Automating RDF Dataset Transformation and Enrichment

Mohamed Ahmed Sherif, Axel-Cyrille Ngonga Ngomo, and Jens Lehmann

Department of Computer Science, University of Leipzig, 04109 Leipzig, Germany
{sherif,ngonga,lehmann}@informatik.uni-leipzig.de

Abstract. With the adoption of RDF across several domains come growing requirements pertaining to the completeness and quality of RDF datasets. Currently, this problem is most commonly addressed by manually devising means to enriching an input dataset. The few tools that aim at supporting this endeavour usually focus on supporting the manual definition of enrichment pipelines. In this paper, we present a supervised learning approach based on a refinement operator for enriching RDF datasets. We show how we can use exemplary descriptions of enriched resources to generate accurate enrichment pipelines. We evaluate our approach against 8 manually defined enrichment pipelines and show that our approach can learn accurate pipelines even when provided with a small number of training examples.

1 Introduction

Over the last years, the Linked Data principles have been used across academia and industry to publish and consume structured data [16]. With this adoption of Linked data come novel challenges pertaining to the integration of these datasets for dedicated applications such as tourism, question answering, enhanced reality and many more. Providing consolidated and integrated datasets for these applications demands the specification of data enrichment pipelines, which describe how data from different sources is to be integrated and altered so as to abide by the precepts of the application developer or data user. Currently, most developers implement customized pipelines by compiling sequences of tools manually and connecting them via customized scripts. While this approach most commonly leads to the expected results, it is time-demanding and resource-intensive. Moreover, the results of this effort can most commonly only be reused for new versions of the input data but cannot be ported easily to other datasets. Over the last years, a few frameworks for RDF data enrichment such as LDIF[1] and DEER[2] have been developed. The frameworks provide enrichment methods such as entity recognition [22], link discovery [15] and schema enrichment [4]. However, devising appropriate configurations for these tools can prove a difficult endeavour, as the tools require (1) choosing the right sequence of enrichment functions and (2) configuring these functions adequately. Both the first and second task can be tedious.

[1] http://ldif.wbsg.de/

[2] http://aksw.org/Projects/DEER.html

In this paper, we address this problem by presenting a supervised machine learning approach for the automatic detection of enrichment pipelines based on a refinement operator and self-configuration algorithms for enrichment functions. Our approach takes pairs of concise bounded descriptions (CBDs) of resources $\{(k_1, k'_1) \ldots (k_n, k'_n)\}$ as input, where k'_i is the enriched version of k_i. Based on these pairs, our approach can learn sequences of atomic enrichment functions that aim to generate each k'_i out of the corresponding k_i. The output of our approach is an enrichment pipeline that can be used on whole datasets to generate enriched versions.

Overall, we provide the following core contributions:

– Definition of a supervised machine learning algorithm for learning dataset enrichment pipelines.
– Provision of self configuration algorithms for five atomic enrichment steps (and re-using and existing self configuration for atomic enrichment via linking).
– An evaluation on eight manually defined enrichment pipelines on real datasets.

The paper is structured as follows: In Section 2 we present the basic notions used in the rest of the paper. Then, in Section 3, we present our refinement operator for knowledge base enrichment pipeline construction. We introduce our refinement-operator-based approach to learning enrichment pipelines in Section 4, which can work on arbitrary sets of atomic enrichment functions. The self-configuration approach implemented for each of these enrichment functions is presented in Section 5. The evaluation of the learning algorithm is presented in Section 6. Here, we evaluate how well our approach performs against manually created specifications. Related approaches and tools for linked data integration and cleaning are described in Section 7. Finally, we conclude and present some of the planned work on our approach in Section 8.

2 Preliminaries

Enrichment: Let $\mathcal{K}$ be the set of all RDF knowledge bases. Let $K \in \mathcal{K}$ be a finite RDF knowledge base. K can be regarded as a set of triples $(s, p, o) \in (\mathcal{R} \cup \mathcal{B}) \times \mathcal{P} \times (\mathcal{R} \cup \mathcal{L} \cup \mathcal{B})$, where $\mathcal{R}$ is the set of all resources, $\mathcal{B}$ is the set of all blank nodes, $\mathcal{P}$ the set of all predicates and $\mathcal{L}$ the set of all literals.

Given a knowledge base K, the idea behind *knowledge base enrichment* is to find an *enrichment pipeline* $M : \mathcal{K} \to \mathcal{K}$ that maps K to an enriched knowledge base K' with $K' = M(K)$. We define M as an ordered list of *atomic enrichment functions* $m \in \mathcal{M}$, where $\mathcal{M}$ is the set of all atomic enrichment functions. $2^{\mathcal{M}}$ is used to denote the power set of $\mathcal{M}$, i.e. the set of all enrichment pipelines. The order of elements in M determines the execution order, e.g. for an $M = (m_1, m_2, m_3)$ this means that m_1 will be executed first, then m_2, finally m_3. Formally,

$$M = \begin{cases} \phi & \text{if } K = K', \\ (m_1, \ldots, m_n), \text{where } m_i \in \mathcal{M}, 1 \leq i \leq n & \text{otherwise,} \end{cases} \tag{1}$$

where ϕ is the empty sequence. Moreover, we denote the number of elements of M with $|M|$. Considering that a knowledge base is simply a set of triples, the task of any

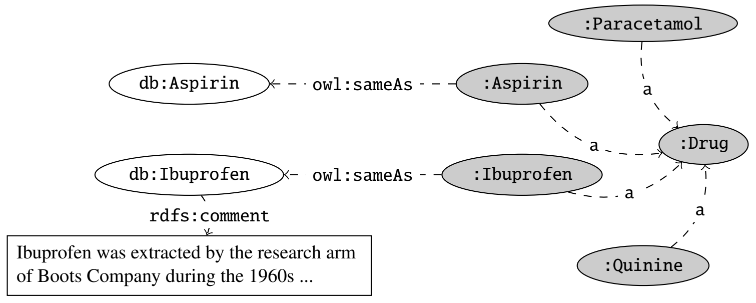


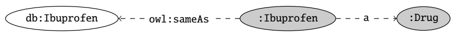
Fig. 1: RDF graph of the running example. Ellipses are RDF resources, literals are rectangular nodes. Gray nodes stand for resources in the input knowledge base while nodes with a white background are part of an external knowledge base.

atomic enrichment function is to (1) determine a set of triples Δ^+ to be added the source knowledge base and/or (2) determine a set of triples Δ^- to be deleted from the source knowledge base. Any other enrichment process can be defined in terms of Δ^+ and Δ^-, e.g. altering triples can be represented as combination of addition and deletion.

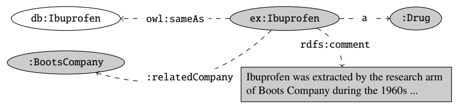
In this article we cover two problems: (1) how to create self-configurable atomic enrichment functions $m \in \mathcal{M}$ capable of enriching a dataset and (2) how to automatically generate an enrichment pipeline M. As running example, we use the portion of *DrugBank* shown in Figure 1. The goal of the enrichment here is to gather information about companies related to drugs for a market study. To this end, the `owl:sameAs` links to *DBpedia* (prefix `db`) need to be dereferenced. Their `rdfs:comment` then needs to be processed using an entity spotter that will help retrieve resources such as the `Boots Company`. Then, these resources need to be attached directly to the resources in the source knowledge base, e.g., by using the `:relatedCompany` property. Finally, all subjects need to be conformed under one subject authority (prefix `ex`).

Refinement Operators: Below, we give definitions of refinement operators and their properties. Refinement operators have traditionally been used, e.g. in [11], to traverse search spaces in structured machine learning problems. Their theoretical properties give an indication of how suitable they are within a learning algorithm in terms of accuracy and efficiency.

Definition 1 (Refinement Operator and Properties). *Given a quasi-ordered space $(S, \succcurlyeq)$ an upward refinement operator r is a mapping from S to 2^S such that $\forall s \in S : s' \in r(s) \Rightarrow s \succcurlyeq s'$. s' is then called a generalization of s. A set $M_2 \in \mathcal{M}$ belongs to the refinement chain of $M_1 \in \mathcal{M}$ iff $\exists i \in \mathbb{N} : M_2 \in r^i(M_1)$, where $r^0(M) = M$ and $r^i(M) = r(r^{i-1}(M))$. A refinement operator r over the quasi-ordered space $(S, \succcurlyeq)$ can abide by the following criteria. r is finite iff $r(s)$ is finite for all $s \in S$. r is proper if $\forall s \in S, s' \in r(s) \Rightarrow s \neq s'$. r is complete if for all s and s', $s' \succcurlyeq s$ implies that there is a refinement chain between s and s'. A refinement operator r over the space $(S, \succcurlyeq)$ is redundant if two different refinement chains can exist between $s \in S$ and $s' \in S$.*



(a) Non-enriched CBD of Ibuprofen


db:Ibuprofen
owl:sameAs
ex:Ibuprofen
a
:Drug
rdfs:comment
:BootsCompany
:relatedCompany
Ibuprofen was extracted by the research arm of Boots Company during the 1960s ...

(b) Enriched CBD of Ibuprofen

Fig. 2: Ibuprofen concise bound description before and after enrichment

3 Knowledge Base Enrichment Refinement Operator

Our refinement operator expects the set of atomic enrichment functions $\mathcal{M}$ as input and returns an enrichment pipeline M as output. Each positive example $e \in \mathcal{E}$ is a pair of CBDs (k, k'), with $k \subseteq K$ and $k' \subseteq K'$, the K' stands for the enriched version of K. Note that we model CBDs as sets of RDF triples. Moreover, we denote the resource with the CBD k as *resource(k)*. For our running example, the set $\mathcal{E}$ could contain the pair shown in Figure 2a as k and in Figure 2b as k'.

The set of all first elements of the pairs contained in $\mathcal{E}$ is denoted *source($\mathcal{E}$)*. The set of all second elements is denoted *target*($\mathcal{E}$). To compute the refinement pipeline M, we employ an upward refinement operator (which we dub ρ) over the space $2^{\mathcal{M}}$ of all enrichment pipelines. We write $M \supseteq M'$ when M' is a subsequence of M, i.e., $m_i \in M \rightarrow m_i = m'_i$, where m_i resp. m'_i is the ith element of M resp. M'.

Proposition 1 (Induced quasi-ordering)**.** $\supseteq$ *induces a quasi-ordering over the set* $2^{\mathcal{M}}$.

Proof. The reflexivity of $\supseteq$ follows from each M being a subsequence of itself. The transitivity of $\supseteq$ follows from the transitivity of the subsequence relation. Note that $\supseteq$ is also antisymmetric. □

We define our refinement operator over the space $(2^{\mathcal{M}}, \supseteq)$ as follows:

$$\rho(M) = \bigcup_{\forall m \in \mathcal{M}} M \mathbin{+\!\!+} m \qquad (\mathbin{+\!\!+} \text{ is the list append operator}) \tag{2}$$

We define precision $P(M)$ and recall $R(M)$ achieved by an enrichment pipeline on $\mathcal{E}$ as

$$P(M) = \frac{\left| \bigcup\limits_{k \in source(\mathcal{E})} M(k) \bigcap \bigcup\limits_{k' \in target(\mathcal{E})} k' \right|}{\left| \bigcup\limits_{k \in source(\mathcal{E})} M(k) \right|}, R(M) = \frac{\left| \bigcup\limits_{k \in source(\mathcal{E})} M(k) \bigcap \bigcup\limits_{k' \in target(\mathcal{E})} k' \right|}{\left| \bigcup\limits_{k' \in target(\mathcal{E})} k' \right|}. \tag{3}$$

The F-measure $F(M)$ is then

$$F(M) = \frac{2P(M)R(M)}{P(M) + R(M)}. \tag{4}$$

Using Figure 2a from our running example as source and Figure 2b as target with the CBD of `:Iboprufen` being the only positive example, an empty enrichment pipeline $M = \phi$ would have a precision of 1, a recall of $\frac{3}{4}$ and an F-measure of $\frac{6}{7}$. Having defined our refinement operator, we now show that ρ is finite, proper, complete and not redundant.

Proposition 2. *ρ is finite.*

Proof. This is a direct consequence of $\mathcal{M}$ being finite. □

Proposition 3. *ρ is proper.*

Proof. As the quasi order is defined over subsequences, i.e. the space $(2^{\mathcal{M}}, \supseteq)$, and we have $|M'| = |M| + 1$ for any $M' \in \rho(M)$, ρ is trivially proper. □

Proposition 4. *ρ is complete.*

Proof. Let M resp. M' be an enrichment pipeline of length n resp. n' with $M' \supseteq M$. Moreover, let m'_i be the ith element of M'. Per definition, $M +\!\!+\ m'_{n+1} \in \rho(M)$. Hence, by applying ρ $n' - n$ times, we can generate M' from M. We can thus conclude that ρ is complete. □

Proposition 5. *ρ is not redundant.*

Proof. ρ being redundant would mean that there are two refinement chains that lead to a single refinement pipeline M. As our operator is equivalent to the list append operation, it would be equivalent to stating that two different append sequences can lead to the same sequence. This is obviously not the case as each element of the list M is unique, leading to exactly one sequence that can generate M. □

4 Learning Algorithm

The learning algorithm is inspired by refinement-based approaches from inductive logic programming. In those algorithms, a search tree is iteratively build up using heuristic search via a fitness function. We formally define a node N in a search tree to be a triple (M, f, s), where M is the *enrichment pipeline*, $f \in [0, 1]$ is the F-measure of M (see Equation 4), and $s \in \{normal, dead\}$ is the status of the node. Given a search tree, the heuristic selects the fittest node in it, where fitness is based on both F-measure and complexity as defined below.

4.1 Approach

For the automatic generation of enrichment pipeline specifications, we created a learning algorithm based on the previously defined refinement operator. The pseudo-code of our algorithm is presented in Algorithm algorithm 1.

Our learning algorithm has two inputs: a set of positive examples $\mathcal{E}$ and a set of atomic enrichment operators $\mathcal{M}$. $\mathcal{E}$ contains pairs of (k, k') where each k contains a CBD of one resource from an arbitrary source knowledge base K and k' contains the CBD of the same resource after applying some manual enrichment. Given $\mathcal{E}$, the goal of our algorithm is to learn an enrichment pipeline M that maximizes $F(M)$ (see Equation 4).

As shown in Algorithm algorithm 1, our approach starts by generating an empty refinement tree τ which contains only an empty root node. Using $\mathcal{E}$, the algorithm then accumulates all the original CBDs in k ($\textsc{Source}(\mathcal{E})$). Using the same procedure, k' is accumulated from $\mathcal{E}$ as the knowledge base containing the enriched version of k

(TARGET($\mathcal{E}$)). Until a termination criterion holds (see Section 4.3), the algorithm keeps expanding the most promising node (see Section 4.2). Finally, the algorithm ends by returning the best pipeline found in τ: (GetPipeline(GetMaxQualityNode(τ))).

Having a *most promising node t* at hand, the algorithm first applies our refinement operator (see Equation 2) against the most promising enrichment pipeline M_{old} included in t to generate a set of atomic enrichment functions $\mathcal{M} \leftarrow \rho(M_{old})$. Consequently, using both k_{old} (as the knowledge base generated by applying M_{old} against k) and k', the algorithm applies the self configuration process of the current atomic enrichment function $m \leftarrow$ SELFCONFIG(m, k_{old}, k) to generate a set of parameters P (a detailed description for this process is found in Section 5). Afterwards, the algorithm runs m against k_{old} to generate the new enriched knowledge base $k_{new} \leftarrow m(k_{old}, P)$. A dead node $N \leftarrow$ CREATENODE(M, 0, *dead*) is created in two cases: (1) m is inapplicable to k_{old} (i.e., $P == null$) or (2) m does no enrichment at all (i.e., k_{new} is isomorphic[3] to k_{old}). Otherwise, the algorithm computes the F-measure f of the generated dataset k_{new}. M along with f are then used to generate a new search tree node $N \leftarrow$ CREATENODE(M, f, *normal*)). Finally, N is added as a child of t (ADDCHILD(t, N)).

4.2 Most promising Node Selection

Here we describe the process of selecting the most promising node $t \in \tau$ as in GETMOSTPROMISINGNODE() subroutine in Algorithm algorithm 1. First, we define *node complexity* as linear combination of the node's children count and level. Formally,

Definition 2 (Node Complexity). *The complexity of a node* $N = (M, f, s)$ *in a refinement tree* τ *is a function* $c : N \times \tau \to [0, 1]$ *, where* $c(N, \tau) = \alpha\frac{|N_c|}{|\tau|} + \beta\frac{N_l}{\tau_d}$, $|N_c|$ *is N's number of children,* $|\tau|$ *is the total number of nodes in* τ, N_l *is N's level,* τ_d *is* τ*'s depth,* α *is the children penalty weight,* β *is the level penalty weight and* $\alpha + \beta = 1$. *Seeking for simplicity, we will use the* $c(N)$ *instead of* $c(N, \tau)$ *in the rest of this paper.*

We can then define the fitness $f(N)$ of a *normal* node N as the difference betweens its enrichment pipeline F-measure (Equation 4) and weighted complexity. $f(N)$ is zero for *dead* nodes. Formally,

Definition 3 (Node fitness). *Let* $N = (M, f, s)$ *be a node in a refinement tree* τ*, N's fitness is the function*

$$f(N) = \begin{cases} 0 & \textit{if } s = \textit{dead}, \\ F(M) - \omega c(N) & \textit{if } s = \textit{normal}. \end{cases} \tag{5}$$

where M is the enrichment pipeline contained in the node N, ω *is the complexity weight and* $0 \leq \omega \leq 1$.

Note, that we use the *complexity* of pipelines as second criterion, which makes the algorithm (1) more flexible in searching less explored areas of the search space, and (2) leads to simpler specification being preferred over more complex ones (Occam's razor[3]). The parameter ω can be used to control the tradeoff between a greedy search ($\omega = 0$) and search strategies closer to breadth first search ($\omega > 0$). The fitness function can be defined independently of the core learning algorithm.

[3] http://www.w3.org/TR/rdf11-concepts/

Algorithm 1: Enrichment Pipeline Learner

```
input  : $X^+$ : Set of positive examples,
         $\mathcal{E}$ : Set of atomic enrichment functions
output: $E$ : Enrichment pipeline
initialize refinement tree $\tau$
$\tau \leftarrow$ CreateRootNode();
$k \leftarrow$ Source($X^+$) ;
$k' \leftarrow$ Target($X^+$) ;
repeat
    Expand most promising node of $\tau$;
    $t \leftarrow$ GetMostPromisingNode($\tau$);
    $E_{old} \leftarrow$ GetPipeline($t$);
    $\mathcal{E} \leftarrow \rho(E_{old})$;
    Create a child of $t$ for each $e \in \mathcal{E}$ ;
    for $e \in \mathcal{E}$ do
        $k_{old} \leftarrow E_{old}(k)$;
        $P \leftarrow$ SelfConfig($e, k_{old}, k'$);
        $k_{new} \leftarrow e(k_{old}, P)$;
        if $P == null$ or $k_{new} == k_{old}$ then
            $N \leftarrow$ CreateNode($E$, 0, dead);
        else
            $f \leftarrow$ F($e$);
            $N \leftarrow$ CreateNode($E, f$, normal);
        AddChild($t, N$);
until TerminationCriterionHolds($\tau$);
return GetPipeline(GetMaxQualityNode($\tau$));
```

Consequently, the most promising node is the node with the maximum fitness through the whole refinement tree τ. Formally, the most promising node t is defined as $t = \underset{\forall N \in \tau}{\arg\max}\, f(N)$, where N is not a *dead* node.

4.3 Termination Criteria

The subroutine TerminationCriterionHolds() in Algorithm algorithm 1 can check manifold termination criteria depending on configuration: (1) optimal enrichment pipeline found (i.e., a fixpoint is reached), (2) maximum number of iterations reached, (3) maximum number of refinement tree nodes reached, or a combination of the aforementioned criteria. Note that the termination criteria can be defined independently of the core learning algorithm.

5 Self-Configuration

To learn an appropriate specification from the input positive examples, we need to develop self-configuration approaches for each of our framework's atomic enrichment functions. The input for each of these self-configuration procedures is the same set of positive examples $\mathcal{E}$ provided to our pipeline learning algorithm (algorithm algorithm 1). The goal of the self-configuration process of an enrichment function is to generate a set of parameters $P = \{(mp_1, v_1), \ldots, (mp_m, v_m)\}$ able to reflect $\mathcal{E}$ as well as possible. In cases when insufficient data is contained in $\mathcal{E}$ to carry out the self-configuration process, an empty list of parameters is returned to indicate inapplicability of the enrichment function.

5.1 Dereferencing Enrichment Functions

The idea behind the self-configuration process of the enrichment by *dereferencing* is to find the set of predicates D_p from the enriched CBDs that are missing from source CBDs. Formally, for each CBD pair (k, k') construct a set $D_p \subseteq \mathcal{P}$ as follows: $D_p = \{p' : (s', p', o') \in k'\} \backslash \{p : (s, p, o) \in k\}$. The dereferencing enrichment function will *dereference* the object of each triple of k_i given that this object is an external URI, i.e. all o in k_i with $(s, p, o) \in k_i$, $o \in \mathcal{R}$ and o is not in the local namespace of the dataset will be dereferenced. Dereferencing an object returns a set of triples. Those are filtered using the previously constructed property set D_p, i.e. when dereferencing o the enrichment function only retains triples with subject o and a predicate contained in D_p. The resulting set of triples is added to the input dataset.

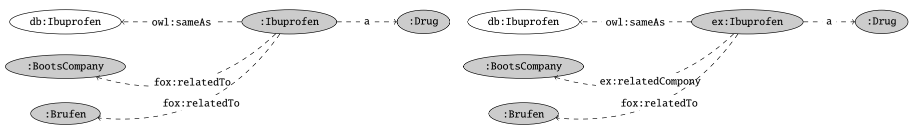
We illustrate the process using our running example: In the first step, we compute the set $D_p = \{$`:relatedCompany`, `rdfs:comment`$\}$ which consists of the properties occurring in the target but not in the source CBD. In the second step, we collect the set of resources to dereference, which only consists of the element `db:Ibuprofen`. In the third step, we perform the actual dereferencing operation and retain triples for which the subject is `db:Ibuprofen` and the predicate is either `:relatedCompany` or `rdfs:comment`. In our example, no triples with predicate `:relatedCompany` exist, but we will find the desired triple (`db:Ibuprofen`, `rdfs:comment`, `"Ibuprofen ..."`), which is then added to the input dataset.

5.2 Linking Module

The linking module relies on LIMES' EUCLID [18] algorithm to perform link discovery. The idea behind this approach is to first perform an automatic property matching. Thereafter, the correct threshold for the matched properties are determined by using a grid search approach in combination with an objective function that is to be maximized. More details on this algorithm can be found in [18].

5.3 NLP Enrichment Function

The basic idea here is to enable the extraction of all possible named entities types. If this leads to the retrieval of too many entities, the unwanted predicates and resources can be discarded (for example using the *filter-based enrichment*). Moreover, the desired predicates will be transformed (e.g., by using the *conformation*-based enrichment). The self-configuration of the NLP enrichment function is static and consists of using one parameter `(NERType, all)`. In our current implementation, we use FOX [17].



(a) NLP enriched CBD of Ibuprofen (b) Conformed CBD of Ibuprofen

Fig. 3: Ibuprofen CBD after NLP and predicate conformation enrichment

The applications of the NLP self configuration to our running example generates all possible entities included in the literal object of the `rdfs:comment` predicate. The result is a set of related named entities all of them related to our `ex:Iboprufen` object by the default predicate `fox:relatedTo` as shown Figure 3a. In the following 2 sections we will see how our enrichment functions can refine some of the generated triples and delete others.

5.4 Conformation Enrichment Functions

The *conformation*-based enrichment currently allows for both *subject-authority-based conformation* and *predicate-based conformation*. The self-configuration process of *subject-authority-based conformation* starts by finding the most frequent subject authority rk in $source(\mathcal{E})$. Also, it finds the most frequent subject authority rk' in the target dataset $target(\mathcal{E})$. Then this self-configuration process generates the two parameters: (`sourceSubjectAuthority`, rk) and (`targetSubjectAuthority`, rk'). After that, the self-configuration process replaces each subject authority rk in $source(\mathcal{E})$ by rk'.

Back to our running example, the authority self-conformation process generates the two parameters (`sourceSubjectAuthority`, `":"`) and (`targetSubjectAuthority`, `"ex:"`). Replacing each `":"` by `"ex:"` generates, in our example, the new conformed URI `"ex:Iboprufen"`.

We define two predicates $p_1, p_2 \in \mathcal{P}$ to be *interchangeable* (denoted $p_1 \leftrightarrows p_2$) if both of them have the same subject and object. Formally, $\forall p_1, p_2 \in \mathcal{P} : p_1 \leftrightarrows p_2 \iff \exists s, o \mid (s, p_1, o) \wedge (s, p_2, o)$.

The idea of the self-configuration process of the *predicate conformation* is to change each predicate in the source dataset to its *interchangeable* predicate in the target dataset. Formally, find all pairs $(p_1, p_2) \mid \exists s, p_1, o \in k \wedge \exists s, p_2, o \in k' \wedge (s, p_1, o) \in k \wedge (s, p_2, o) \in k'$. Then, for each pair (p_1, p_2) create two self-configuration parameters (`sourceProperty`, p_1) and (`targetProperty`, p_2). The predicate conformation will replace each occurrence of p_1 by p_2.

In our example, let us suppose that we ran the NLP-based enrichment first then we got a set of related named entities all of them related to our `ex:Iboprufen` object by the default predicate `fox:relatedTo` as shown in Figure 3a. Subsequently, applying the predicate conformation self-configuration will generate (`sourceProperty`, `fox:relatedTo`) and (`targetProperty`, `ex:relatedCompany`) parameters. Consequently, the predicate conformation module will replace `fox:relatedTo` by `ex:relatedCompany` to generate Figure 3b.

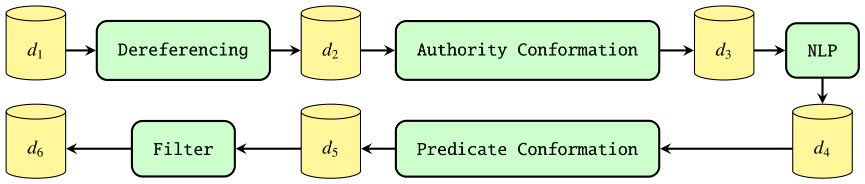


Fig. 4: Graph representation of the learned pipeline of our running example, where d_1 is the positive example source presented in Figure 2a and d_6 is the positive example target presented in Figure 2b.

5.5 Filter Enrichment Function

The idea behind the self-configuration of *filter*-based enrichment is to preserve only valuable triples in the source CBDs k and discard any unnecessary triples so as to achieve a better match to k'. To this end, the self-configuration process starts by finding the intersection between source and target examples $I = \bigcup_{(k,k')\in\mathcal{E}} k \cap k'$. After that, it generates an enrichment function based on a SPARQL query which is only preserving predicates in I. Formally, the self-configuration results in the parameter set $P = \bigcup_{p\in K\cap K'\cap\mathcal{P}} p$.

Back to our running example, let us continue from the situation in the previous section (Figure 3b). Performing the self-configuration of filters will generate $P = \{$`fox:relatedTo`$\}$. Actually applying the filter enrichment function will remove all unrelated triples containing the predicate `fox:relatedTo`. Figure 4 shows a graph representation for the whole learned pipeline for our running example.

6 Evaluation

The aim of our evaluation was to quantify how well our approach can automate the enrichment process. We thus assumed being given manually created training examples and having to reconstruct a possible enrichment pipeline to generate target CBDs from the source CBDs. In the following, we present our experimental setup including the pipelines and datasets used. Thereafter, we give an overview of our results, which we subsequently discuss in the final part of this section.

6.1 Experimental Setup

We used three publicly available datasets for our experiments:

1. From the biomedical domain, we chose *DrugBank*[4] as our first dataset. We chose this dataset because it is linked with many other datasets[5] , from which we can extract enrichment data using our atomic enrichment functions. For our experiments

[4] *DrugBank* is the Linked Data version of the DrugBank database, which is a repository of almost 5000 FDA-approved small molecule and biotech drugs, for RDF dump see `http://wifo5-03.informatik.uni-mannheim.de/drugbank/drugbank_dump.nt.bz2`

[5] See `http://datahub.io/dataset/fu-berlin-drugbank` for complete list of linked dataset with *DrugBank*.

we deployed a manual enrichment pipeline M_{manual}, in which we enrich the drug data found in *DrugBank* using abstracts dereferenced from *DBpedia*, then we conform both *DrugBank* and *DBpedia* source authority URIs to one unified URI. For *DrugBank* we manually deployed two experimental pipelines:

   – $M^1_{DrugBank} = (m_1, m_2)$, where m_1 is a dereferencing function that dereferences any `dbpedia-owl:abstract` from DBpedia and m_2 is an authority conformation function that conforms the *DBpedia* subject authority[6] to the target subject authority of *DrugBank*[7].
   – $M^2_{DrugBank} = M^1_{DrugBank} \mathbin{+\!\!+} m_3$, where m_3 is an authority conformation function that conforms *DrugBank*'s authority to the *Example* authority[8].

2. From the music domain, we chose the *Jamendo*[9] dataset. We selected this dataset as it contains a substantial amount of embedded information hidden in literal properties such as `mo:biography`. The goal of our enrichment process is to add a geospatial dimension to *Jamendo*, e.g., location of a recording or place of birth of a musician. To this end, we deployed a manual enrichment pipeline, in which we enrich *Jamendo*'s music data by adding additional geospatial data found by applying the NER of the NLP enrichment function against `mo:biography`. For *Jamendo* we deploy manually one experimental pipeline:
   – $M^1_{Jamendo} = \{m_4\}$, where m_4 is an NLP function that find *locations* in `mo:biography`.
3. From the multi-domain knowledge base *DBpedia* [12] we selected the class `AdministrativeRegion` to be our third dataset. As DBpedia is a knowledge base with a complex ontology, we build a set of 5 pipelines of increasing complexity:
   – $M^1_{DBpedia} = \{m_5\}$, where m_5 is an authority conformation function that conforms the *DBpedia* subject authority to the *Example* target subject authority.
   – $M^2_{DBpedia} = m_6 \mathbin{+\!\!+} M^1_{DBpedia}$, where m_6 is a dereferencing function that dereference any `dbpedia-owl:ideology`.
   – $M^3_{DBpedia} = M^2_{DBpedia} \mathbin{+\!\!+} m_7$, where m_7 is a NLP function that find *all* named entities in `dbpedia-owl:abstract`.
   – $M^4_{DBpedia} = M^3_{DBpedia} \mathbin{+\!\!+} m_8$, where m_8 is a filter function that filters for abstracts.
   – $M^5_{DBpedia} = M^3_{DBpedia} \mathbin{+\!\!+} m_9$, where m_9 is a predicate conformation function that conforms the source predicate `dbpedia-owl:abstract` to the target predicate of `dcterms:abstract`.

Altogether, we manually generated a set of 8 pipelines, which we then applied against their respective datasets. The resulting pairs of CBDs were used to generate the positive examples, which were simply randomly selected pairs of distinct CBDs. All generated pipelines are available at the project web site.[10]

All experiments were carried out on a 8-core PC running *OpenJDK* 64-Bit Server 1.6.0_27 on *Ubuntu* 12.04.2 LTS. The processors were 8 Hexa-core *AMD Opteron* 6128 clocked at 2.0 GHz. Unless stated otherwise, each experiment was assigned 6 GB of

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

[7] `http://wifo5-04.informatik.uni-mannheim.de/drugbank/resource/drugs`

[8] `http://example.org`

[9] *Jamendo* contains a large collection of music related information about artists and recordings, for RDF dump see `http://moustaki.org/resources/jamendo-rdf.tar.gz`

[10] `https://github.com/GeoKnow/DEER/evaluations/pipeline_learner`

Table 1: Test of the effect of ω on the learning process using the *Drugbank* dataset, where $|\mathcal{E}| = 1$, M is the manually created pipeline, $|M|$ is the complexity of M, $T_{M(KB)}$ is the time for applying M to the entire dataset, M' is the pipeline generated by our algorithm, T_l is the learning time, $|\tau|$ is the size of the refinement tree τ, I_n is the number of iteration done by our learning algorithm, and all times are in minutes.

ω	$\|M\|$	$T_{M(KB)}$	$\|M'\|$	$T_{M'(KB)}$	T_l	$\|\tau\|$	I_n	$P(M')$	$R(M')$	$F(M')$
0	3	3.7	1	0.3	0.3	61	10	1.0	0.99	0.99
0.25	3	3.9	1	0.3	0.2	61	10	1.0	0.99	0.99
0.50	3	3.8	1	0.3	0.1	61	10	1.0	0.99	0.99
0.75	3	3.8	3	0.4	0.1	25	4	1.0	1.0	1.0
1.0	3	3.9	1	0.5	0.2	61	10	1.0	0.99	0.99

memory. As termination criteria for our experiments, we used (1) a maximum number of iterations of 10 or (2) an optimal enrichment pipeline found.

6.2 Results

We carried out two sets of experiments to evaluate our refinement based learning algorithm. In the first set of experiments, we tested the effect of the complexity weight ω to the search strategy of our algorithm. The results are presented in Table 1. In the second set of experiments, we test the effect of the number of positive examples $|\mathcal{E}|$ on the generated F-measure Results are presented in Table 2.

Configuration of the Search Strategy We ran our approach with varying values of ω to determine the value to use throughout our experiments. This parameter is used for configuring the search strategy in the learning algorithm, in particular the bias towards simple pipelines. As shown in Section 4.2, this is achieved by multiplying ω with the node complexity and subtracting this as a penalty from the node fitness. To configure ω, we used the first pipeline $M^1_{DrugBank}$. The results suggest that setting ω to 0.75 leads to the best results in this particular experiment. We thus adopted this value for the other studies.

Effect of Positive Examples We measured the F-measure achieved by our approach on the datasets at hand. The results shown in Table 2 suggest that when faced with data as regular as that found in the datasets Drugbank, DBpedia and Jamendo, our approach really only needs a single example to be able to reconstruct the enrichment pipeline that was used. This result is particularly interesting, because we do not always generate the manually created reference pipeline described in the previous subsection. In many cases, our approach detects a different way to generate the same results. In most cases (71.4%) the pipeline it learns is actually shorter than the manually created pipeline. However, in some cases (4.7%) our algorithm generated a longer pipeline to emulate the manual configuration. As an example, in case of $M^1_{Jamendo}$ the manual configuration was just one enrichment function, i.e, NLP-based enrichment to find all *locations* in `mo:biography`. Our algorithm learns this single manually configured enrichment as

Table 2: Test of the effect of increasing number of positive examples in the learning process. For this experiment $\omega = 0.75$, M is the manually created pipeline, $|M|$ is the size of M, $T_{M(KB)}$ is the time for applying the manual pipeline to the entire dataset, M' is the pipeline generated by our algorithm, T_l is the learning time, $|\tau|$ is the size of the refinement tree τ, I_n is the number of iterations performed by our learning algorithm, and all times are in minutes.

M	$\lvert\mathcal{E}\rvert$	$\lvert M\rvert$	$T_{M(KB)}$	$\lvert M'\rvert$	$T_{M'(KB)}$	T_l	$\lvert\tau\rvert$	I_n	$P(M')$	$R(M')$	$F(M')$
$M^1_{DBpedia}$	1	1	0.2	1	1.6	1.3	7	1	1.0	1.0	1.0
	2	1	0.2	1	1.8	1.3	7	1	1.0	1.0	1.0
$M^2_{DBpedia}$	1	2	23.3	1	0.1	0.2	7	1	1.0	0.99	0.99
	2	2	15	2	17	0.3	55	9	0.99	1.0	0.99
$M^3_{DBpedia}$	1	3	14.7	3	15.2	6.1	55	9	1.0	0.99	0.99
	2	3	15	2	15.1	0.1	55	9	0.99	0.99	0.99
$M^4_{DBpedia}$	1	4	0.4	2	0.1	0.7	13	2	0.99	0.99	0.99
	2	4	0.6	2	0.3	0.9	13	2	0.99	1.0	0.99
$M^5_{DBpedia}$	1	5	22	2	0.1	0.7	13	2	1.0	1.0	1.0
	2	5	25.5	2	0.2	0.9	13	2	1.0	1.0	1.0
$M^1_{DrugBank}$	1	2	3.5	1	4.1	0.1	61	10	0.99	0.99	0.99
	2	2	3.6	1	3.4	0.1	61	10	0.99	0.99	0.99
$M^2_{DrugBank}$	1	3	25.2	1	0.1	0.1	61	10	1.0	0.99	0.99
	2	3	22.8	1	0.1	0.1	61	10	1.0	0.99	0.99
$M^1_{Jamendo}$	1	1	10.9	2	10.6	0.1	13	2	0.99	0.99	0.99
	2	1	10.4	2	10.4	0.1	7	1	0.99	0.99	0.99

(1) an NLP enrichment function that extracts all named entities types and then (2) a filter enrichment function that filters all non-location triples. Our results also suggest that our approach scales when using a small number of positive example as on average the learning time for one positive example was around 48 seconds.

7 Related Work

Linked Data Enrichment is an important topic for all applications that rely on a large number of knowledge bases and necessitate a unified view on this data, e.g., Question Answering frameworks [13], Linked Education [6] and all forms semantic mashups [9]. In recent work, several challenges and requirements to Linked Data consumption and integration have been pointed out [14]. For example, the R2R framework [2] addresses those by allowing to publish mappings across knowledge bases that allow to map classes and defined the transformation of property values. While this framework supports a large number of transformations, it does not allow the automatic discovery of possible transformations. The Linked Data Integration Framework LDIF [21], whose goal is to support the integration of RDF data, builds upon R2R mappings and technologies such as SILK [10] and LDSpider[11]. The concept behind the framework is to enable users to create periodic integration jobs via simple XML configurations. Still these configurations have to be created manually. The same drawback holds for the Semantic Web Pipes[12] [20], which follows the idea of Yahoo Pipes[13] to enable the integration of data in formats such as RDF and XML. By using Semantic Web Pipes, users can efficiently create semantic mashups by using a number of operators (such as getRDF, getXML, etc.) and connecting these manually within a simple interface. KnoFuss [19] addresses data integration from the point of view of link discovery. It begins by detecting URIs that stand for the same real-world entity and either merging them to one or linking them via `owl:sameAs`. In addition, it allows to monitor the interaction between instance and dataset matching (which is similar to ontology matching [7]). Fluid Operations' Information Workbench[14] allows to search through, manipulate and integrate for purposes such as business intelligence. [5] describe a framework for semantic enrichment, ranking and integration of web videos, and [1] present semantic enrichment framework of *Twitter* posts. Finally, [8] tackles the linked data enrichment problem for sensor data via an approach that sees enrichment as a process driven by situations of interest. To the best of our knowledge, the work we presented in this paper is the first generic approach tailored towards learning enrichment pipelines of Linked Data given a set of atomic enrichment functions.

8 Conclusions and Future Work

In this paper, we presented an approach for learning enrichment pipelines based on a refinement operator. To the best of our knowledge, this is the first approach for learning RDF based enrichment pipelines and could open up a new research area. We also presented means to self-configure atomic enrichment pipelines so as to find means to enrich datasets according to examples provided by an end user. We showed that our approach can easily reconstruct manually created enrichment pipelines, especially when given a prototypical example and when faced with regular datasets. Obviously, this does not mean that our approach will always achieve such high F-measures. What our results suggest is primarily that if a human uses an enrichment tool to enrich his/her dataset manually, then our approach can reconstruct the pipeline. This seems to hold even for relatively complex pipelines.

Although we achieved reasonable results in terms of scalability, we plan to further improve time efficiency by parallelising the algorithm on several CPUs as well as load balancing. The framework underlying this study supports directed acyclic graphs as enrichment specifications by allowing to split and merge datasets. In future work, we will thus extend our operator to deal with graphs in addition to sequences. Moreover, we will look at pro-active enrichment strategies as well as active learning.

Acknowledgments *This work were supported by the* GeoKnow *project (GA 318159).*

[11] `http://code.google.com/p/ldspider/`
[12] `http://pipes.deri.org/`
[13] `http://pipes.yahoo.com/pipes/`
[14] `http://www.fluidops.com/information-workbench/`

References

1. F. Abel, Q. Gao, G.-J. Houben, and K. Tao. Semantic enrichment of twitter posts for user profile construction on the social web. In *Proc. of ESWC*, pages 375–389. Springer, 2011.
2. C. Bizer and A. Schultz. The R2R Framework: Publishing and Discovering Mappings on the Web. In *Proceedings of COLD*, 2010.
3. A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Inf. Process. Lett.*, 24(6):377–380, Apr. 1987.
4. L. Buhmann and J. Lehmann. Pattern based knowledge base enrichment. In *12th International Semantic Web Conference, 21-25 October 2013, Sydney, Australia*, 2013.
5. S. Choudhury, J. G. Breslin, and A. Passant. *Enrichment and ranking of the youtube tag space and integration with the linked data cloud*. Springer, 2009.
6. S. Dietze, S. Sanchez-Alonso, H. Ebner, H. Q. Yu, D. Giordano, I. Marenzi, and B. P. Nunes. Interlinking educational resources and the web of data: A survey of challenges and approaches. *Program: electronic library and information systems*, 47(1):60–91, 2013.
7. J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007.
8. S. Hasan, E. Curry, M. Banduk, and S. O'Riain. Toward situation awareness for the semantic sensor web: Complex event processing with dynamic linked data enrichment. *Semantic Sensor Networks*, page 60, 2011.
9. H. H. Hoang, T. N.-P. Cung, D. K. Truong, D. Hwang, and J. J. Jung. Semantic information integration with linked data mashups approaches. *International Journal of Distributed Sensor Networks*, 2014, 2014.
10. R. Isele and C. Bizer. Learning Linkage Rules using Genetic Programming. In *Sixth International Ontology Matching Workshop*, 2011.
11. J. Lehmann and P. Hitzler. Concept learning in description logics using refinement operators. *Machine Learning journal*, 78(1-2):203–250, 2010.
12. J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 2014.
13. V. Lopez, C. Unger, P. Cimiano, and E. Motta. Evaluating question answering over linked data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 21:3–13, 2013.
14. I. Millard, H. Glaser, M. Salvadores, and N. Shadbolt. Consuming multiple linked data sources: Challenges and experiences. In *COLD Workshop*, 2010.
15. A.-C. Ngonga Ngomo. On link discovery using a hybrid approach. *Journal on Data Semantics*, 1:203 – 217, December 2012.
16. A.-C. Ngonga Ngomo, S. Auer, J. Lehmann, and A. Zaveri. Introduction to linked data and its lifecycle on the web. In *Reasoning Web. Semantic Technologies for Intelligent Data Access*, pages 1–90. Springer Berlin Heidelberg, 2014.
17. A.-C. Ngonga Ngomo, N. Heino, K. Lyko, R. Speck, and M. Kaltenböck. SCMS - semantifying content management systems. In *ISWC 2011*, 2011.
18. A.-C. Ngonga Ngomo and K. Lyko. Unsupervised learning of link specifications: deterministic vs. non-deterministic. In *Proceedings of the Ontology Matching Workshop*, 2013.
19. A. Nikolov, V. Uren, E. Motta, and A. Roeck. Overcoming schema heterogeneity between linked semantic repositories to improve coreference resolution. In *Proceedings of the 4th Asian Conference on The Semantic Web*, pages 332–346, 2009.
20. D. L. Phuoc, A. Polleres, M. Hauswirth, G. Tummarello, and C. Morbidoni. Rapid prototyping of semantic mash-ups through semantic web pipes. In *WWW*, pages 581–590, 2009.
21. A. Schultz, A. Matteini, R. Isele, C. Bizer, and C. Becker. LDIF - linked data integration framework. In *COLD*, 2011.
22. R. Speck and A. N. Ngomo. Ensemble learning for named entity recognition. In *Proc. of ISWC (International Semantic Web Conference) 2014*, pages 519–534, 2014.