression. The rationale behind this change is that knowledge engineers can understand short expressions better than more complex ones and it is essential not to miss those. We also introduced improvements to enhance the readability of suggestions: Each suggestion is reduced, i.e. there is a guarantee that they are as succinct as possible. For example, $\exists\texttt{hasLeader}.\top \sqcap \texttt{Capital}$ is reduced to `Capital` if the background knowledge allows to infer that a capital is a city and each city has a leader. This reduction algorithm uses the complete and sound *Pellet* reasoner, i.e. it can take any possible complex relationships into account by performing a series of subsumption checks between class expressions. A caching mechanism is used to store the results of those checks, which allows to perform the reduction very efficiently after a warm-up phase. We also make sure that "redundant" suggestions are omitted. If one suggestion is longer and subsumed by another suggestion and both have the same characteristics, i.e. classify the relevant individuals equally, the more specific suggestion is filtered. This avoids expressions containing irrelevant subexpressions and

ensures that the suggestions are sufficiently diverse. Moreover, stochastic approximations of the heuristic are used in order to increase its scalability. The exact description of those is beyond the scope of the deliverable.

3.3 Learning Equivalent and Super Classes using CELOE

Given an OWL file, CELOE can be invoked by specifying which class should be learned. The instances of this class are then treated as positive examples and other instances as negative examples. The difference between learning equivalence class relationships and subclass relationships are different parameters for the CELOE internal heuristic.

In LOD2, we are mainly concerned with suggesting enrichment axioms on large knowledge bases. Those knowledge bases are usually not available as OWL files, but as SPARQL endpoints. Even if they were available as files, they would be too large to be consumed by standard OWL reasoners. To be able to apply CELOE on SPARQL endpoints, we employ the approach in [13]. It performs several SPARQL queries to obtain a relevant fragment of the considered SPARQL endpoint, which is small enough to be processed by a reasoner, while having only small impact on the performance of learning algorithms like CELOE.

Chapter 4

Enrichment via Other OWL Axioms

In the previous section, we discussed enrichment by learning complex class expressions, which is useful for creating expressive ontologies. However, there is a large variety of other axioms in OWL 2, which are also important and should, therefore, also be taken into consideration in the enrichment process. Such axioms include domain and range of properties, inverse or functional properties, disjoint classes etc. We first describe our general methodology for creating enrichment suggestions for most OWL 2 axiom types and then present details for each axiom type in separate sections.

4.1 General Method

In the axiom learning processes we consider here, the search space is usually much smaller than what we had to take in the previous chapter[1]. Therefore, we considered it reasonable to use more light-weight methods for obtaining enrichment suggestions. The methods usually take an entity (a class or property in our case) as input, generates a set of OWL axioms as output and proceeds in three phases:

1. In the first phase, SPARQL queries are used to obtain general information about the knowledge base, in particular we retrieve axioms, which allow to construct the class hierarchy. Not all of the axiom types

[1]This holds under the assumption that we only consider named classes (not arbitrary complex class expressions) in some axioms, such as disjoint class axioms, domain, range etc.

require such background knowledge, in which case this phase can be ommitted. It can also be configured whether to use an OWL reasoner for inferencing over the schema or just taking explicit knowledge into account.[2] Naturally, the schema only needs to be obtained once and can then be re-used by all algorithms and all entities.

2. The second phase consists of obtaining data via SPARQL, which is relevant for the learning the considered axiom. We will briefly describe this phase for each axiom type in the following sections.

3. In the third phase, the score of axiom candidates is computed and the results returned.

Many of our employed heuristics to suggest axioms are based on counting. For instance, when determining whether a class A is appropriate as domain of a property p, we count the number of triples using p in predicate position and the number of subjects in those triples which are instances of A. The latter value is divided by the first value to obtain a score. We illustrate this using a simple example. Let the following triples be given (Turtle syntax):

```
@prefix dbpedia: <http://dbpedia.org/resource/> .
@prefix dbo:     <http://dbpedia.org/ontology/> .
dbpedia:Luxembourg dbo:currency    dbpedia:Euro ;
                   rdf:type        dbo:Country .
dbpedia:Ecuador    dbo:currency    dbpedia:United_States_dollar ;
                   rdf:type        dbo:Country .
dbpedia:Ifni       dbo:currency    dbpedia:Spanish_peseta ;
                   rdf:type        dbo:PopulatedPlace .
dbo:Country        rdfs:subClassOf dbo:PopulatedPlace.
```

In the above example, we would obtain a score of 66,7% (2 out of 3) for the class `dbo:Country` and 100% (3 out of 3) for the class `dbo:PopulatedPlace`.[3]

A disadvantage of using this straightforward method of obtaining a score is that it does not take the *support* for an axiom in the knowledge base into account. Specifically, there would be no difference between having 100 out of 100 correct observations or 3 out of 3 correct observations.

For this reason, we do not just consider the count, but the average of the 95% confidence interval of the count. This confidence interval can be computed efficiently by using the improved Wald method defined in [1]. Assume we have m observations out of which s were successful, then the approximation of the 95% confidence interval is as follows:

$$\max(0, p' - 1.96 \cdot \sqrt{\frac{p' \cdot (1-p')}{m+4}}) \text{ to } \min(1, p' + 1.96 \cdot \sqrt{\frac{p' \cdot (1-p')}{m+4}})$$

[2]Note the OWL reasoner only loads the schema of the knowledge base and, therefore, this option usually works even in cases with several hundred thousand classes in our experiments, which used the HermiT reasoner.

[3]If the reasoning option is turned off, the score would be 33,3%.

$$\text{with } p' = \frac{s+2}{m+4}$$

This formula is easy to compute and has been shown to be accurate in [1].

In the above case, this would change the score from 66,7% to 57.3% for `dbo:country` and 100% to 69.1% for `dbo:PopulatedPlace`. This indicates that there is not much support for either of those choices in the knowledge base. 100 out of 100 correct observations would score much higher (97.8%).

The actual scores for the DBpedia Live as of August 2011 are 96.8% for `dbo:PopulatedPlace` and 97.2% for `dbo:country`). Note that in this implementation, more general classes in the hierarchy would always score higher. It might be desirable to correct this by slightly penalising very general classes. The drawback of such a penalty could be that the scores would be more difficult to understand for users. We leave the decision on such a penalty and a possible implementation as an area for future work.

4.2 Learning Subclass Axioms

In this section and the following sections, we will just focus on phase 2 of the above described workflow. This phase consists of obtaining the data required for generating enrichment suggestions. Since we mainly expect the data to be available in triple stores in the LOD2 projects, the data acquisition is implemented via SPARQL queries. We will briefly present the necessary SPARQL query (or queries) here.

The first axiom type, we consider, are subclass axioms. Generating suggestions for subclass axioms allows to create a taxonomy from instance data. The data for this will be fetched via the following query:

```
SELECT ?type (COUNT(?ind) AS ?count) WHERE {
  ?ind a ?type.
  {
    SELECT ?ind WHERE {
      ?ind a <$class>.
    } LIMIT $limit OFFSET $offset
  }
} GROUP BY ?type
```

The query assumes a `$class` as input for which we want to learn superclasses. It retrieves all instances of a class and then counts the types for each of those instances. A higher count indicates better candidates for superclasses.[4] Between superclass candidates with the same score, the most specific ones according to the background knowledge should be preferred.

[4] Of course, we filter the input class `$class`. This could be done directly in the SPARQL query, but experimentally this was less efficient. This could be due to triple store optimisers working better on queries without `FILTER` instructions.

Remark: We often use SPARQL 1.1 queries in this deliverable, for instance in the above queries sub-SELECTs were employed. Many triple stores already support (parts of) SPARQL 1.1, which is why we do not consider the usage of such queries as significant drawback. Moreover, in some cases, we obtained order of magnitude improvements when being able to use SPARQL 1.1 syntax compared to a similar implementation adhering to SPARQL 1.0.

4.3 Learning Disjointness

For disjointness, we can use the same query as above:

```
SELECT ?type (COUNT(?ind) AS ?count) WHERE {
  ?ind a ?type.
  {
    SELECT ?ind WHERE {
      ?ind a <$class>.
    } LIMIT $limit OFFSET $offset
  }
} GROUP BY ?type
```

The only difference is that this time, a lower count indicates a candidate for disjointness. In future work, we plan to perform a more fine-grained analysis for the case of disjointness. In particular when considering the case of generating disjointness axioms for a complete knowledge base (not just a single class), it is usually not desired to add disjointness axioms for almost each pair of classes. Instead, disjointness axioms should be specified as far up in the class hierarchy as possible.

Also note that there are a number of further approaches for learning disjointness in [38]. Several criteria, including taxonomic overlap (which is what we used here), existing subsumption axioms, semantic similarity etc. are used. Those approaches may be partially integrated in our approaches, while trying to preserve efficiency and ease of use. The DL-Learner enrichment modules currently only expect a SPARQL endpoint and a resource as input, which makes it very easy to integrate them in different existing tools. Using third party tools may increase accuracy of the generated suggestions, but could also make deployment more difficult. These aspects will be considered in the further development of the algorithms in D3.3.2 and D3.3.3.

4.4 Property Domain

For domains of object properties and data properties, there are two possible ways of obtaining data for enrichments: One approach is to issue a single query, which counts the occurrences of types in the subject position of triples having the property. The disadvantage of this approach is that it puts high

computational load on the SPARQL endpoint in case of very large data sets and a very frequently used property. If a property is used in hundreds of millions of triples, the query may generate high load.

Single Query

```
SELECT ?type COUNT(DISTINCT ?ind) WHERE {
  ?ind <$property> ?o.
  ?ind a ?type.
} GROUP BY ?type
```

An alternative implementation is to iterate through all results as shown below. This way, each individual query is inexpensive for the endpoint as the information is obtained in small chunks. Moreover, in DL-Learner, we impose runtime limits on algorithms. This means that we stop iterating through results once a time threshold (by default 10 seconds) has been reached. The score for suggestions is then approximated from the obtained sample. The iterative implementation of property domain enrichment is enabled by default, because of its better scalability.

Iterative Query

```
SELECT DISTINCT ?ind ?type WHERE {
  ?ind <$property> ?o.
  ?ind a ?type.
}
LIMIT $limit
OFFSET $offset
```

4.5 Property Range

For property ranges, we issue different queries depending on whether a resource is a data or object property. The object property case is analogous to learning domains as shown above. For data properties, we make use of the fact that every triple is annotated with its datatype in RDF, i.e. we can just count occurring datatypes.

Object Properties

```
SELECT DISTINCT ?ind ?type WHERE {
  ?s <$property> ?ind.
  ?ind a ?type.
}
LIMIT $limit
OFFSET $offset
```

Data Properties

```
SELECT ?ind (DATATYPE(?val) AS ?datatype) WHERE {
  ?ind <$property> ?val.
}
```

```
LIMIT $limit
OFFSET $offset
```

4.6 Transitive Properties

To detect whether a property might be transitive, we divide closed paths of length 2 for a property by the total number of paths of length 2 for a given property. A path of length 2 of a property p consists of two triples (s, p, o) and (o, p, s'). We call it closed when a triple (s, p, s') exists.

```
SELECT (COUNT(?o) AS ?all) WHERE {
  ?s <$property> ?o.
  ?o <$property> ?o1.
}

SELECT (COUNT(?o2) AS ?transitive) WHERE {
  ?s <$property> ?o.
  ?o <$property> ?o1.
  ?s <$property> ?o1.
}
```

4.7 Functional Properties

For functional properties, we count the number of all cases in which a property has at least two different values:

```
SELECT COUNT(DISTINCT ?s) AS ?all WHERE {
  ?s <$property> ?o.
}

SELECT COUNT(DISTINCT ?s) AS ?nonfunctional WHERE {
  ?s <$property> ?o.
  ?s <$property> ?o1.
  FILTER(?o != ?o1)
}
```

The count is dived by the total number of subjects with this property and subtracted from 1.

4.8 Reflexive Properties

For reflexive properties, we count the number of cases, in which subject and object position of a triple are equal for a given property. This is divided by the number of distinct subjects for this property.

```
SELECT (COUNT(?s) AS ?all) WHERE {
  ?s <$property> ?o.
}

SELECT (COUNT(?s) AS ?reflexive) WHERE {
  ?s <$property> ?s.
}
```

4.9 Irreflexive Properties

```
SELECT (COUNT(?s) AS ?all) WHERE {
  ?s <$property> ?o.
}

SELECT (COUNT(?s) AS ?irreflexive) WHERE {
  ?s <$property> ?o.
  FILTER(?s != ?o)
}
```

4.10 Symmetric Properties

For symmetric properties, we count in how many cases for a given triple (s, p, o) a triple (o, p, s) exists and divide it by the total number of triples with this property.

```
SELECT (COUNT(?s) AS ?all) WHERE {
  ?s <$property> ?o.
}

SELECT (COUNT(?s) AS ?symmetric) WHERE {
  ?s <$property> ?o.
  ?o <$property> ?s.
}
```

4.11 Property Hierarchies

To obtained candidates for subproperty axioms, we first fetch all triples with this property in predicate position. For a candidate property, we then count how often it connects the same subject and object as the input property.

```
SELECT ?p (COUNT(?s) AS ?count) WHERE {
  ?s ?p ?o.
  {
    SELECT ?s ?o WHERE {
      ?s <$property> ?o.
    } LIMIT $limit OFFSET $offset
  }
} GROUP BY ?p
```

Chapter 5

Enrichment Ontology

In the introduction, we discussed that enrichment is usually a semi-automatic process. Each enrichment suggestions generated by an algorithm should be reviewed by a knowledge engineer who can then decide to accept or reject it. Because of this, there is a need for serialising enrichment suggestions such that the generation of them is independent of the process of accepting or rejecting them. Since all enrichment suggestions are OWL axioms, they could simply be written in an RDF or OWL file. However, this might be insufficient, because we lose a lot of metadata this way, which could be relevant for the knowledge engineer. For this reason, we created an enrichment ontology, which is partially building on related efforts in [32] and `http://vocab.org/changeset/schema.html`. Such an interchange format is also relevant, because the process of generating enrichments for all schema elements in very large knowledge bases will often take several hours to complete. Furthermore, it may be desirable to include sufficient metadata to be able to reproduce algorithm runs for creating enrichment suggestions. We briefly describe the main elements of the enrichment ontology.

5.1 Classes

Suggestion Base class for enrichment suggestions.

- **AddSuggestion** Contains suggestions for adding axioms to an ontology.
- **RemoveSuggestion** Contains suggestions for removing axioms from an ontology.

SuggestionSet This class is used to group several suggestions with similar characteristics, e.g. those generated by the same run of an algorithm.

Algorithm The class containing all algorithms.

Creation Contains all processes of creating a suggestion.

Manual Manual enrichment suggestion processes, e.g. a person suggesting a particular change to an ontology.

Automatic Automatic and semi-automatic enrichment suggestions.

AlgorithmRun Contains runs of a particular algorithm.

Parameter Contains parameters of an algorithm.

Change Actually performed changes in an ontology, e.g. an accepted suggestion could become an instance of Change. This can be used to track which changes have already been performed as result of the enrichment process.

ChangeSet A set of instances of Change.

5.2 Object Properties

hasSuggestion Links a set of suggestions to its elements.

Domain: SuggestionSet

Range: Suggestion

hasAxiom Connects a suggestion to the axiom contained in it. Currently, this axiom is stored as Manchester OWL Syntax text string.

Domain: Suggestion

hasParameter Links to a parameter of an algorithm.

Domain: AlgorithmRun

Range: Parameter

creator Specifies who or what has created a set of enrichment suggestions.

Domain: SuggestionSet

Range: Creation

usedAlgorithm Specifies the used algorithm

Domain: AlgorithmRun

Range: Algorithm

hasInput Can be used specify input resources of an algorithm, for instance SPARQL endpoints.

Domain: AlgorithmRun

5.3 Data Properties

confidence The confidence with which a Creator made a suggestion. The confidence is value between 0 and 1 with 0 indicating no confidence and 1 indicating absolute confidence.

Domain: Suggestion

Range: xsd:double

explanation A textual explanation why a suggestion was given. This could be a remark made by a person or a summary of statistical analysis results of an algorithm.

Domain: Suggestion

parameterName The name of a parameter of an algorithm.

Domain: Parameter

parameterValue The value of a parameter of an algorithm.

Domain: Parameter

timestamp Timestamp of the start of automatic process.

Domain: Automatic

version The version of the used algorithm.

Domain: Algorithm

Chapter 6

Using DL-Learner for Enrichment

To install DL-Learner, perform the following steps:

- install Java version 6 or higher (`http://www.java.com/en/download/`)
- download DL-Learner from `http://sourceforge.net/projects/dl-learner/files/DL-Learner/`
- extract the downloaded archive

To run the above enrichment algorithms on a SPARQL endpoint, you can use the provided enrichment interface of DL-Learner. It is a commandline interface, which you can start with "./enrichment" in Unix systems and "enrichment.bat" Windows systems. Figure 6.1 shows the help screen, which is printed when running `enrichment -?`. We will briefly explain the options:

`-e` and `-g` are used to specify the used endpoint and optionally a graph in this endpoint.

`-r` is used to specify the resource (property or class), which should be enriched. The system automatically determines whether this resource is a class, object property or data property and runs the corresponding algorithms. If this parameter if omitted, enrichment for the complete knowledge base is performed, i.e. the system loops over all classes and properties in the SPARQL endpoint and generates enrichments.

The `-f` switch can be used to control the format of the output. By default, the suggestions are just printed to the console, but they can also be saved in a file in combination with the `-o` option, e.g. using the previously described enrichment ontology. This is useful for decoupling the enrichment suggestion

generation process from the actual presentation of those suggestions to a knowledge engineer.

The `-i` switch allows to turn inference on or off. If it is turned on, phase 1 in the process described at the beginning of Chapter 4 is enabled. Powerful reasoning capabilities may improve the quality of suggestions, in particular for those axioms, which rely on knowing the class hierarchy of the knowledge base, e.g. domain and range axioms.

`-t` allows to specify a threshold for enrichment suggestions, i.e. suggestions with a lower score will be omitted.

```
Option                          Description
------                          -----------
-?, -h, --help                  Show help.
-e, --endpoint <URL>            SPARQL endpoint URL to be used.
-f, --format                    Format of the generated output (plain,
                                  rdf/xml, turtle, n-triples).
                                  (default: plain)
-g, --graph [URI]               URI of default graph for queries on
                                  SPARQL endpoint.
-i, --inference [Boolean]       Specifies whether to use inference. If
                                  yes, the schema will be loaded into
                                  a reasoner and used for computing
                                  the scores. (default: true)
-o, --output [File]             Specify a file where the output can be
                                  written.
-r, --resource [URI]            The resource for which enrichment
                                  axioms should be suggested.
-t, --threshold [Double]        Confidence threshold for suggestions.
                                  Set it to a value between 0 and 1.
                                  (default: 0.7)

Additional explanations: The resource specified should be a class,
object property or data property. DL-Learner will try to automatically
detect its type. If no resource is specified, DL-Learner will generate
enrichment suggestions for all detected classes and properties in the
given endpoint and graph. This can take several hours.
```

Figure 6.1: Depiction of the help screen of the enrichment script.

Examples

Obtain enrichment suggestions for the `currency` property in DBpedia Live:

```
-e http://live.dbpedia.org/sparql -g http://dbpedia.org
-r http://dbpedia.org/ontology/currency
```

Output those enrichments to a file `results.txt`:

```
-e http://live.dbpedia.org/sparql -g http://dbpedia.org
-r http://dbpedia.org/ontology/currency -o results.txt
```

Write the enrichments in Turtle syntax in a file using the enrichment ontology:

```
-e http://live.dbpedia.org/sparql -g http://dbpedia.org
-r http://dbpedia.org/ontology/currency -o results.ttl
-f turtle
```

Do the same task with an increased threshold and without inference

```
-e http://live.dbpedia.org/sparql -g http://dbpedia.org
-r http://dbpedia.org/ontology/currency -o results.ttl
-f turtle -t 0.9 -i false
```

Generate all enrichments for DBpedia Live (will take several hours to complete):

```
-e http://live.dbpedia.org/sparql -g http://dbpedia.org
```

Chapter 7

Preliminary Evaluation

To assess the feasibility of our approaches, we evaluated them on DBpedia [3, 2, 14, 21]. DBpedia is one of the most widely used data sets and contains 385 million triples for the English DBpedia edition describing 3.64 million things. We performed an enrichment on the DBpedia Live ontology, which at that point in time (August 2011) consisted of 272 classes, 629 object properties and 706 data properties. We used a confidence threshold of 0.7 for the algorithm runs. Table 7.1 contains basic runtime information on the algorithms. It shows how many enrichment suggestions were made per axiom type, the runtime of the algorithm, the average score and the average of the maximum scores of each algorithm run.

Table 7.2 shows our evaluation results.

In this evaluation we defined recall with respect to the existing DBpedia ontology. For instance, 180/185 in the subClassOf row indicates that we were able to re-learn 180 out of 185 such subclass relationships from the original DBpedia ontology. Higher numbers are an indicator that the methods do not miss many possible enrichment suggestions.

The next column shows how many additional axiom were suggested, i.e. how many axioms were suggested which are not in the original DBpedia ontology.

The last three columns are the result of a manual evaluation. The authors observed at most 100 axioms per type (possibly less, in case fewer than 100 suggestions were made) and evaluated them manually. We put them in three different categories: “yes” indicates that it is likely that they would be accepted by a knowledge engineer, “maybe” are corner cases and “no” are those, which would probably be rejected. Obviously, the evaluation is limited significance, since only the authors themselves judged the resulting axioms. A full evaluation is subject to Deliverable D3.3.2.

algorithm	Avg. #suggestions	Avg. runtime in ms	timeout in %	Avg. score	Avg. maximum score
CELOE	9.1	268134.0	0.00	0.88	0.90
disjoint classes learner	10.0	11957.0	0.00	1.00	1.00
simple subclass learner	2.5	98233.0	0.00	0.95	0.98
disjoint objectproperty	10.0	12384.0	0.16	1.00	1.00
equivalent objectproperty	1.1	12509.0	0.16	0.96	0.96
functional objectproperty	1.0	19990.0	0.48	0.89	0.89
inversefunctional objectproperty	1.0	113590.0	4.29	0.86	0.86
objectproperty domain	3.1	11577.0	0.00	0.93	0.96
objectproperty range learner	2.6	14253.0	0.16	0.84	0.87
object subPropertyOf	1.1	56363.0	0.32	0.93	0.93
symmetric objectproperty	1.0	12730.0	0.32	0.80	0.80
transitive objectproperty	1.0	16830.0	1.91	0.84	0.84
irreflexive objectproperty	1.0	17357.0	0.16	0.97	0.97
reflexive objectproperty	0.0	10013.0	0.16	-	-
disjoint dataproperty	10.0	137204.0	3.97	1.00	1.00
equivalent dataproperty	1.0	161229.0	4.25	0.92	0.92
functional dataproperty	1.0	18281.0	0.42	0.94	0.94
dataproperty domain	2.8	10340.0	0.28	0.94	0.96
dataproperty range learner	1.0	13516.0	0.42	0.96	0.96
data subPropertyOf	1.0	91284.0	4.11	0.88	0.88
OVERALL	3.0	55389.0	1.08	0.92	0.93

Table 7.1: Basic runtime information on the algorithms.

axiom type	recall	additional axioms	Estimated precision no	maybe	yes
SubClassOf	180/185	155	5	20	75
EquivalentClasses	0/0	1812	20	30	50
DisjointClasses	0/0	2449	0	0	100
SubObjectPropertyOf	0/0	45	18	9	18
EquivalentObjectProperties	0/0	40	40	0	0
DisjointObjectProperties	0/0	5670	0	0	100
ObjectPropertyDomain	385/449	675	10	22	68
ObjectPropertyRange	173/435	427	4	59	37
TransitiveObjectProperty	0/0	12	5	5	2
FunctionalObjectProperty	0/0	352	8	18	74
InverseFunctionalObjectProperty	0/0	173	72	3	25
SymmetricObjectProperty	0/0	3	0	0	3
ReflexiveObjectProperty	0/0	0	-	-	-
IrreflexiveObjectProperty	0/0	536	1	0	99
SubDataPropertyOf	0/0	197	86	8	6
EquivalentDataProperties	0/0	213	20	9	71
DisjointDataProperties	0/0	62	0	0	100
DataPropertyDomain	448/493	623	27	33	40
DataPropertyRange	118/597	79	0	0	100
FunctionalDataProperty	14/14	509	4	17	79

Table 7.2: Evaluation results.

In summary, we observed that axioms regarding the class hierarchy basically seem to be more easy to learn than axioms building the property hierarchy. We also noticed that we could suggest new axioms for all axiom types except for the ReflexiveObjectProperty ones. The reason is that DBpedia does not contain corresponding instance data. In general, we believe that reflexivity only amounts to a small proportion in most real world knowledge bases. The low recall for the range axioms of object and data properties is mostly due to either missing or different type information on the triples' objects.

Below, we list some of the observations we made:

- The test set for irreflexive properties contained `dbo:isPartOf`, which is usually considered as a reflexive property. It is the only incorrect suggestion in this test set.
- The 5 missing subclass axioms are as follows:
  1. `dbo:Ginkgo subClassOf: dbo:Plant` (only 1 triple, thus it had low support)
  2. `dbo:MixedMartialArtsLeague subClassOf: dbo:SportsLeague` (only 1 triple, therefore it had low support)
  3. `dbo:VoiceActor subClassOf: dbo:Actor` (only 1 triple, therefore it had low support)
  4. `dbo:PoloLeague subClassOf: dbo:SportsLeague` (only 3 triples, therefore it had low support)
  5. `dbo:Bridge subClassOf: dbo:Building` (none of the subjects is a building)
- For the ObjectPropertyDomain axioms regarding `dbo:hometown`, the results of the learning procedure are as follows: For the existing domain `dbo:Person` we only got a score of 0.3, whereas `dbo:Band` and `dbo:Organisation` achieved a score of approx. 0.7. In this case, the existence of the domain `dbo:Person` makes all instances of `dbo:Band` also of type `dbo:Person`.
- We discovered 3 symmetric object properties, namely `dbo:neighboringMunicipality`, `dbo:sisterCollege` and `dbo:currentPartner`.
- For most of the data properties the learned axioms of the types SubDataPropertyOf and EquivalentDataProperties contained properties of

the DBpedia namespace `http://dbpedia.org/property/`(dbp), e.g. EquivalentDataProperties(`dbo:drugbank`,`dbp:drugbank`). These properties don't have a declared type `owl:DataProperty` but only `rdf:Property`, so the results here have to be dealt with carefully.

- We missed some DataPropertyRanges, because sometimes the defined range in the ontology is a different datatype, compared to the one of the literal values in the triples. For instance `dbo:background` has a defined range `xsd:string`, but in the instance data the literals only have a language tag (which makes them implicit to `rdf:PlainLiteral`), or for instance `dbo:budget` (range in ontology: `xsd:double`, datatype of values: `http://dbpedia.org/datatype/usDollar`).

- In some cases we learned a different datatype, so we missed the existing one and found an additional one. Most of this additional axioms make sense, e.g. for `dbo:populationTotal` we learned `xsd:integer`, whereas in the ontology `xsd:nonNegativeInteger` is defined.

Chapter 8

Conclusion and Pointers

8.1 Conclusion

We presented a set of approaches for schema enrichment, which covers most OWL 2 axioms. Those approaches were implemented and released in the DL-Learner machine learning framework. In our preliminary evaluation, we showed the feasibility of the methods for knowledge bases of the size of DBpedia. The evaluation will be extended in Deliverable D3.3.2.

In future work, we will investigate enhancements of the presented methods as indicated in the discussions of the respective approaches. We also collaborate with partners outside of LOD2 on schema induction. For instance, the methods presented in [36] may be useful to further improve or complement our enrichment results. Later in LOD2, our results will be integrated in the ORE user interface as a result of Deliverable D3.3.3.3.

8.2 Pointers

Homepage: `http://dl-learner.org`
Sourceforge.net Project Page: `http://sourceforge.net/projects/dl-learner/`
Bugs & Feature Requests: `http://sourceforge.net/tracker/?group_id=203619`
Mailing Lists: `http://sourceforge.net/mail/?group_id=203619`
Latest Release: `http://sourceforge.net/projects/dl-learner/files/`

Documentation for DL-Learner (e.g. various configuration options) can be found at `http://dl-learner.org`. We recommend to read `http://dl-learner.org/files/dl-learner-manual.pdf` to get started.

DL-Learner is Open Source and licensed under the GNU General Public License 3. (Copyright (c) 2007-2011, Jens Lehmann).

DL-Learner uses several other libraries. An incomplete list is as follows:

- OWL API (licensed under LGPL)
- Pellet (licensed under AGPL 3 , (c) Clark & Parsia LLC)
- FaCT++ (licensed under LGPL, (c) The Victoria University of Manchester)
- HermiT (licensed under LGPL)
- Jena (Jena license, (c) Copyright Hewlett-Packard)
- Protege (licensed under MPL)

Bibliography

[1] Alan Agresti and Brent A. Coull. Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician*, 52(2):119–126, 1998.

[2] Sören Auer, Chris Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *Proceedings of the 6th International Semantic Web Conference (ISWC)*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2008.

[3] Sören Auer and Jens Lehmann. What have Innsbruck and Leipzig in common? extracting semantics from wiki content. In *Proceedings of the ESWC (2007)*, volume 4519 of *Lecture Notes in Computer Science*, pages 503–517, Berlin / Heidelberg, 2007. Springer.

[4] Franz Baader, Bernhard Ganter, Ulrike Sattler, and Baris Sertkaya. Completing description logic knowledge bases using formal concept analysis. In *IJCAI 2007*. AAAI Press, 2007.

[5] Franz Baader, Baris Sertkaya, and Anni-Yasmin Turhan. Computing the least common subsumer w.r.t. a background terminology. *J. Applied Logic*, 5(3):392–420, 2007.

[6] Liviu Badea and Shan-Hwei Nienhuys-Cheng. A refinement operator for description logics. In *ILP 2000*, volume 1866 of *LNAI*, pages 40–59. Springer, 2000.

[7] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's razor. In *Readings in Machine Learning*, pages 201–204. Morgan Kaufmann, 1990.

[8] Namyoun Choi, Il-Yeol Song, and Hyoil Han. A survey on ontology mapping. *SIGMOD Record*, 35(3):34–41, 2006.

[9] William W. Cohen, Alexander Borgida, and Haym Hirsh. Computing least common subsumers in description logics. In *AAAI 1992*, pages 754–760, 1992.

[10] William W. Cohen and Haym Hirsh. Learning the CLASSIC description logic: Theoretical and experimental results. In *KR 1994*, pages 121–133. Morgan Kaufmann, 1994.

[11] Floriana Esposito, Nicola Fanizzi, Luigi Iannone, Ignazio Palmisano, and Giovanni Semeraro. Knowledge-intensive induction of terminologies from metadata. In *ISWC 2004*, pages 441–455. Springer, 2004.

[12] Nicola Fanizzi, Claudia d'Amato, and Floriana Esposito. DL-FOIL concept learning in description logics. In *ILP 2008*, volume 5194 of *LNCS*, pages 107–121. Springer, 2008.

[13] Sebastian Hellmann, Jens Lehmann, and Sören Auer. Learning of OWL class descriptions on very large knowledge bases. *Int. J. Semantic Web Inf. Syst.*, 5(2):25–48, 2009.

[14] Sebastian Hellmann, Claus Stadler, Jens Lehmann, and Sören Auer. DBpedia live extraction. In *Proc. of 8th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE)*, volume 5871 of *Lecture Notes in Computer Science*, pages 1209–1223, 2009.

[15] Luigi Iannone and Ignazio Palmisano. An algorithm based on counterfactuals for concept learning in the semantic web. In *IEA/AIE 2005*, pages 370–379, June 2005.

[16] Luigi Iannone, Ignazio Palmisano, and Nicola Fanizzi. An algorithm based on counterfactuals for concept learning in the semantic web. *Applied Intelligence*, 26(2):139–159, 2007.

[17] Jens Lehmann. Hybrid learning of ontology classes. In *Machine Learning and Data Mining in Pattern Recognition*, volume 4571 of *LNCS*, pages 883–898. Springer, 2007.

[18] Jens Lehmann. DL-Learner: learning concepts in description logics. *Journal of Machine Learning Research (JMLR)*, 10:2639–2642, 2009.

[19] Jens Lehmann. *Learning OWL Class Expressions*. PhD thesis, University of Leipzig, 2010. PhD in Computer Science.

[20] Jens Lehmann, Sören Auer, Lorenz Bühmann, and Sebastian Tramp. Class expression learning for ontology engineering. *Journal of Web Semantics*, 9:71 – 81, 2011.

[21] Jens Lehmann, Chris Bizer, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - a crystallization point for the web of data. *Journal of Web Semantics*, 7(3):154–165, 2009.

[22] Jens Lehmann and Lorenz Bühmann. Ore - a tool for repairing and enriching knowledge bases. In *Proceedings of the 9th International Semantic Web Conference (ISWC2010)*, Lecture Notes in Computer Science, Berlin / Heidelberg, 2010. Springer.

[23] Jens Lehmann and Lorenz Bühmann. Autosparql: Let users query your knowledge base. In *Proceedings of ESWC 2011*, 2011.

[24] Jens Lehmann and Christoph Haase. Ideal downward refinement in the EL description logic. In *Inductive Logic Programming, 19th International Conference, ILP 2009, Leuven, Belgium*, 2009.

[25] Jens Lehmann and Christoph Haase. Ideal downward refinement in the el description logic. In *Proceedings of the 19th International Conference on Inductive Logic Programming*, volume 5989 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2009.

[26] Jens Lehmann and Pascal Hitzler. Foundations of refinement operators for description logics. In *ILP 2007*, volume 4894 of *LNCS*, pages 161–174. Springer, 2008. Best Student Paper Award.

[27] Jens Lehmann and Pascal Hitzler. A refinement operator based learning algorithm for the ALC description logic. In *ILP 2007*, volume 4894 of *LNCS*, pages 147–160. Springer, 2008. Best Student Paper Award.

[28] Jens Lehmann and Pascal Hitzler. Concept learning in description logics using refinement operators. *Machine Learning journal*, 78(1-2):203–250, 2010.

[29] Francesca A. Lisi. Building rules on top of ontologies for the semantic web with inductive logic programming. *Theory and Practice of Logic Programming*, 8(3):271–300, 2008.

[30] Francesca A. Lisi and Floriana Esposito. Learning SHIQ+log rules for ontology evolution. In *SWAP 2008*, volume 426 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[31] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf, editors. *Foundations of Inductive Logic Programming*, volume 1228 of *LNCS*. Springer, 1997.

[32] Raúl Palma, Peter Haase, Óscar Corcho, and Asunción Gómez-Pérez. Change representation for OWL 2 ontologies. In Rinke Hoekstra and Peter F. Patel-Schneider, editors, *OWLED*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[33] Sebastian Rudolph. Exploring relational structures via FLE. In Karl Erich Wolff, Heather D. Pfeiffer, and Harry S. Delugach, editors, *ICCS 2004*, volume 3127 of *LNCS*, pages 196–212. Springer, 2004.

[34] Baris Sertkaya. OntocomP system description. In Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, and Ulrike Sattler, editors, *Proceedings of the 22nd International Workshop on Description Logics (DL 2009), Oxford, UK, July 27-30, 2009*, volume 477 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[35] Pavel Shvaiko and Jerome Euzenat. Ten challenges for ontology matching. Technical report, August 01 2008.

[36] Johanna Völker and Mathias Niepert. Statistical schema induction. In Grigoris Antoniou, Marko Grobelnik, Elena Paslaru Bontas Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Z. Pan, editors, *ESWC (1)*, volume 6643 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2011.

[37] Johanna Völker and Sebastian Rudolph. Fostering web intelligence by semi-automatic OWL ontology refinement. In *Web Intelligence*, pages 454–460. IEEE, 2008.

[38] Johanna Völker, Denny Vrandecic, York Sure, and Andreas Hotho. Learning disjointness. In *ESWC 2007*, volume 4519 of *LNCS*, pages 175–189. Springer, 2007.

[39] Harris Wu, Mohammad Zubair, and Kurt Maly. Harvesting social knowledge from folksonomies. In *Proceedings of the seventeenth conference on Hypertext and hypermedia*, HYPERTEXT '06, pages 111–114, New York, NY, USA, 2006. ACM.