REPORT ON DELIVERABLE D3.7

Final release of the POI Integration Software

SLIPO

PROJECT NUMBER: 731581
START DATE OF PROJECT: 01/01/2017
DURATION: 36 months

Horizon 2020

SLIPO is a research project funded by European Commission's Horizon 2020 Programme.

The information in this document reflects the author's views and the European Community is not liable for any use that may be made of the information contained therein. The information in this document is provided "as is" without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/ her sole risk and liability.

Dissemination Level	Public
Due Date of Deliverable	Month 36 31/12/2019
Actual Submission Date	20/12/2019
Work Package	WP3 - Data Integration and Quality Assurance
Tasks	Task 3.1 Interlinking, Task 3.2 Enrichment, Task3.3 Fusion
Type	Other
Approval Status	Submitted for approval
Version	1.0
Number of Pages	136
Filename	D3.7_Final_release_of_the_POI_integration_software.pdf

Abstract

This document presents the final versions of the POI integration software components of the SLIPO Toolkit, i.e., LIMES (linking), DEER (enrichment), and FAGI (fusion). For each software component, we first briefly present the state of the art in its respective fields, as well as their interim versions delivered in M18 and presented in the corresponding deliverables D3.1, D3.2, and D.3.3. In the following, for each software component we present their latest versions, detail the improvements introduced, and present evaluation experiments assessing their effectiveness and scalability.

History

version	date	reason	revised by
0.1	28/08/2019	First version	Spiros Athanasiou, Kostas Patroumpas, Nikos Karagiannakis, Mohamed Sherif
0.2	15/10/2019	Updates in all sections; minor changes in structure	Mohamed Sherif, Axel Ngonga, Kevin Dreßler, Simon Scerri, Nikos Karagiannakis, Michail Alexakis, Yannis Kouvaras, Dimitris Skoutas
0.5	28/10/2019	Updates in all sections; revisions in documentation and evaluation	Kostas Patroumpas, Nikos Karagiannakis, Mohamed Sherif
0.6	09/11/2019	Updates in Sections 5,6, 7 and Annexes	Mohamed Sherif, Michail Alexakis, Yannis Kouvaras, Dimitris Skoutas
0.7	02/12/2019	Updates in Sections 6 and 7	Mohamed Sherif, Kevin Dreßler
0.8	04/12/2109	Multiple revisions across all sections	Spiros Athanasiou, Dimitris Skoutas
0.9	17/12/2019	Minor edits in Sections 2,3 and the Annexes; updates in the experimental evaluation	Mohamed Sherif, Kevin Dreßler
1.0	20/12/2019	Final version	Spiros Athanasiou

Author list

organization	name	contact information
ATHENA RC	Spiros Athanasiou	spathan@imis.athena-innovation.gr
ATHENA RC	Kostas Patroumpas	kpatro@imis.athena-innovation.gr
ATHENA RC	Dimitris Skoutas	dskoutas@imis.athena-innovation.gr
ATHENA RC	Yannis Kouvaras	jkouvar@imis.athena-innovation.gr
ATHENA RC	Michail Alexakis	alexakis@imis.athena-innovation.gr
ATHENA RC	Nikos Karagiannakis	nkaragiannakis@imis.athena-innovation.gr
ATHENA RC	Pantelis Chronis	pchronis@imis.athena-innovation.gr
INFAI	Mohamed Sherif	sherif@informatik.uni-leipzig.de

organization	name	contact information
INFAI	Axel Ngonga	ngonga@informatik.uni-leipzig.de
INFAI	Kevin Dreßler	dressler@informatik.uni-leipzig.de
IAIS	Simon Scerri	simon.scerri@iais.fraunhofer.de

Executive Summary

This document presents the final versions of the POI integration software components of the SLIPO Toolkit, i.e., LIMES (linking), DEER (enrichment), and FAGI (fusion).

- **LIMES** implements a series of measures and distance functions for link discovery among POI resources, as well as machine learning algorithms for automatic discovery of links among POI resources. Further, LIMES incorporates mechanisms for interlinking validation and quality statistics/indicators extraction to facilitate and assure the quality of the interlinking process.
- The final version of LIMES (v1.7) extends the interim framework of LIMES (v1.0) presented in Deliverable D3.1. In particular, we optimized the POI-specific (i) pointset distance, (ii) topological relations and (iii) temporal relation discovery algorithms to enable big POI datasets handling, such as the ones provided by our partners in the SLIPO project. All the new optimised approaches are combined with the existing techniques within LIMES and are integrated with the SLIPO Workbench. Moreover, we implemented a simple, yet efficient in our setting, distributed execution scheme-based on FLINK and SPARK. Finally, we implemented a dedicated component in LIMES for extracting statistics and quality indicators from the discovered links.
- **DEER** is our software framework for the enrichment of POI datasets. Considering that a POI knowledge base is simply a set of triples, the goal of the enrichment process is to (a) determine a set of triples $\Delta+$ to be *added* to the source knowledge base and/or (b) determine a set of triples $\Delta-$ to be *deleted* from the source knowledge base. This type of enrichment is dubbed in our framework as the *dereferencing-based enrichment*. In addition to dereferencing-based enrichment, DEER provides other enrichment methods such as *entity recognition* and *schema enrichment*, as well as mechanisms for quality statistics/indicators extraction to facilitate and assure the quality of the POI enrichment process.
- The final version of DEER extends the interim framework of DEER (v1.0.0), presented in Deliverable D3.2. In particular, we extended the enrichment paradigm of DEER to support *multiple* datasets as input and output as well as non-linear enrichment graphs. We optimized our parallel execution engine to enable DEER to handle big POI datasets and the increased complexity of the enrichment graphs. Moreover, we developed an optimized implementation of POI-specific enrichment operators, such as the geo-distance operator. We also implemented an algorithm for automatic learning the of enrichment graphs. Finally, we implemented a dedicated component in DEER v2.0 for extracting statistics and quality indicators for the input datasets as well as the output (enriched) datasets.
- **FAGI** is our software framework for the scalable and quality assured fusion of POI datasets. Fusion consists in receiving two datasets containing POIs and their properties (attributes), as well as a set of links linking POI entities between the two datasets, and producing a third, final dataset, which contains consolidated descriptions of the linked POIs. Each POI entity in the final, fused dataset is described by a set of non-redundant, non-conflicting, complete, properties, that have been derived by merging the initial descriptions for the linked POIs. FAGI implements a series of rules and actions for link validation and fusion, as well as mechanisms for selecting the proper fusion actions in each

case. Further, FAGI incorporates mechanisms for fusion validation and quality statistics/indicators extraction to facilitate and assure the quality of the fusion process.

- The final version of FAGI (v3.0) extends the interim framework of FAGI v2.0 presented in Deliverable D3.3, with several improvements increasing the efficiency and scalability of fusion for POI entities. In summary, we evaluated a series of similarity measures, extended and fine-tuned them, increasing their similarity matching accuracy on the POI datasets at hand. Then, we created a set of predefined fusion rules that are incorporated into FAGI, that serve as a guide for the end users of the platform into constructing new fusion rules. We also implemented a dedicated component for extracting statistics and quality indicators from both the input datasets and the output (fusion) results. To support automatic recommendation of validation and fusion actions, we implemented a learning mechanism for addressing the (a) validation of the pair of linked POIs, deciding whether it should be accepted or rejected, and the (b) selection of the most fitting action from a set of available fusion actions. All the new techniques are integrated within the SLIPO Workbench. Finally, we have implemented a simple, yet efficient in our setting, distributed execution scheme.

The layout of document is as follows.

In Section 1, and for each software component, we briefly describe their objectives in the frame of the SLIPO project, discuss the state of the art and provide some background knowledge. Next, we briefly report our achievements until the end of the project and the delivery of the final versions of all software components.

In Section 2, we present LIMES, our link discovery framework for linking POI datasets. First, we introduce its features and architecture, and present an evaluation of our new linking optimization algorithms.

In Section 3, we present DEER, our POI enrichment framework. First, we introduce its features and architecture, and present an evaluation of our new DEER execution engine.

In Section 4, we present FAGI, our fusion framework. First, we introduce its features and architecture, and present an evaluation of its efficiency and scalability.

In Sections 5, 6, and 7 we present the user guides for LIMES, DEER, and FAGI respectively, including building and installation instructions, configuration settings, and a short demonstration of the usage of the software.

Abbreviations and Acronyms

AEO	Atomic Enrichment Operators
API	Application Programming Interface
CLI	Command-Line Interface
CRS	Coordinate Reference System
FAGI	Fusion and Aggregation of Geospatial Information
KPI	Key Performance Indicator
LD	Link Discovery
LS	Link Specification
ML	Machine Learning
NER	Named Entity Recognition
POI	Point of Interest
RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
UI	User Interface
VM	Virtual Machine
WGS84	World Geodetic System 1984 (EPSG:4326)
XML	Extensible Markup Language

Table of Contents

1. Introduction 14

1.1. POI data interlinking 14

1.1.1. Introduction 14

1.1.2. State of the art 16

1.1.3. Interlinking in the SLIPO lifecycle 18

1.1.4. Achievements - LIMES v1.7 18

1.2. POI data enrichment 20

1.2.1. Introduction 20

1.2.2. State of the art 21

1.2.3. Enrichment in the SLIPO lifecycle 21

1.2.4. Achievements – DEER v2.2.0 22

1.3. POI data fusion 24

1.3.1. Introduction 24

1.3.2. State of the art 25

1.3.3. Fusion in the SLIPO lifecycle 27

1.3.4. Achievements - FAGI v3.0 28

2. The LIMES Framework 30

2.1. Features and Functionality 30

2.1.1. CLI Client 30
2.1.2. LIMES Server 31
2.1.3. Graphical User Interface 31
2.1.4. Limes Web User Interface 32
2.2. Architecture 34
2.2.1. Controller Layer 35
2.2.2. Input Layer 35
2.2.3. Data Layer 36
2.2.4. Execution Layer 36
2.2.5. Machine Learning layer 37
2.2.6. Output Layer 37
2.2.7. Core Modules 37
2.3. Libraries and Frameworks 38
2.4. License 40
2.5. Documentation 40
2.6. Experimental evaluation 40
2.6.1. Big Data Prototypes for LIMES 40
2.6.2. Topological relation discovery using RADON2 43
3. The DEER Framework 48
3.1. Features and Functionality 48
3.1.1. Dataset Readers 48
3.1.2. Dataset Writers 48
3.1.3. Enrichment Operators 49
3.2. Architecture 52

3.2.1. FARADAY-CAGE 53
3.2.2. DEER Plugin Types 53
3.3. Libraries and Frameworks 53
3.4. License 55
3.5. Documentation 55
3.6. Experimental Evaluation 55
4. The FAGI framework 57
4.1. Features and functionality 57
4.1.1. POI fusion 58
4.1.2. POI link Validation 65
4.1.3. Recommending POI link validation and fusion 68
4.1.4. Quality indicators and statistics 69
4.1.5. Tracking changes in FAGI 71
4.1.6. Dataset output modes 72
4.2. Architecture 73
4.2.1. Input and output 73
4.2.2. Core modules 74
4.2.3. Partitioning scheme 76
4.3. Libraries and Frameworks 77
4.4. License 78
4.5. Documentation 78
4.6. Experimental Evaluation 78

4.6.1. Datasets and measures 79
4.6.2. Results 80
5. LIMES Usage Manual 83
5.1. Building Installation 83
5.1.1. Generating Jar File (Headless) 83
5.1.2. Generating Jar File (GUI) 83
5.2. Configuration Settings 84
5.2.1. Metadata 84
5.2.2. Prefixes 84
5.2.3. Data Sources 84
5.2.4. Metric Expression for Similarity Measurement 87
5.2.5. Execution (optional) 93
5.2.6. Machine Learning 94
5.2.7. Granularity (optional) 96
5.2.8. Acceptance Condition 97
5.2.9. Review Condition 97
5.2.10. Output Format 97
5.3. Configuration File Examples 97
5.4. Running LIMES 100
5.4.1. Running LIMES core 100
5.4.2. Running LIMES GUI 100
5.4.3. Running LIMES from Java 100
5.5. LIMES GUI 100

5.5.1. Menu Bar 101
5.5.2. Toolbox 102
5.5.3. Metric Builder 102
5.5.4. Creating New Configuration via GUI 103
5.5.5. Creating a new Link Specification via GUI 105
5.5.6. Link Specification Running 105
5.5.7. Machine Learning via GUI 107
5.5.8. Batch Learning via GUI 108
5.5.9. Unsupervised Learning via GUI 109
6. DEER Usage Manual 110
6.1. Building Installation 110
6.1.1. Generating Jar File (Headless) 110
6.1.2. Import DEER using Maven 110
6.2. Configuring DEER 111
6.2.1. FARADAY-CAGE Core Vocabulary 111
6.2.2. DEER IO Plugins 112
6.2.3. DEER Enrichment Operator Plugins 112
6.3. Running DEER 118
6.4. Extending DEER 118
7. FAGI Usage Manual 120
7.1. Building Installation 120

7.1.1. Command line version 120
7.1.2. Web interface version 120
7.2. Configuration Settings 121
7.2.1. Configuration specification 121
7.2.2. Rule Specification 124
7.3. Demonstration 125
7.3.1. FAGI input configuration 126
7.3.2. FAGI validation and fusion rule specification 127
7.3.3. FAGI execution 131
8. References 132
9. Annex 136
9.1. Datasets characteristics 136
9.1.1. Dataset A 136
9.1.2. Dataset B 136

1. Introduction

In this section, we first discuss the basic concepts and goals of POI data interlinking, enrichment, and fusion. Then, in each field we present the current state of the art, and finally we briefly present our achievements regarding the scalable, effective, and quality-assured interlinking, enrichment, and fusion of POIs.

1.1. POI data interlinking

1.1.1. Introduction

Establishing links between knowledge bases is one of the key steps of the Linked Data publication process[1]. A plethora of approaches has thus been devised to support this process [LDS+17]. In this report, we present the LIMES framework, which was designed to accommodate a large number of link discovery approaches within a single extensible architecture. LIMES was designed as a declarative framework (i.e., a framework that processes link specifications), to address two main challenges.

- **Time-efficiency:** The mere size of existing knowledge bases (e.g., 30+ billion triples in LinkedGeoData [LGDD+12], 20+ billion triples in LinkedTCGA [LCG+13]) makes efficient solutions indispensable to the use of link discovery frameworks in real application scenarios. LIMES addresses this challenge by providing time-efficient approaches based on the characteristics of metric spaces [LIMES+11, GRR+12], orthodromic spaces [ORCHID] and on filter-based paradigms [REWED].
- **Accuracy:** Efficient solutions are of little help if the results they generate are inaccurate. LIMES accommodates solutions that allow the generation of links between knowledge bases with a high accuracy. These solutions abide by paradigms such as batch and active learning [RAVEN, EAGLE, ULLS], unsupervised learning [ULLS] and even positive-only learning [WOMBAT].

1.1.1.1. The Link Discovery Problem

Interlinking, which is generally referred to as **Link Discovery (LD)** in literature, is the process of finding related entities in (not necessarily distinct) knowledge bases. An interlinking task generates a set of mappings among such knowledge bases as its output.

Formally, given two sets S, T of source and target resources, respectively, and a relation R, the goal of interlinking is to find the set $M \subseteq S \times T$ of pairs $(s, t) \in S \times T$ such that $R(s, t)$. Note that, S and T are two not necessarily distinct sets of instances. One way to automate this discovery is to compare the $s \in S$ and $t \in T$ based on their properties using a (in general complex) similarity metric. Two entities are then considered to be linked via R if their similarity is superior to a threshold θ. If R is `owl:sameAs`, then we are faced with a deduplication task. The direct computation of the pairs for which R holds is commonly very tedious if at all possible. Thus, most frameworks for LD resort to approximating the set of pairs for which R holds by using Link Specifications (LS). A LS can be regarded as a classifier C_{ij} that maps each element of the knowledge bases' cartesian product $K_i \times K_j$ to one of the classes of $Y = \{+1, -1\}$, where K_i is called the set of source

[1] https://www.w3.org/DesignIssues/LinkedData.html

instances while K_j is the set of target instances. $(s, t) \in K_i \times K_j$ is considered by C_{ij} to be a potential link when $C_{ij}(s, t) = +1$. Otherwise, (s, t) is considered not to be a potential link.

The formal specification of LD adopted herein is akin to that proposed in [OLDHA]. Given two (not necessarily distinct) sets S, T of source and target resources respectively, as well as a relation R, the goal of LD is to find the set $M = \{(s,t) \in S \times T : R(s, t)\}$ of pairs $(s, t) \in S \times T$ such that $R(s, t)$. In most cases, computing M is a non-trivial task. Hence, a large number of frameworks (e.g., SILK [SILK], LIMES [LIMES] and KnoFuss [Knofuss]) aim to approximate M by computing the **mapping** $M' = \{(s,t) \in S \times T : \sigma(s,t) \geq \theta\}$, where σ is a similarity function and θ is a similarity threshold. For example, one can configure these frameworks to compare the dates of birth, family names and given names of persons across census records to determine whether they are duplicates. We call the equation which specifies M' a **link specification** (short **LS**; also called linkage rule in the literature, see e.g., [SILK]). Note that the LD problem can be expressed equivalently using distances instead of similarities as follows: Given two sets S and T of resources, a (complex) distance measure δ and a distance threshold $\tau \in [0, \bullet[$, determine $M' = \{(s,t) \in S \times T : \delta(s,t) \leq \tau\}$. Note that a distance function δ can always be transformed into a normed similarity function σ by setting $\sigma(x, y) = (1+\delta(x, y))-1$. Hence, the distance threshold τ can be transformed into a similarity threshold θ by means of the equation $\theta = (1 + \tau)^{-1}$. Consequently, the concepts of distance and similarity are used interchangeably within this document.

Under this so-called **declarative paradigm**, two entities s and t are then considered to be linked via R if $\sigma(s, t) \geq \theta$. Naive algorithms require $O(|S||T|) \in O(n^2)$ computations to output M'. Given the large size of existing POI knowledge bases, time-efficient approaches able to reduce this runtime are hence a central component of LIMES. In addition, note that the choice of appropriate σ and θ is central to ensure that M is approximated correctly by M'. Approaches that allow the computation of accurate σ and θ are thus fundamental for the LIMES framework [LDS+17]).

1.1.1.2. Formal Overview

Several approaches can be chosen when aiming to define the syntax and semantics of LSs in detail [ORCHID, SILK, Knofuss]. In LIMES, we chose a grammar with a syntax and semantics based on set semantics. This grammar assumes that LSs consist of two types of atomic components:

- **similarity measures** *m*, which allow the comparison of property values or portions of the concise bound description of two resources and
- **operators** *op*, which can be used to combine these similarities into more complex specifications.

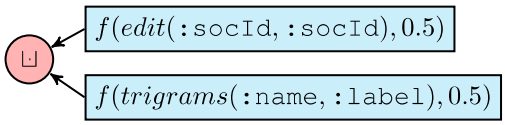
Similarity measures could be *atomic* or *complex*. Without loss of generality, an **atomic similarity measure** is function that maps an input pair of resources (sets S, T of source and target resources respectively) into a similarity score between zero and one inclusive. Formally, we define an *atomic similarity measure a* as a function $a : S \times T \rightarrow [0,1]$. An example of an atomic similarity measure is the *edit* similarity. We define the edit similarity of two strings str_1 and str_2 as $(1 + lev(str_1, str_2))^{-1}$, where *lev* stands for the *Levenshtein* distance.). **A complex measure** m combines measures m_1 and m_2 using measure operators such as *min* and *max*. We use mappings $M \subseteq S \times T$ to store the results of the application of a similarity measure to $S \times T$ or subsets thereof. We denote the set of all mappings as $\mathbb{M}$.

We define a filter f as a function $f(m, \theta)$, where m is a measure (either simple of complex) and θ is the filter threshold. The output of the filter f is the subset of mappings M (generated from the application of the

measure *m*) with similarity values above or equal the filter threshold θ. We call a specification *atomic* when it consists of exactly one filter function. For example, the upper part of Figure 1 represent the atomic link specification *f(edit(:socId, :socId), 0.5)*. For such atomic LS, the filter *f* will generate the subset of mappings between source and target resources with social security numbers' *edit distance* less than of equal *0.5*.

A *complex specification* can be obtained by combining two specifications L_1 and L_2 through an operator that allows the merging of the results of L_1 and L_2. Here, we use the operators: *mappings-union* $\sqcap$, *mappings-intersect* $\sqcup$ and *mappings-difference* \ as they are complete w.r.t. the Boolean algebra and frequently used to define LSs. We denote the set of all LSs as $\mathbb{L}$.

An example of a complex LS is given in Figure 1. The result of such complex LS would be the *mappings-union* ($\sqcup$) of the mappings generated by applying the two atomic LSs *f(edit(:socId, :socId), 0.5) and f(trigrams(:name, :label), 0.5).*



LS	$[[LS]]_M$
$f(m, \theta)$	$\{(s,t) \mid (s,t) \in M \wedge m(s,t) \geq \theta\}$
$L_1 \sqcap L_2$	$\{(s,t) \mid (s,t) \in [[L_1]]_M \wedge (s,t) \in [[L_2]]_M\}$
$L_1 \sqcup L_2$	$\{(s,t) \mid (s,t) \in [[L_1]]_M \vee (s,t) \in [[L_2]]_M\}$
$L_1 \backslash L_2$	$\{(s,t) \mid (s,t) \in [[L_1]]_M \wedge (s,t) \notin [[L_2]]_M\}$

Figure 1: Example of a complex LS. The filter nodes are rectangles while the operator nodes are circles. :socID stands for social security number.

1.1.2. State of the art

There has been a significant amount of work pertaining to the two key challenges of LD: **efficiency** and **accuracy**. Here, we focus on frameworks for link discovery. Dedicated approaches and algorithms can be found in the related work sections of [LIMES+11, GRR+12, OLDHA, EAGLE, ULLS, 1 COALA, AEGLE, EEJW, RADON]. As described in [LDS+17], most LD frameworks address the **efficiency challenge** by aiming to discover unnecessary computations of the similarity or distance measures efficiently. The accuracy challenge is commonly addressed using machine learning techniques.

One of the first LD frameworks is **SILK** [SILK], which uses a multi-dimensional blocking technique (MultiBlock [SILK]) to optimize the linking runtime through a rough index pre-matching. To parallelize the linking process, SILK relies on *MapReduce*. Like LIMES, SILK configuration supports input files in XML and is also able to retrieve RDF data by querying SPARQL endpoints. Both frameworks also allow user-specified link types between resources as well as *owl:sameAs* links. SILK incorporates element-level matchers on selected properties using string, numeric, temporal and geo-spatial similarity measures. SILK also supports multiple matchers, as it allows the comparison of different properties between resources that are combined together

using match rules. To this end, SILK implements supervised and active learning methods for identifying LS for linking.

KNOFUSS [Knofuss] incorporates blocking approaches derived from databases. Like LIMES, it supports unsupervised machine learning techniques based on genetic programming. However, in contrast to both aforementioned tools, KNOFUSS supports no other link types apart from `owl:sameAs` and implements only string similarity measures between matching entities properties. Additionally, its indexing technique for time complexity minimization is not guaranteed to achieve result completeness.

ZHISHI.LINKS [ZHISHI.LINKS] is another LD framework that achieves efficiency by using an index-based technique that comes at the cost of effectiveness. Similar to KNOFUSS, it only supports `owl:sameAs` links but implements geo-spatial relations between resources. In comparison with all previous three tools, it supports both property-based and semantic-based matching, using knowledge obtained by an ontology.

SERIMI [SERIMI] is a LD tool that retrieves entities for linking by querying SPARQL endpoints. In contrast to the previous frameworks, it only supports single properties to be used for linking resources. However, it incorporates an adaptive technique that weighs differently the properties of a knowledgebase and chooses the most representative property for linking.

SLINT+ [SLINT+] is a LD tool that is very similar to SERIMI but supports comparisons between multiple properties. Finally, there are a set of frameworks (**RIMOM** [RIMoM], **AGREEMENTMAKER** [AGREM], **LOGMAP** [LOGMAP], **CODI** [CODI]) that initially supported ontology matching and then evolved to support LD between resources.

RIMOM is based on Bayesian decision matching in order to link ontologies and transforms the problem of linking into a decision problem. Even though RIMOM utilizes dictionaries for ontology matching, it does not support them in entity linking.

Similar to RIMOM, **AGREEMENTMAKER** also uses dictionaries in ontology level. AGREEMENTMAKER incorporates a variety of matching methods based on different properties considered for comparison and the different granularity of the components.

LOGMAP is an efficient tool for ontology matching that scales up to tens (even hundreds) of thousands of different classes included in an ontology. It propagates the information obtained from ontology matching to link resources, by using logical reasoning to exclude links between resources that do not abide by the restrictions obtained from ontology matching.

CODI is a probabilistic-logical framework for ontology matching based on the syntax and semantics of *Markovlogic*. It incorporates typical matching approaches that are joined together to increase the quality of ontology alignment.

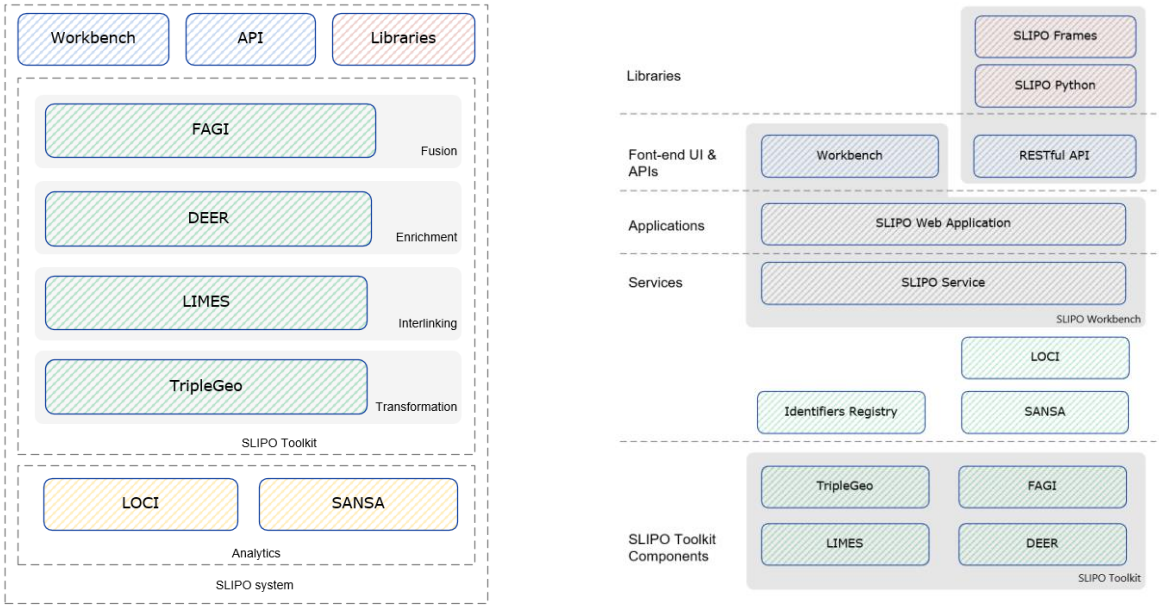
The basic difference between the previous LD frameworks and **LIMES** is that LIMES provides both theoretical and practical guarantees for efficient and scalable LD. LIMES is guaranteed to lead to exactly the same matching as a brute force approach while at the same time reducing significantly the number of comparisons. The approaches incorporated in LIMES facilitate different approximation techniques to compute estimates of the similarity between instances. These estimates are then used to filter out a large number of those instance pairs that do not suffice the mapping conditions. By these means, LIMES can reduce the number of comparisons needed during the mapping process by several orders of magnitude.

LIMES supports the first planning technique for link discovery, HELIOS, that minimizes the overall execution of a link specification, without any loss of completeness. As shown in our previous works (see [HELIOS, AEGLE, UERMTG]), LIMES is one of the most scalable link discovery frameworks currently available.

1.1.3. Interlinking in the SLIPO lifecycle

The SLIPO POI integration lifecycle is realized through the SLIPO Workbench, a platform for defining, executing and managing POI integration workflows (*see Deliverable D1.4 "Final SLIPO System"*). These workflows combine all the components of the SLIPO Toolkit, supporting the integrated execution of all four core POI integration steps: *transformation, interlinking, enrichment and fusion*. Additionally, the SLIPO system prescribes a set of value-added services on top of integrated POI datasets.

LIMES, the interlinking framework of SLIPO. LIMES incorporates many algorithms for performing efficient interlinking among POI resources. In the context of SLIPO, LIMES receives as input two RDF POI datasets conforming to the SLIPO ontology. Thus, LIMES's input POI data are first transformed by *TripleGeo*, into the proper RDF format and schema. Further, apart from the two input POI datasets, LIMES requires as input a configuration file containing the LIMES configuration parameters. LIMES's output consists in a single file, which contains the mapping (i.e. *links*) between POI entities from both input POI datasets. The output of LIMES is essential for running the next SLIPO tools (i.e., *FAGI* and *DEER*). Figure 2 shows an overview of the SLIPO architecture.



(a) High level architecture (b) Logical architecture

Figure 2: SLIPO Architecture

1.1.4. Achievements - LIMES v1.7

LIMES v1.7 is the final version of LIMES that has been developed in the context of the SLIPO project and focuses on POI-specific interlinking. One of the major goals of the project is to abstract as much complexity as possible from the end users of the SLIPO Workbench. So, in order to keep user interaction at a minimum

and requiring no knowledge of Linked Data technologies and concepts, we aimed at adapting and fine-tuning LIMES's functionality specifically for POI data, as well as at automating the interlinking process as much as possible. To this end, we emphasized on the development of the backend of the platform, aiming to enrich and specialize the core linking functionality of the framework. Next, we enumerate the new features and functionality of LIMES, as a result of our work during the project.

- **POI-specific point-set distance.** New point-set distances based on the vector representations of the POI resources (e.g., Hausdorff, mean, surjection and sumOfMin). Altogether, we implemented a set of 10 point-set distance functions based on our survey published on [PSDM].
- **Topological relation discovery** based on the vector representations of the POI resources (e.g. one POI resource contains/crosses/touches another POI resource). For instance, find all the parking locations within shopping malls. Our novel algorithm, RADON [RADON], for rapid discovery of topological relation among POI resources with 2D geometries. Our evaluation [RANDON2] prove that RADON is able to overperform state-of-the-art approaches up to 3 orders of magnitude while maintaining a precision and a recall of one.
- **Temporal relation discovery** based on the temporal timestamps within the POI resources (e.g., one POI take place after/before/during another POI). For example, a specific area is used as a parking location only during a football match. We were able to propose AEGLE [AEGLE], a novel approach for the efficient computation of links between POIs' temporal representations according to Allen's interval algebra. Our evaluations of the runtime of AEGLE show that AEGLE outperforms the state of the art by up to 4 orders of magnitude while maintaining a precision and a recall of one.
- **Combining the new techniques with the one already in LIMES.** In LIMES v1.7 we integrated the novel algorithms for the 10 POI-point-set distances as well as RADON and AEGLE into the LIMES core. In particular, a new mapper is implemented for each of the new relation types. Such mappers are combined with the already existing mappers for efficient link discovery of the new types of relation specific for POI resources.
- **Integration with the SLIPO Workbench**. LIMES v1.7 realizes two deployment modes: (a) standalone, as an individual software that accepts as input linked POI datasets and provides as output a mapping file contains the links between the input POI datasets; (b) deployment within the SLIPO Workbench, where LIMES serves as an integral component of the *SLIPO Toolkit* and is loosely integrated by the SLIPO Workbench with the other software components into forming POI integration workflows.
- **Scalability**. In LIMES v1.7 we have implemented a simple, yet efficient in our setting, distributed execution scheme, that functions independently of core interlinking of LIMES. Specifically, we have two implementations based on Apache Spark and Apache Flink frameworks.
- **Quality-based Linking**. We implemented a dedicated component in LIMES v1.7 for extracting **statistics** and **quality indicators** from the discovered links. These involve qualitative as well as quantitative statistics/quality indicators on the individual POIs within the input datasets, as well as the discovered links. For example, qualitative quality indicators include precision, recall and F-measure of the generated mapping from source to target datasets. On the other side, the number of generated links is an example of a quantitative quality measure.

- The purpose of this component is twofold: (a) present statistics and indicators to the end user that will assist him/her to *assess the quality, potential issues and gaps* in the input datasets; select the most *fitting link specification*; potentially re-configure and re-execute more appropriate linking processes based on improved link specifications, and (b) to be utilized within the rules of the similarity functions, and as features in the learning algorithms, in order to improve the accuracy of the automatic linking algorithms.

A detailed presentation of LIMES is provided in Section **Error! Reference source not found.**.

1.2. POI data enrichment

1.2.1. Introduction

Dataset enrichment is the process of **adding or deleting** some triples to/from some resources of such dataset in. order to produce an **enhanced** version of the input dataset. Enrichment is one of the main parts of the data integration process. In the frame of the SLIPO project, Enrichment focuses on Point of Interest (POI) entities, that are characterized by a set of major properties (*name, coordinates, category*), as well as potentially several additional properties (*address, telephone, email, rating, etc.*). Enrichment considers one or more input dataset(s) containing POIs. The goal of *enrichment* is to produce one or more *enriched* dataset(s), which contains better descriptions of the input POIs. That is, each POI entity in the final, enriched dataset must be described by a set of RDF triples, that have been derived by merging the initial description for the POI with those generated via various enrichment operations. Note that, some enrichment approaches can define a set of triples to be removed from the original POI descriptions. Those removed sets of triples are either wrong, or inaccurate. The enrichment process can also replace inaccurate triples with ones with correct values.

1.2.1.1. Formal Overview

Considering that a POI knowledge base is simply a set of triples, the goal of an enrichment process is to (a) determine a set of triples $\Delta+$ to be *added* the source knowledge base and/or (b) determine a set of triples $\Delta-$ to be *deleted* from the source knowledge base. Any other enrichment process can be defined in terms of $\Delta+$ and $\Delta-$, e.g., altering triples can be represented as combination of addition and deletion.

Formally, Let $\mathbb{K}$ be the set of all RDF knowledge bases. Let $K \in \mathbb{K}$ be a finite RDF knowledge base. K can be regarded as a set of triples $(s, p, o) \in (\mathbb{R} \cup \mathbb{B}) \times \mathbb{P} \times (\mathbb{R} \cup \mathbb{L} \cup \mathbb{B})$, where $\mathbb{R}$ is the set of all resources, B is the set of all blank nodes, P the set of all predicates and L the set of all literals. Given a knowledge base K, the idea behind POI knowledge base enrichment is to find an enrichment plan $E: K \rightarrow K$ that maps K to an enriched knowledge base K' with $K' = E(K)$. We define E as a directed acyclic graph (DAG) of atomic enrichment operators (AEO) $e \in E$, where E is the set of all atomic enrichment operators. $2^{\mathbb{E}}$ is used to denote the power set of $\mathbb{E}$, i.e., the set of all enrichment DAGs.

1.2.2. State of the art

POI Enrichment is an important topic for all applications that rely on a large number of POI knowledge bases and necessitate a unified view on this data, e.g., Question Answering frameworks [SILK], Linked Education [ORCHID] and all forms of semantic mashups [EAGLE].

In recent work, several challenges and requirements data consumption and integration have been pointed out [Knofuss]. For example, the **R2R framework** [LGDD+12] addresses those by allowing to publish mappings across knowledge bases that allow to map classes and define 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*** [RADON], whose goal is to support the integration of RDF data, builds upon R2R mappings and technologies such as *SILK* [SILK] and *LDSpider*[2]. 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. **Semantic Web Pipes**[3] [MKTI] follows the idea of **Yahoo Pipe**[4] 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** [Knofuss] 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 [REWED]). **Fluid Operations' Information Workbench**[5] allows to search through, manipulate and integrate for purposes such as business intelligence. The work in [GRR+12] describes a framework for semantic enrichment, ranking and integration of web videos, and [LDS+17] presents a semantic enrichment framework for *Twitter* posts. Finally, [RAVEN] tackles the linked data enrichment problem for sensor data via an approach that sees enrichment as a process driven by situations of interest.

1.2.3. Enrichment in the SLIPO lifecycle

The SLIPO POI integration lifecycle is realized through the SLIPO Workbench, a platform for defining, executing and managing POI integration workflows (*see Deliverable D1.4 "Final SLIPO System"*). These workflows combine all the components of the SLIPO Toolkit, supporting the integrated execution of all four core POI integration steps: *transformation, interlinking, enrichment and fusion*. Additionally, the SLIPO system prescribes a set of value-added services on top of integrated POI datasets (Figure 2(a)).

DEER, the POI enrichment framework of SLIPO. DEER incorporates many approaches for performing efficient enrichment among POI resources. In the context of SLIPO, DEER receives as input one or more RDF POI dataset(s) conforming to the SLIPO ontology. DEER's input POI data are first transformed by *TripleGeo*, into the proper RDF format and schema. Moreover, DEER input datasets may be linked via *LIMES* prior to be enriched by DEER. Further, DEER requires as input a configuration file containing the DEER configuration

[2] http://code.google.com/p/ldspider/

[3] http://pipes.deri.org

[4] http://pipes.yahoo.com

[5] http://www.fluidops.com/information-workbench/

parameters. DEER's output consists in one or more files. The output file(s) contain the enriched versions of the input datasets.

1.2.4. Achievements – DEER v2.2.0

DEER v2.2.0 is the final version of DEER that has been developed in the context of the SLIPO project and focuses on POI-specific enrichment. One of the major goals of the project is to abstract as much complexity as possible from the end users of the SLIPO Workbench. So, in order to keep user interaction at a minimum and requiring no knowledge of Linked Data technologies and concepts, we aimed at adapting and fine-tuning DEER's functionality specifically for POI data, as well as at automating the enrichment process as much as possible. To this end, we emphasized on the development of the backend of the platform, aiming to enrich and specialize the core enrichment functionality of the framework. Next, we enumerate the new features and functionality of DEER, as a result of our work during the entire SLIPO project.

- **Multiple input/output datasets.** We extended the enrichment paradigm of DEER to support *multiple* datasets as input and output. The new DEER enrichment paradigm with such multiple input/output datasets is in the form of DAGs, where the input/output datasets act as the access points of the DEER execution engine.

Reading multiple datasets enables DEER to manipulate more data coming from multiple datasets. This is of main importance when dealing with complex data such POI datasets. Also, the current version of DEER is able to read input data from multiple sources, including local and remote files, as well as querying POIs in the form of RDF data from SPARQL endpoints.

Writing multiple output datasets enables DEER to generate multiple enriched dataset versions using the same set of input datasets. For example, consider the case of having a dataset of POIs representing mobility data, where the user wants to use such POI dataset in two different scenarios; one for tourism and one for geo-marketing.

- **Non-linear enrichment graphs.** DEER v0.1 was only able to provide linear enrichment pipelines. During the SLIPO project, we implemented the *first non-linear enrichment pipelines execution engine* in DEER v1.0, in order to deal with the complexity of POI datasets. The non-linear enrichment pipelines allow DEER to enable new enrichment operations in ways which were not possible before, such as: *merge, clone, split and fusion* of the POI datasets.

The new non-linear enrichment pipelines together with the multiple input/output capabilities enable us to implement our automatic parallelization approach, which we will introduce in the next section.

- **Automatic parallelization approach.** In the current version of DEER, we implemented the new *parallel execution engine* for the DEER execution graph. Our parallelization approach applies *graph colouring* to the DEER execution graph to find the parts of the graph that could be executed in parallel. Moreover, our parallelization approach ensures a balanced load through all the processing units and in the same time minimum usage of system resources.
- We extended our initial implementation of the parallel execution engine currently available in DEER v2.2 (*dubbed FARADAY-CAGE*). The new version of the FARADAY-CAGE features the automatic

validation of plugin parameters using SHACL[6] as well as an API to restructure the execution graph. This is also used by our learning algorithm, which allows for rich tooling, such as caching of previous computation steps, in order to increase productivity and decrease waiting time for users of DEER.

We also parallelize most of DEER's enrichment operators internally in order to achieve the maximum utilization of the system where DEER runs. In addition, we optimize many of our enrichment operators for better performance. For instance, the *dereferencing enrichment operator* groups related resources together in order to generate much less number SPARQL queries to dereference the data of interest.

- **POI-specific enrichment operators.** in DEER v2.2 we provide a set of novel POI-specific enrichment operators, for example:
    - *The geo-fusion enrichment operator (see Section 3.1.3.9)* enables DEER to fuse POI data coming from multiple datasets. For example, using the geo-fusion operator DEER is able to fuse multiple geometric representations of a specific POI resources (e.g., resort, roads or city districts) to keep the most accurate one.
    - *The geo-distance enrichment operator (see Section 3.1.3.10)* enables DEER to is to enrich POI resources with the distance to the other POI resources of a certain type. For example, using the geo-distance enrichment operator, DEER will be able to enrich a hotel dataset with the distances to all restaurants or gas stations.
    - *The geo-locator operator* enables DEER to is to enrich POI resources by converting either a resource address to a co-ordinate or vice versa.
- **Automatic learning of enrichment graphs.** The machine learning algorithm introduced within DEER v0.1 for its automatic configuration was based on a refinement operator. The idea of that algorithm was to enable novice users to define adequate RDF dataset enrichment workflows. As highly modular applications in general require a lot of manual configuration of their components and therefore presume expert knowledge to precisely define how the modules operate and interact with each other. The learning algorithm within DEER v0.1 was only able to learn pipeline on enrichment operators as this was the only paradigm DEER v0.1 was able to handle.

Since introducing DAG-shaped RDF dataset enrichment workflows in DEER v1.0, the complexity of the learning problem is greatly increased in comparison to DEER v0.1. Therefore, we base our learning approach within DEER v2.2 on Genetic Programming (GP) instead of refinement operators, since GP is known for its ability to find good solutions for hard symbolic regression problems, albeit at the cost of being non-deterministic. Our new machine learning approach aims to assure the quality of the enriched datasets by monitoring the F-measure of the enriched dataset as our quality indicators.

- **Optimize the performance of the POI-enrichment operators.** With DEER v2.2 we optimize our POI-operators to scale to millions of POI resources. Also, we implemented a self-configurator for the geo-distance operator. Moreover, in DEER v2.2 we also implemented more novel POI-specific enrichment operators such as the *geo-locator enrichment operator* to enrichment of POI resources with geo-coordinates and/or address information.

[6] https://www.w3.org/TR/shacl/#dfn-shacl-superclass

- **Quality based enrichment**. We implemented a dedicated component in DEER v2.2 for extracting **statistics** and **quality indicators** for the input datasets as well as the output (enriched) datasets. These involve statistics on the POIs within the input datasets, enrichment operators and the enriched datasets, such as the number of enriched resources and each enrichment operator run time.
- The purpose of this component is twofold: (a) present statistics and indicators to the end user that will assist him/her to *assess the quality, potential issues and gaps* in the input datasets; select the most *fitting configuration for each enrichment operator*; potentially re-configure and re-execute more appropriate enrichment processes based on improved configuration, and (b) to be utilized within the learning algorithms, in order to improve the accuracy of the automatic enrichment process.

1.3. POI data fusion

1.3.1. Introduction

1.3.1.1. Fusion task definition

Fusion is the process of **merging** the descriptions (*attributes, metadata, properties*) of two or more resources that correspond to the same real-world entity, to produce a **richer, cleaner and universal description for the respective entity**.

Fusion constitutes the final part of the data integration process, which usually consists of three steps: schema integration (mapping), duplicate detection (interlinking) and fusion [BN08]. In the frame of the SLIPO project, fusion focuses on Point of Interest (POI) entities, that are characterized by a set of major properties (*name, coordinates, category*), as well as potentially several additional properties (*address, telephone, email, rating, etc.*). Fusion considers two input datasets containing POIs and their properties and a set of links that connect POI entities between the two datasets. Mapping between the properties of the linked entities and interlinking of the POI entities between the two datasets are performed in previous steps of the SLIPO integration workflow (*see TripleGeo [SLIPOD22] and LIMES [SLIPOD31] software respectively*). The goal of fusion is to produce a final dataset, which contains consolidated descriptions of the linked POIs. That is, each POI entity in the final, fused dataset must be described by a set of **non-redundant, non-conflicting, complete properties, that have been derived by merging the initial descriptions for the POI**. Additionally, considering the big picture of the POI integration lifecycle (Figure 1), the fusion process is tightly interconnected with validation and quality assurance. To this end, the fusion process needs to incorporate several mechanisms to assess the quality of the proposed fusion actions and their results.

1.3.1.2. Fusion challenges

The fusion process starts at the point where a set of pairs of linked POI entities is created, and consequently, each pair carries along several duplicate descriptions (values) regarding each of their properties. For example, a pair of linked POIs potentially comprises two name properties, two coordinate fields and two

addresses. The values of these pairs of matching properties might be exactly equal; in that case, fusing the values of the individual properties is straightforward. However, this is usually not the case; the values of the matching properties differ due to various reasons (*e.g., formatting, spelling errors, measurement-curation-validation errors, multiple valid attributes for the same real-world characteristic*) and to various extents (*from a simple syntax error in a name, to completely different phone numbers*).

Understandably, this setting may significantly hinder the qualitative assessment of both the initial POI data, and the fusion results. For example, if two linked POI entities, with very high *geospatial proximity*, have moderately different names, should they be validated as the same real-world POI, or should their link be rejected as wrong? In this case, the decision usually depends on the semantics of the POIs (*e.g., which POI categories they belong to*), on the semantics of specific terms that comprise their names (*e.g., is the POI a neighborhood of a city or a gas station?*), on the combination of similarities on several individual matching properties of the POIs, etc. On the other hand, POI descriptions are often *incomplete*, lacking even major property values, such as address street or number, phone and even POI category. In this case, the user has even less information available to decide whether and how two POIs should be fused. In another example, the user/system may be very confident that the pair of POIs correspond to the same real-world POI (*e.g., most of their matching properties have the same or very close value*). However, only the names of the two POIs are moderately different. In that case, the user has several options as to how to fuse the names of the POI: *select one of the two values*; *keep both values in separate properties*; *merge (i.e., concatenate) the two values and keep them in the same property*; *keep the most recent value*; *keep the value from the most trustworthy dataset*, etc. Again, deciding the most proper option differs from case to case and depends on several factors.

It becomes apparent that several Key Performance Indicators (KPIs) of the POI integration lifecycle directly depend on the effective application of fusion mechanisms: Accuracy of POI location; Accuracy of POI categories; Completeness; Timeliness; Lack of conflicting POIs and/or properties (*see D1.1 "Use Cases and Requirements", Sections 2.1.3.15-2.1.3.18 [SLIPOD11]*). One of the most important contributions of fusion is increasing the quality of a dataset, with respect to increasing the accuracy of values and handling conflicting values. Additionally, increasing the completeness and improving the timeliness of the data are important goals towards increasing the value of POI datasets. Thus, fusion comprises a crucial step of the integration process that increases the value of POI data by consolidating them and improving their quality.

1.3.2. State of the art

Traditionally, geospatial fusion (or conflation) consists in merging different geospatial datasets of overlapping regions, in such way that the best quality elements of individual datasets are kept in the final dataset [CK08]. Reviewing the considerable amount of approaches, some standardized techniques have emerged, which typically operate in three main steps. *Feature matching* aims at finding a set of conjugate point pairs (control points) in two datasets which most probably correspond to each other and can be used as reference points. These are usually identified based on strict geospatial matching. Then, *match checking* increases the accuracy of control points by filtering out pairs with ambiguous quality. This can also utilize non-spatial metadata of objects to produce set of more accurate control points. Finally, *spatial attribute alignment* uses the control points to align the rest of the geospatial objects in both datasets by space partitioning and transformation techniques. Such methods utilize a combination of spatial and non-spatial

criteria in matching/scoring functions [Saa88, CCM+98], statistics on geospatial properties/relations [WF99, CSK+06], or iteration [Saa88].

In the last years, the focus of geospatial data fusion tasks has shifted from traditional geometry conflation to **non-geospatial (*thematic*) metadata fusion**. This can be partially attributed to the advancements in mapping and geocoding that resulted in the production of more accurate geospatial data (*in terms of POI geometries and coordinates*). On top of that, being able to provide a set of very accurate thematic metadata for POIs became the number-one priority for commercial applications. For example, it is currently more important to provide a correct and as specific as possible *category* for a POI (*e.g., "Mexican fast food court", instead of just "restaurant"*) or up-to-date information (*e.g., a cafeteria that has closed and a cocktail bar has opened at the exact same spot*), rather than provide the precise geometry of a POI. If we also consider the rise of crowdsourcing, social networks, and the adoption of Linked Data practices, we find ourselves in an era where geospatial information is accompanied by *a wealth of non-spatial metadata* (*e.g., textual descriptions, categories/tags, reviews, ratings*). Thus, previous fusion approaches might be considered to a large extent as *obsolete*, at least when considering their objective in the context of POI integration: the major goal of fusion is not so much to align maps, but rather to decide on which *fusion strategies* (*e.g., keep both, keep largest, keep more recent*) to be used in order to merge the properties that characterize a pair of linked POIs. In this context, several fusion frameworks have been recently developed to address the challenge of Linked Data fusion.

Sieve [MMB12] focuses on quality assessment and fusion of Linked Data, being part of a larger framework for Linked Data integration [SMI+12] that provides several techniques for data fusion. It considers factors such as timeliness of data, provenance, as well as user-configurable preference lists on features of the dataset. The fusion process is defined through XML configuration files, where the user can specify: (a) the classes of the objects to be considered for fusion, (b) the properties to be fused, and (c) for each property, the fusion function to be applied. Sieve supports the following functions (strategies):

- Filter: removes all values for which the input quality assessment metric is below a given threshold.
- KeepFirst: keeps the value with the highest value for a given quality assessment metric. In case of ties, the function keeps the first in order of input.
- KeepAllValuesByQualityScore: similar to KeepFirst, but in case of ties, it keeps all values with the highest score.
- Average: takes the average of all input data for a given property.
- Voting: picks the value that appeared most frequently across sources. Each named graph has one vote, the most voted value is chosen.
- WeightedVoting: picks the value that appeared most frequently across highly rated sources. Each named graph has one vote proportional to its score for a given quality metric, the value with highest aggregated scores is chosen.

ODCleanStore [MK+12] is another framework that supports linking, cleaning, transformation and quality assessment operations on Linked Data. The fusion component supports several user configurable fusion strategies, which also consider provenance and quality metadata:

- ANY, MIN, MAX, SHORTEST, LONGEST. An arbitrary value, minimum, maximum, shortest, or longest is selected from the conflicting values V

- AVG, MEDIAN, CONCAT. Computes the average, median, or concatenation of conflicting values
- BEST. The value with the highest aggregate quality is selected
- LATEST. The value with the newest time is selected
- ALL. All input values are preserved

WInte.r [LBB17] is a recently released framework for end-to-end data integration. The framework implements well-known methods for data pre-processing, schema matching, identity resolution, data fusion, and result evaluation. Regarding fusion, the system expects as input datasets in a consolidated schema and correspondences (links) between their records. From these correspondences, groups of records which represent the same entity are collected for each attribute. Such an entity/attribute group then contains all values for this combination from the input datasets. To decide for a final value, a *conflict resolution function* (fusion action) is applied to the values. A selected *Data Fusion Strategy* defines the conflict resolution function and an *evaluation rule* for each attribute. These functions determine how a final value is chosen from multiple possible values and how it is evaluated. The supported options for fusion are:

- Voting: Applies majority voting to the values.
- ClusteredVote: Clusters all values using the provided similarity measure and returns the centroid of the largest resulting cluster.
- Intersection: Creates the intersection of all values (applicable if values are sets).
- IntersectionKSources: Creates a set of all values that are included in at least k input values (applicable if values are sets).
- Union: Creates the union of all values (applicable if values are sets).
- FavourSources: Returns the value from the source with the highest score (as defined by the user).
- MostRecent: Returns the value from the source that is most recent (as defined by the user).
- Average: Returns the average of all values (numeric).
- Median: Returns the median of all values (numeric).
- LongestString: Returns the longest value by character count (strings).
- ShortestString: Returns the shortest value by character count (strings).

FAGI v1.0 [GVK+15], the first version of our fusion software, was *the only available framework dedicated on fusion of geospatial Linked Data*. It supports an extended set of fusion actions regarding both geometries and thematic properties of the entities, as well as batch fusion actions and a basic link discovery functionality. Further, through its interactive map-based web interface, it supports authoring and quality assessment, allowing the collation of vector geometries of linked entities with the underlying map layers and the manual adjustment of geometries.

1.3.3. Fusion in the SLIPO lifecycle

The SLIPO POI integration lifecycle is realized through the SLIPO Workbench, a platform for defining, executing and managing POI integration workflows (*see Deliverable D1.4 "Final SLIPO System"*). These workflows combine all components of the SLIPO Toolkit, supporting the integrated execution of all four core POI integration steps: *transformation, interlinking, enrichment and fusion*. Additionally, the SLIPO system prescribes a set of value-added services on top of integrated POI datasets (Figure 2(a)).

FAGI implements an extended functionality for performing automated fusion on the properties of linked POI entities, as well as quality indicators extraction and link validation. In the context of SLIPO, FAGI receives as input RDF POI datasets conforming to the SLIPO ontology. Thus, FAGI's input POI data are first transformed by TripleGeo, into the proper RDF format and schema. Further, apart from two input POI datasets, FAGI requires as input a file containing the links between the POIs of the two datasets, which is produced by LIMES. FAGI's main output consists in a single file, which contains consolidated POI entities from both input POI datasets.

1.3.4. Achievements - FAGI v3.0

FAGI v3.0 is the final version of FAGI developed in the context of the SLIPO project and focuses on POI-specific fusion and validation. One of the major goals of the project is to abstract as much complexity as possible from the end users of the SLIPO Workbench. So, in order to keep user interaction at a minimum and require no knowledge of Linked Data technologies and concepts, we aimed at adapting and fine-tuning FAGI's functionality specifically for POI data, as well as at automating the fusion process as much as possible. To this end, we emphasized on the development of the backend of the platform, aiming to enrich and specialize the core fusion functionality of the framework. Next, we enumerate the new features and functionality of FAGI, achieved during the project.

- **Specialization on POI data**. Our work was driven by the major commercial POI datasets that were available in the project. We explored and analyzed these datasets individually, but more importantly, against each other, with respect to the types of POIs they contain, and their naming conventions. Additionally, we explored several other properties of the POI entities (*name, address, phone, website, email*), examining the differences their values present in pairs of linked POIs. Finally, we collaborated with the industrial partners (*producers, owners or users of the commercial datasets*) in gathering training labels (*annotations*) on fusion and validation actions.
- Based on this analysis, we evaluated a series of **similarity measures**, extended and fine-tuned them, increasing their similarity matching accuracy on the POI datasets at hand. In parallel, utilizing the same similarity functions, we developed POI-specific **rule mechanisms** for deciding fusion actions, taking into account conditions on the property values of the linked POIs. On top of the developed functions, conditions, rules and actions, we have implemented a simple mechanism for constructing validation and fusion **rule specifications**, that allow the consideration of several factors for deciding a fusion action for a pair of matching properties or a validation action for a pair of linked POIs.
- Finally, we created a first set of **predefined fusion rules** that are incorporated into FAGI and, apart from the core fusion functionality they offer, they serve as a guide for the end users of the platform into constructing new fusion rules, depending on the context, the datasets and the application scenario.
- **Quality-based fusion**. We implemented a dedicated component in FAGI v3.0 for extracting **statistics** and **quality indicators** from both the input datasets and the output (fusion) results. These involve statistics on the individual POIs within the input linked datasets, as well as statistics regarding the pairs of linked POIs. For example, they include distribution of specific POI properties, % of empty/non-empty property values, measures on how (dis)similar the values of matching properties for linked POIs are, etc.

- The purpose of this component is twofold: (a) present statistics and indicators to the end user that will assist her to *assess the quality, potential issues and gaps* in the input datasets; select the most *fitting fusion actions* per case; potentially re-configure and re-execute more appropriate fusion processes based on improved fusion specifications, and (b) to be utilized within the rules of the similarity functions, and as features in the learning algorithms, in order to improve the accuracy of the automatic fusion and validation action recommendation mechanisms.
- **Learning mechanisms for fusion and validation action recommendation**. To support automatic recommendation of validation and fusion actions, we implemented an initial learning mechanism for addressing the two major tasks we consider, as classification problems: (a) **validation** of the pair of linked POIs, deciding whether it should be accepted or rejected; (b) selection of the most **fitting action** from a set of available fusion actions. The current implementation works auxiliary to the validation and rule specification mechanism, by identifying and annotating in the fused datasets ambiguous POI links and fusion results. Our method exploits an extended set of training features that we defined and implemented specifically for the task of POI fusion and validation actions classification/recommendation.
- **Integration with the SLIPO Workbench**. FAGI v3.0 realizes two deployment modes: (a) standalone, as an individual software that accepts as input linked POI datasets and provides as output a fused POI dataset; (b) deployment within the SLIPO Workbench, where FAGI serves as an integral component of the SLIPO toolkit, and is loosely integrated by the SLIPO Workbench with the other software components into forming POI integration workflows.
- **Scalability**. In FAGI v3.0 we have implemented a simple, yet efficient in our setting, distributed execution scheme, that functions independently of the core fusion and validation functionality of FAGI. Specifically, we have implemented a component that partitions the two input POI datasets into subsets, based on subsets of the input links between POIs. Then, for each subset of the input datasets, a separate instance of FAGI is deployed. Finally, the output of all FAGI instance is merged in a final output containing all fused POI entities.

A detailed presentation of FAGI is provided in Section 4.

2. The LIMES Framework

LIMES, the **Li**nk Discovery Framework for **Me**tric **S**paces, is a framework for discovering links between entities contained in Linked Data sources. LIMES is a hybrid framework that combines the mathematical characteristics of metric spaces as well prefix-, suffix- and position filtering to compute pessimistic approximations of the similarity of instances. These approximations are then used to filter out a large amount of those instance pairs that do not suffice the mapping conditions. By these means, LIMES can reduce the number of comparisons needed during the mapping process by several orders of magnitude and complexity without losing a single link.

2.1. Features and Functionality

LIMES can be run in four modes: Command Line Interface (CLI) client, LIMES server and Graphical User Interface (GUI) client and our novel web interface. All modes offer the same functionality but over different interfaces to support a variety of use cases. In the fooling sections we introduce each mode.

2.1.1. CLI Client

For this purpose, the user should simply run

```
java -jar path_to_limes.jar config.xml [OPTIONS...]
```

The following optional command line flags and options are available:

- -f $format sets the format of configuration file. Possible values for $format are"XML" (default) or "RDF"
- -d $file_path configures the path for the statistics JSON output file
- -1 enforces 1-to-1 mappings, i.e. for each source resource only keep the link with the highest probability
- -g $file_path configures a reference mapping file (a.k.a. gold standard) to compute precision, recall and f measure
- -F $format sets the format of the gold standard. Possible values for $format are"csv" (default), "tab" or "rdf"". Only effective when -g $file_path is also specified
- -s runs the LIMES server
- -p $port used to specify port of LIMES server, defaults to port 8080
- -l $limit limits the number of resources processed by LIMES server to $limit, defaults to -1 (no limit). *CAUTION: Setting this option will compromise the correctness of LIMES and is only encouraged to reduce server load for demo purposes. *
- -h prints out a help message

- -o $file_path sets the path of the logging file

In case your system runs out of memory, please use the -Xmx option (must appear before the -jar option) to allocate more memory to the Java Virtual Machine.

2.1.2. LIMES Server

LIMES can be run as an HTTP Server, implementing a RESTful API and serving a browser frontend by default. Configuration files are accepted via POST multipart/form-data uploads. Each configuration file gets assigned a unique *job_id*. Given this job id, the user can query the server for the status of the job, its logs, a list of result files and the contents of these result files.

The following RESTful operations are currently implemented:

- submit/ **(POST)** --- used to upload configuration files as multipart/form-data POST messages and returns the assigned *job_id* in a JSON object. **Accepts XML Configuration file** (See example below)
- status/:id **(GET)** --- returns the status (a numerical code) for a given job in a JSON object. The following statuses are currently implemented:
    - **-1 (Unknown)** - a configuration file for the given *job_id* has not been found on the server
    - **0 (Scheduled)** - the configuration file is present and the job is waiting for execution
    - **1 (Running)** - the job is currently running
    - **2 (Finished)** - the job is finished and its output files are ready for delivery
- logs/:id **(GET)** --- returns the java logs for the given job. Useful for troubleshooting.
- list/measures **(GET)** --- returns the list of available measures.
- list/operators **(GET)** --- returns the list of available operators.
- list/preprocessings **(GET)** --- returns the list of available preprocessings.
- sparql/:urlEncodedEndpointUrl **(GET, POST, OPTIONS)** --- proxy for SPARQL queries. Useful in browser when SPARQL endpoints do not implement CORS headers.
- upload **(POST)** --- used to upload source and/or target files as multipart/form-data POST messages and returns the

assigned *upload_id* in a JSON object.

- uploads/:uploadId/sparql **(GET)** --- query uploaded files. Useful in browser.
- results/:id **(GET)** --- returns a list of result files in a JSON object.
- result/:id/:filename **(GET)** --- returns the contents of a given result file for a given job id.

2.1.3. Graphical User Interface

In addition to supporting configurations as input files, LIMES provides a graphical user interface (GUI) to assist the end user during the LD process [GUI+16]. The framework supports the manual creation of LS as shown in Figure 3 for users who already know which LS they would like to execute to achieve a certain

linking task. However, most users do not know which LS suits their linking task best and therefore need help throughout this process. Hence, the GUI provides wizards which ease the LS creation process and allow configuration of the machine learning algorithms. After the selection of an algorithm, the user can modify the default configurations if necessary and run it. If a batch or unsupervised learning approach is selected, the execution is carried out once and the results are presented to the user. In the case of active learning, the LIMES GUI presents the most highly informative link candidates to the user, who must now label these candidates as either a match or non-match (see Figure 3). This procedure is repeated until the user is satisfied with the result. In all cases, the GUI allows the exportation of the configurations generated by machine learning approaches for future use.

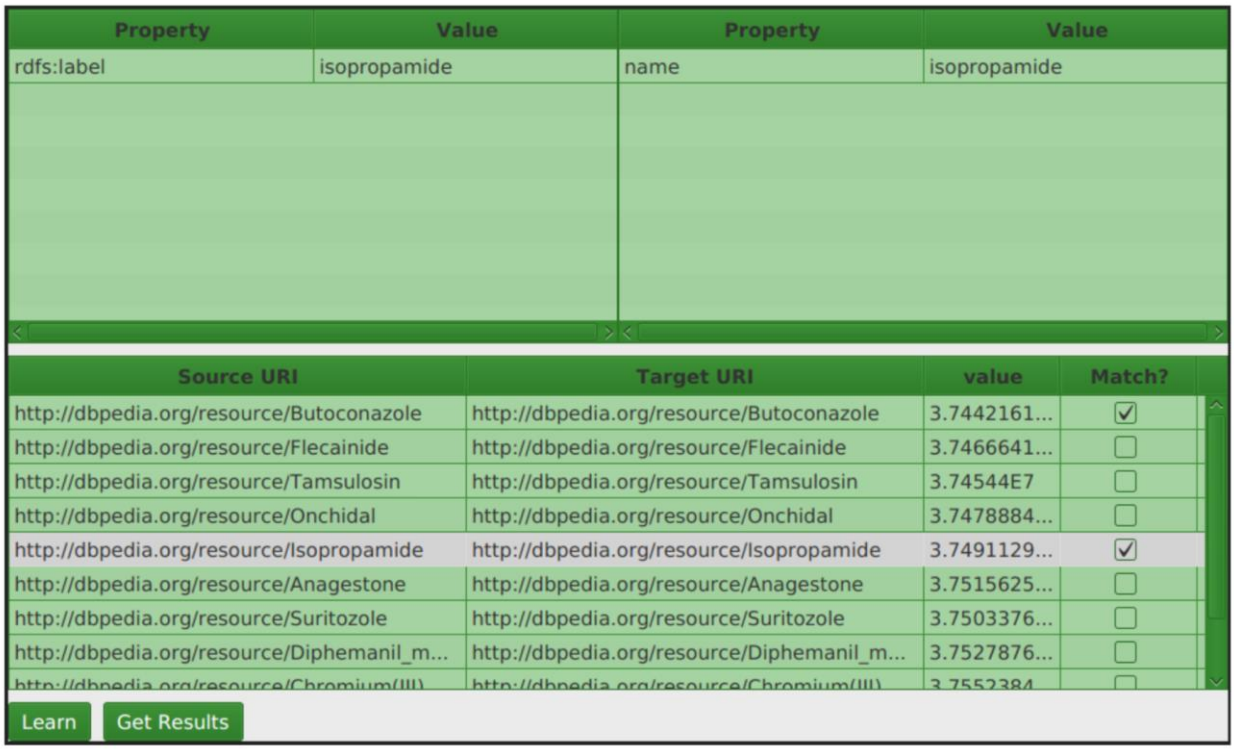


Figure 3: User feedback interface for active learning

2.1.4. Limes Web User Interface

Starting from LIMES v.1.7.0, we provide our novel web user interface for LIMES, called **LimesWebUI**[7]. The aim of LimesWebUI is to aid our users throughout the configuration as well as the execution process. LimesWebUI consists of the following main components:

I. **The prefixes component** consists of the set of name spaces to be used through the rest of the Limes configuration process. In most cases, our interface is able to automatically find the common prefixes[8]5. In case the user wants to add a custom prefix, (s)he still can type the prefix manually.

[7] Publicly accessible at http://limes.aksw.org

[8] From https://prefix.cc/context

II. In the **data source and target components**, the user can define the source and target sets of resources to be linked. In particular, the Endpoint field provide a list of common endpoints, where the user can select the one that provide the datasets (s)he interested on. Still, the user can manually input other endpoints if not in the provided list. Then, in the Restriction field the user can select the class within the dataset to retrieve its instances for linking. Note that, our web UI is able to retrieve all the classes automatically for the user via a SPARQL query.

III. **The manual metric component**. Once the user chooses the source/target datasets' endpoints and classes, LimesWebUI will automatically load the respective proper- ties within the source/target instances. The user can either use our interface either to build a manual LS or to configure one of our machine learning algorithms to learn it. For building the manual metric, our interface provides a workspace that uses a custom version of the Blockly API[9], where the user can simply drag and drop the LS elements from the toolbox. Using our work-space, the user can de- fine complex LS which consists of multiple measures, operators and preprocessing functions. Figure 4 shows an example of a complex LS in our work-space. Note that, our interface is able to save/load the work-space for later use.

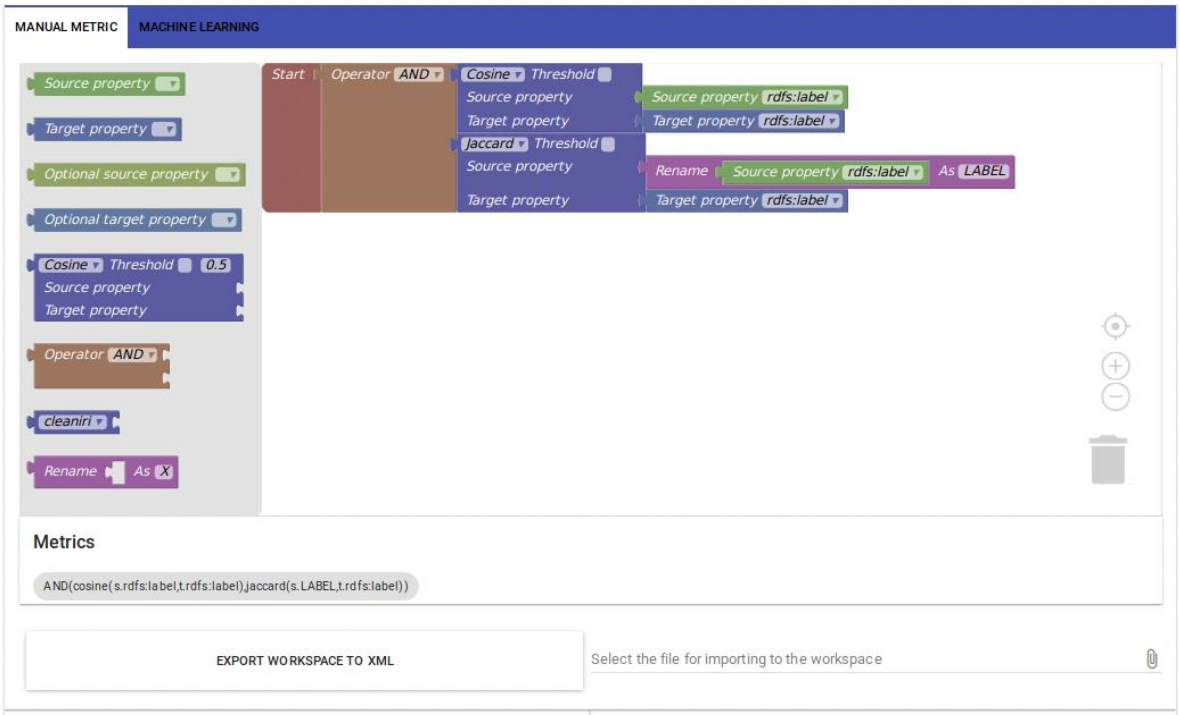


Figure 4: LIMES manual metric example.

IV. **The machine learning (ML) component** consists of: (1) The ML algorithm name to be used. e.g., Wombat [WOMBAT] and Eagle [LIMES+11]. (2) The type of the ML algorithm. i.e., supervised batch, supervised active or unsupervised. (3) A list of parameters for fine tuning the currently selected ML algorithm. Note that, the list of the default parameters of the currently selected ML algorithm will be loaded initially by our web UI. Figure 5 shows an example of using LimesWebUI for configuring the unsupervised version of the Wombat-simple ML algorithm.

[9] https://developers.google.com/blockly

Figure 5: Using LimesWebUI for configuring the machine learning algorithm Wombat.

V. In **the acceptance and review components**, the user can define the acceptance and review thresholds. i.e., the similarity threshold by which a link should be considered by Limes as accepted or to-be-reviewed link. Hence, Limes save such a link into either the accepted/review file.

VI. Using **the output component**, the user can choose an output serialization such as turtle, n-triples, tab separated values and comma separated values.

Finally, the user is able to display the generated XML configuration file, save or even run it. LimesWebUI assigns a unique execution ID (EID) for each linking task. In case an execution takes time, the user can simply close his browser and check for the status of his task later using his/her EID. Once an execution is done, the resulted accepted and to-be-reviewed links are stored in our server, where the user can retrieve them at any time using the respective EID.

2.2. Architecture

As shown in Figure 6, the LIMES framework consists of 6 main layers, each of which can be extended to accommodate new or improved functionality. The input to the framework is a *configuration file*, which describes how the sets S and T are to be retrieved from two knowledge bases K_1 and K_2 (e.g., remote SPARQL endpoints). Moreover, the configuration file describes how links are to be computed. To this end, the user can choose to either provide a LS explicitly or a configuration for a particular machine learning algorithm. The result of the framework is a (set of) mapping(s). In the following, we give an overview of the inner workings of each of the layers.

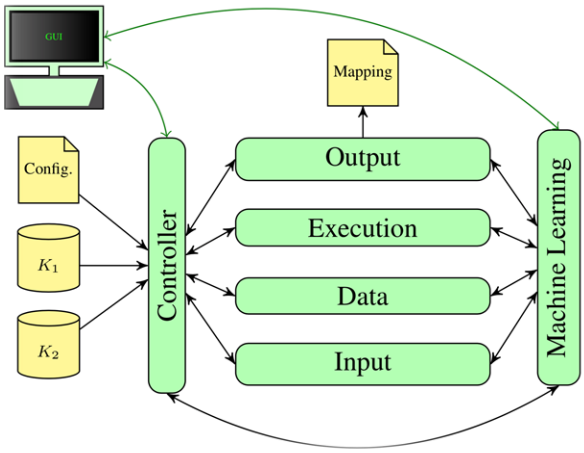


Figure 6: General architecture of LIMES.

2.2.1. Controller Layer

The processing of the configuration file by the different layers is orchestrated by the *controller layer*. The controller instantiates the different implementations of input modules (e.g., reading data from files or from SPARQL endpoints), the data modules (e.g., file cache, in-memory), the execution modules (e.g., planners, number of processors for parallel processing) and the output modules (e.g., the serialization format and the output location) according to the configuration file or using default values. Once the layers have been instantiated, the configuration is forwarded to the input layer.

2.2.2. Input Layer

The input layer reads the configuration and extracts all the information necessary to execute the specification or the machine learning approach chosen by the user. This information includes (1) the location of the input knowledge bases K1 and K2 (e.g., SPARQL endpoints or files), (2) the specification of the sets S and T, (3) the measures and thresholds or the machine learning approach to use. The current version of LIMES supports RDF configuration files based on the LIMES Configuration Ontology (LCO)[10] (see Figure 7) and XML configuration files based on the LIMES Specification Language (LSL) [GUI+16]. If the configuration file is valid (w.r.t. the LCO or LSL), the input layer then calls its *query module*. This module uses the configuration for S and T to retrieve instances and properties from the input knowledge bases that adhere to the restrictions specified in the configuration file. All data retrieved is then forwarded to the data layer via the controller.

[10] https://github.com/dice-group/LIMES/blob/master/limes-core/resources/lco.owl

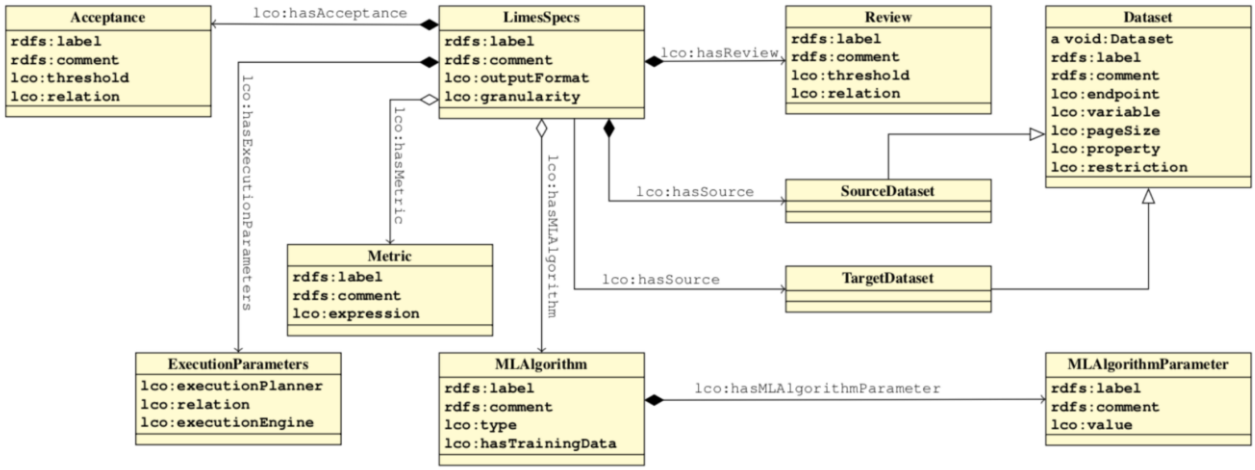


Figure 7: LCO ontology

2.2.3. Data Layer

The *data layer* stores the input data gathered by the input layer by using memory, file or hybrid storage techniques. Currently, LIMES relies on a *hybrid cache* as default storage implementation. This implementation generates a hash for any set of resources specified in a configuration file and serializes this set into a file. Whenever faced with a new specification, the cache first determines whether the data required for the computation is already available locally by using the hash aforementioned. If the data is available, it is retrieved from the file into memory. In other cases, the hybrid cache retrieves the data as specified, generates a hash and caches it on the hard drive. By these means, the data layer addresses the practical problem of data availability, especially from remote data sources. Once all data is available, the controller layer chooses (based on the configuration file) between execution of complex measures specified by the user of running a machine learning algorithm.

2.2.4. Execution Layer

If the user specifies the LS to execute manually, the LIMES controller layer calls the *execution layer*. The execution layer then re-writes, plans and executes the LS specified by the user. The *rewriter* aims to simplify complex LS by removing potential redundancy so as to eventually speed up the overall execution. To this end, it exploits the subset relations and the Boolean algebra which underpin the syntax and semantics of complex specifications. The *planner module* maps a LS to an equivalent execution plan, which is a sequence of atomic execution operations from which a mapping results. These operations include `EXECUTE` (running a particular atomic similarity measure, e.g., the trigram similarity) and operations on mappings such as UNION (union of two mappings) and DIFF (difference of two mappings). For an atomic LS, i.e., a LS such that σ is an atomic similarity measure (e.g., Levenshtein similarity, qGrams similarity, etc.), the planner module generates a *simple plan* with one `EXECUTE` instruction. For a complex LS, the planner module determines the order in which atomic LS should be executed as well as the sequence in which intermediary results should be processed. For example, the planner module may decide to first run some atomic LS L1 and then filter the results using another atomic LS L2 instead of running both L1 and L2 independently and merge the

results. The *execution engine* is then responsible for executing the plan of an input LS and returns the set of potential links as a mapping.

2.2.5. Machine Learning layer

In case the user opts for using a machine learning approach, the LIMES controller calls the *machine learning layer*. This layer instantiates the machine learning algorithm selected by the user and executes it in tandem with the execution layer. To this end, the machine learning approaches begin by retrieving the training data provided by the user if required. The approaches then generate LSs, which are run in the execution layer. The resulting map- pings are evaluated using quality measures (e.g., F-measure, pseudo-F-measure [ULLS]) and returned to the user once a termination condition has been fulfilled. Currently, LIMES implements the EAGLE [EAGLE], COALA [COALA], EUCLID [EUCLID] and WOMBAT [WOMBAT] algorithms. A brief overview of these approaches is given in Section 4. More details on the algorithms can be found in the corresponding papers.

2.2.6. Output Layer

The *output layer* serializes the output of LIMES. Currently, it supports the serialization into files in any RDF serialization also chosen by the user. In addition, the frame- work support CSV and XML as output formats.

2.2.7. Core Modules

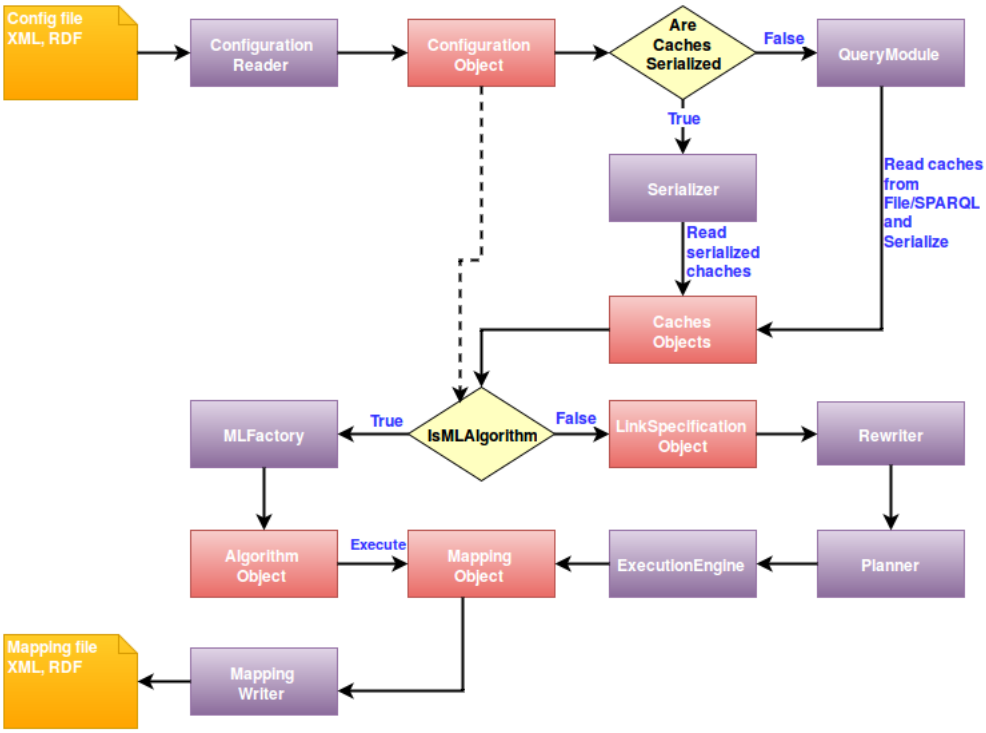


Figure 8: Main modules of LIMES

As shown in Figure 8, the LIMES framework consists of eight main modules of which each can be extended to accommodate new or improved functionality. The central module of LIMES is the **controller** module,

which coordinates the matching process. The matching process is carried out as follows: First, the **controller** calls the **configuration** module, which reads the configuration file and extracts all the information necessary to carry out the comparison of instances, including the URL of the SPARQL-endpoints of source (S) and the target (T) knowledge bases, the restrictions on the instances to map (e.g., their type), the expression of the metric to be used and the threshold to be used.

Given that the configuration file is valid w.r.t. the **LIMES Specification Language (LSL)**, the **query** module is called. This module uses the configuration for the target and source knowledge bases to retrieve instances and properties from the SPARQL-endpoints of the source and target knowledge bases that adhere to the restrictions specified in the configuration file. The query module writes its output into a file by invoking the **cache** module. Once all instances have been stored in the cache, the controller chooses between performing Link Discovery or Machine Learning. For Link Discovery, LIMES will re-write, plan and execute the Link Specification (LS) included in the configuration file, by calling the **rewriter**, **planner** and **engine** modules respectively. The main goal of LD is to identify the set of links (mapping) that satisfy the conditions opposed by the input LS. For Machine Learning, LIMES calls the **machine learning** algorithm included in the configuration file, to identify an appropriate LS to link S and T. Then it proceeds in executing the LS. For both tasks, the mapping will be stored in the output file chosen by the user in the configuration file. The results are finally stored into an RDF or a XML file.

The advantages of LIMES are manifold. First, it implements **highly time-optimized** mappers, making it a complexity class faster than other Link Discovery Frameworks. Thus, the larger the problem, the faster LIMES is w.r.t. other Link Discovery Frameworks. Secondly, LIMES supports a large set of string, numeric, topological and temporal similarity metrics, that provide the user with the opportunity to perform various comparisons between resources. In addition, **LIMES is guaranteed to lead to exactly the same matching as a brute force approach while at the same time reducing significantly the number of comparisons**. In addition, LIMES supports a **large number of input and output formats** and can be extended very easily to fit new algorithms, new datatypes, and new pre-processing functions, thanks to its modular architecture.

In general, LIMES can be used to set links between two data sources, e.g., a novel data source created by a data publisher and existing data source such as *DBpedia*. This functionality can also be used to detect duplicates within one data source for knowledge curation. The only requirement to carry out these tasks is a simple XML-based or TURTLE-based configuration file.

2.3. Libraries and Frameworks

LIMES has dependencies to the following open-source tools/libraries:

- Apache Jena[11]: A Java framework for building Semantic Web applications.
- Java Topology Suite[12]: An API of 2D spatial predicates and functions conforming to the OGC Simple Features Specification for SQL.

[11] https://jena.apache.org/

[12] https://github.com/locationtech/jts

- Google Guava[13]: A set of libraries that includes collection types (such as multimap and multiset), immutable collections, a graph library, functional types, an in-memory cache, and APIs/utilities for concurrency, I/O, hashing, primitives, reflection and string processing.
- Apache Commons Text[14]: A library focused on algorithms working on strings.
- Apache Commons Lang[15]: Provides a host of helper utilities for the java.lang API, notably String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and System properties. Additionally it contains basic enhancements to java.util.Date and a series of utilities dedicated to help with building methods, such as hashCode, toString and equals.
- Google code json-simple[16]: A simple Java toolkit for JSON. It is used to encode or decode JSON text.

Other libraries include:

- algorithms.edjoin
- commons-fileupload
- commons-lang
- com.googlecode.lanterna
- com.ibm.icu
- com.vividsolutions
- eu.medsea.mimeutil
- fr.ign.cogit
- jgraphx
- junit
- net.sf.ehcache
- net.sf.jgap
- nz.ac.waikato.cms.weka
- org.aksw.jena-sparql-api
- org.apache.logging.log4j
- org.apache.maven.plugins
- org.fusesource.jansi
- uk.ac.shef.wit

[13] https://github.com/google/guava

[14] https://commons.apache.org/proper/commons-text/

[15] https://commons.apache.org/proper/commons-lang/

[16] https://code.google.com/archive/p/json-simple/

2.4. License

LIMES is an open source software and is available from GitHub[17] under the terms of the GNU AFFERO GENERAL PUBLIC LICENSE[18].

2.5. Documentation

For a detailed documentation of LIMES, one can consult the following resources:

- LIMES web GUI: http://limes.aksw.org/
- LIMES website: http://aksw.org/Projects/LIMES.html
- User manual: https://dice-group.github.io/LIMES/

2.6. Experimental evaluation

In this Section, we present the experiments that assess the efficiency and scalability of the latest version of LIMES. In our experiments, we evaluated two of our new implemented technologies in LIMES: (1) Our big data prototype using FLINK and SPARK as well as (2) the topological relation discovery using RADON2.

2.6.1. Big Data Prototypes for LIMES

Within the scope of SLIPO, we aim to achieve time-efficient interlinking of large-scale POI datasets, which consist of 10 Million POIs and more. Our interlinking framework LIMES is designed to run on a single server and therefore only supports vertical scaling. However, vertical scaling is not sufficient to handle the huge amount of POIs that we aim to support in SLIPO. Therefore, we needed to investigate how to pair horizontal scaling technologies with our geospatial interlinking algorithms.

To this end, we chose two popular big data frameworks, **Apache SPARK** and **Apache FLINK**, which are based on the Map-Reduce pattern and can automatically distribute work horizontally between multiple workers that run in parallel on multiple computing nodes. Moreover, both of these frameworks extend the classical Map-Reduce paradigm with the ability to persist intermediary results. This feature enables more sophisticated algorithms, like our geospatial interlinking algorithms, to efficiently make use of the massive parallel computation capabilities of server clusters.

2.6.1.1. Evaluation

In order to determine the best system for our algorithms, we developed two prototypes of the Big-Data LIMES, for which we implemented our geospatial interlinking algorithm on top of (1) Apache SPARK and (2) Apache FLINK, respectively.

For our evaluation of the two prototypes, we made sure to implement our algorithm in the most efficient way for each system. In order to measure the run0time scalability of the two prototypes, we generated a

[17] https://github.com/dice-group/LIMES/blob/master

[18] https://github.com/dice-group/LIMES/blob/master/LICENSE

large-scale synthetic POI data benchmark. Moreover, we deployed both prototypes in the same setup. We used two computing nodes in a cluster where each node was assigned 90 CPU cores and 160GB RAM. We also include the result for the single server version of LIMES, in order to see the impact of the horizontal scaling approach using the big data frameworks. Each experiment was repeated ten times and we report the mean run time in seconds. The resulting data, as displayed in Table 1, suggests that the advantages of the parallel execution starts to outweigh the I/O overhead at about 1 Million POIs. At 10 Million POIs our big data prototypes with SPARK and FLINK sped up the computation of the interlinking compared to the single sever LIMES by **6 and 10 times**, respectively.

Table 1: LIMES scalability evaluation of SPARK and FLINK (Runtime in seconds).

Dataset Size	LIMES + SPARK	LIMES + FLINK	Single Server LIMES
10^5	20	67	4
10^6	31	67	70
10^7	191	113	1217

Although the evaluation data suggests that our FLINK prototype scales slightly better than our SPARK prototype, we chose to deploy the SPARK prototype within the SLIPO Workbench. This was done as (1) the suggested speed up of FLINK over SPARK is not substantial and (2) within the SLIPO architecture, we already have a SPARK cluster deployed for SANSA. Therefore, using the SPARK prototype induced lower deployment and maintainability costs of our technology stack.

2.6.1.1.1. Setup of the SPARK Prototype

In order to set up the SPARK prototype in a SPARK cluster, the HR3-SPARK implementation can be accessed at https://github.com/dice-group/LIMES/tree/feature/hr3-SPARK. After compilation using mvn package shade:shade --Dmaven.test.skip=true the prototype evaluation can be submitted to SPARK in the usual way using the following three mandatory arguments:

1. configurationUrl - where to find the configuration (normal filesystem link)
2. evaluationUrl - where to store the evaluation result (HADOOP link)
3. outputUrl - where to store the generated links (HADOOP link)

2.6.1.1.2. Setup of the FLINK Prototype

The setup of the FLINK Prototype was more complicated, since we did not have a FLINK cluster readily available in the SLIPO architecture. In order to set up the FLINK prototype, we therefore first needed to first set up a FLINK cluster. The code for the cluster setup can be found at https://github.com/dobraczka/docker-swarm-FLINK. The implementation of our prototype will be available at https://github.com/dice-group/LIMES/tree/feature/HR3-FLINK .

- Step 1: Setting up a swarm. Following the standard tutorial[19] a swarm is initiated on the master node using:

```
docker swarm init
```

Other nodes can join the swarm given the command that is displayed after successfully initiating the swarm

- Step 2: Creating a network. Using the command:

```
docker network create -d overlay --attachable --scope=swarm --subnet 172.17.6.0/24
FLINK
```

An overlay network is created over the whole swarm. The network is named FLINK. The subnet address has to be adjusted if another cluster would be used.

- Step 3: Setting up a HADOOP docker service. Deploy the HADOOP stack using

```
docker stack deploy -c docker-compose-HADOOP-swarm.yml HADOOP
```

- Step 4: Deploy the FLINK services

As explicated in the notes of the repository:

"FLINK has problems with dockers routing mesh, therefore you have to use --endpoint-mode dnsrr in the script and because you cannot do this with stack deploy this deploy script is necessary in the first place."

Use the deploy script with:

```
./deploy.sh --jmport 6123 --jmheap 200000 --tmheap 200000 --slots 60 --javaopts
"-XX:+UseG1GC" up
```

Adjusting the parameters if needed. The script is used to set the appropriate parameters in the FLINK/conf/FLINK-conf.yaml of the containers. The java options are necessary because there were problems using the default garbage collection.

The services can be stopped using:

```
./deploy.sh down
```

- **Step 5: Putting the data into HADOOP.** Use this command to copy a file to the HADOOP namenode container:

```
docker cp somefile $(docker ps --filter name=HADOOP_namenode --format={{.ID}}):/
```

Enter the container with:

```
docker exec -it $(docker ps --filter name=HADOOP_namenode --format={{.ID}})
/bin/bash
```

Add the file to HADOOP:

```
HADOOP fs -put somefile /
```

[19] https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/

- **(Optional): Distributing data to FLINK containers.** Files should be put into HADOOP if possible, but if needed files can be distributed to all FLINK containers comfortably using this bash command:

```
for i in $(docker ps --filter name=FLINK --format={{.ID}}); do
    docker cp somefile $i:pathdestination
done
```

- **Step 6: Running the application.** The build jar of LIMES with the appropriate main class is needed:

```
docker exec -it $(docker ps --filter name=FLINK-master --format={{.ID}}) FLINK
run LimesFLINK.jar --input mocking.ttl
```

2.6.2. Topological relation discovery using RADON2

In this Section, we present the experiments that assess the efficiency and scalability RADON2, our new topological relation extraction algorithm in LIMES. In our experiments, we evaluated the running time of RADON2 in LIMES, as it is the primary concern in the context of the SLIPO project. First, we briefly present the POI datasets used in the evaluation. Next, we present our extension of the RADON [RADON] algorithm (dubbed RADON2) for topological relation discovery. Finally, we present the OAEI results, which show that RADON2 outperforms the other state of the art in most of the cases.

2.6.2.1. Datasets

RADON2 has been evaluated on the *Ontology Alignment Evaluation Initiative* (OAEI2018)[20] in the Link Discovery Track Task 2 (Spatial). The **Spatial Benchmark** generator **(Task 2)** can be used to evaluate the performance of systems that deal with topological relations proposed in the state of the art DE-9IM (Dimensionally Extended nine-Intersection Model) model [LCG+13]. This benchmark generator implements all topological relations of DE-9IM between *trajectories* in the two-dimensional space. To the best of our knowledge, such a generic benchmark that takes as input trajectories and checks the performance of linking systems for spatial data, does not exist. For the design, we focused on (a) on the correct implementation of all the topological relations of the DE-9IM topological model and (b) on producing large datasets large enough to stress the systems under test. The supported relations are: *Equals, Disjoint, Touches, Contains/Within, Covers/CoveredBy, Intersects, Crosses, Overlaps.*

Task 2 consists of two subtasks:

- Task 2.1: TomTom dataset
  - 2.1.1 : Match `LineStrings` to `LineStrings`
  - 2.1.2 : Match `LineStrings` to `Polygons`
- Task 2.2: Spaten dataset
  - 2.2.1 : Match `LineStrings` to `LineStrings`
  - 2.2.2 : Match `LineStrings` to `Polygons`

[20] http://oaei.ontologymatching.org/2018/

The namespaces of the datasets are:

1) TomTom[21] for the Traces (`LineStrings`) and the namespace `http://www.tomtom.com/ontologies/regions#` for the Regions (`Polygons`)

2) Spaten[22] for the Traces (`LineStrings`) and the namespace `http://www.spaten.com/ontologies/regions#` for the Regions (`Polygons`)

The training datasets are available online[23]. The zipped file contains two datasets called source and target as well as the set of expected mappings (i.e., reference alignment).

2.6.2.2. RADON2 vs. RADON

The basic idea behind the original RADON approach [RADON] for topological relation discovery is to provide an indexing method combined with space tiling that allows for efficient computation of topological relations between geospatial resources. In particular, RADON presents a novel sparse index for geospatial resources. Then, based on bounding boxes of the indexed geospatial resources, RADON applies a strategy for discarding unnecessary computations of DE-9IM relations.

In RADON2 [RADON2], our concern is focused on optimizing the computing of intersection matrix (IM)used in DE9-IM standard. In the original RADON, the intersection matrix is computed for each topological relation, while RADON2 we compute the IM once for all relations among the same pair of resources. We then apply the mask for each relation to the computed IM. In particular, we buffer the IM of each pair of geometries so that all topological relations of same pair can be retrieved with no need to recompute their respective IM again. By applying this strategy, we can save the time for recomputing the IM for each individual topological relation. Moreover, calculating IM at once for each pair of geometries for all topological relations does not affect the completeness of the linking result. i.e., the F-measure of RADON2 is the same as the F-measure of RADON, which is always 1.

---

[21] http://www.tomtom.com/ontologies/traces#

[22] http://www.spaten.com/ontologies/traces#

[23] users.ics.forth.gr/~jsaveta/OAEI_HOBBIT_LinkDiscovery_2017_5/OAEI_2017_5_Sandbox_Link_Discovery.zip

2.6.2.3. Results and discussion

RADON2 has been evaluated on the Link Discovery Track Task 2 (Spaten) of OAEI2018. The basic idea behind this task was to measure how well the systems can identify DE-9IM (Dimensionally Extended nine-

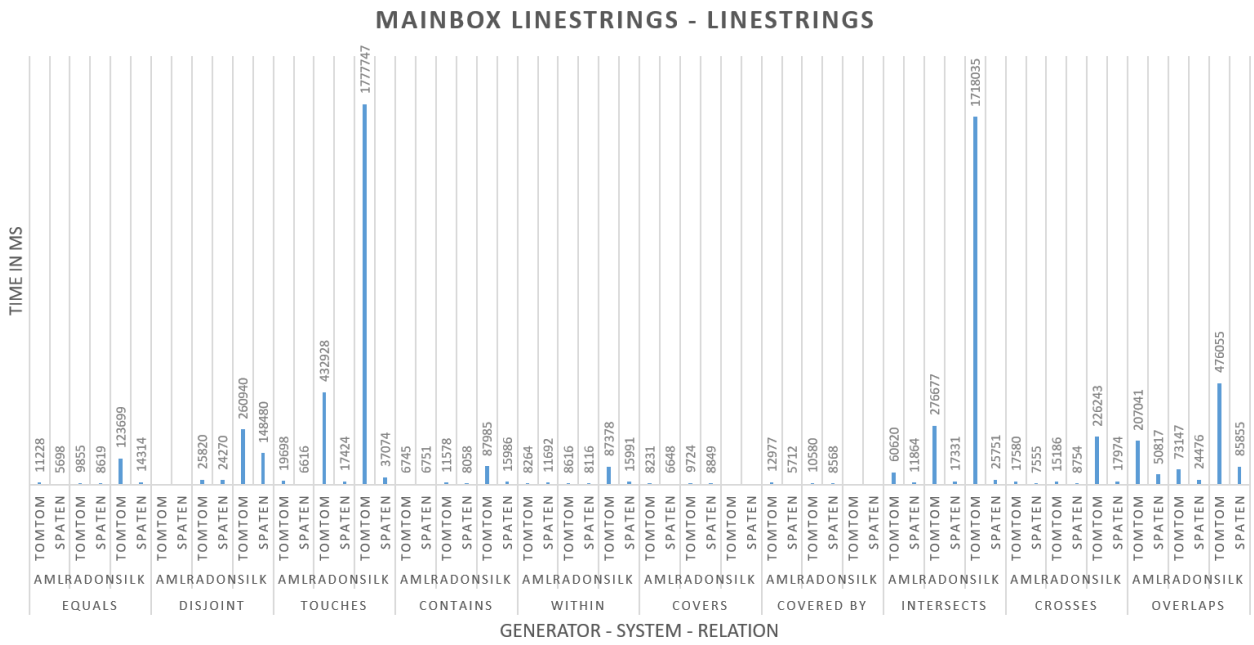


Figure 9: Runtime results of `linestrings-linestrings` Mainbox dataset

Intersection Model) topological relations. The supported spatial relations were: `equals`, `within`, `contains`, `disjoint`, `touches`, `meets`, `covers`, `coveredBy`, `intersects`, `crosses` and `overlaps`. The geospatial resources traces were represented in Well-known text (WKT) format as `LineStrings`. The result is produced as a set of links called a mapping: MS, T = {(s_i, r, t_j) | $s_i \in S$, $t_j \in T$}. All

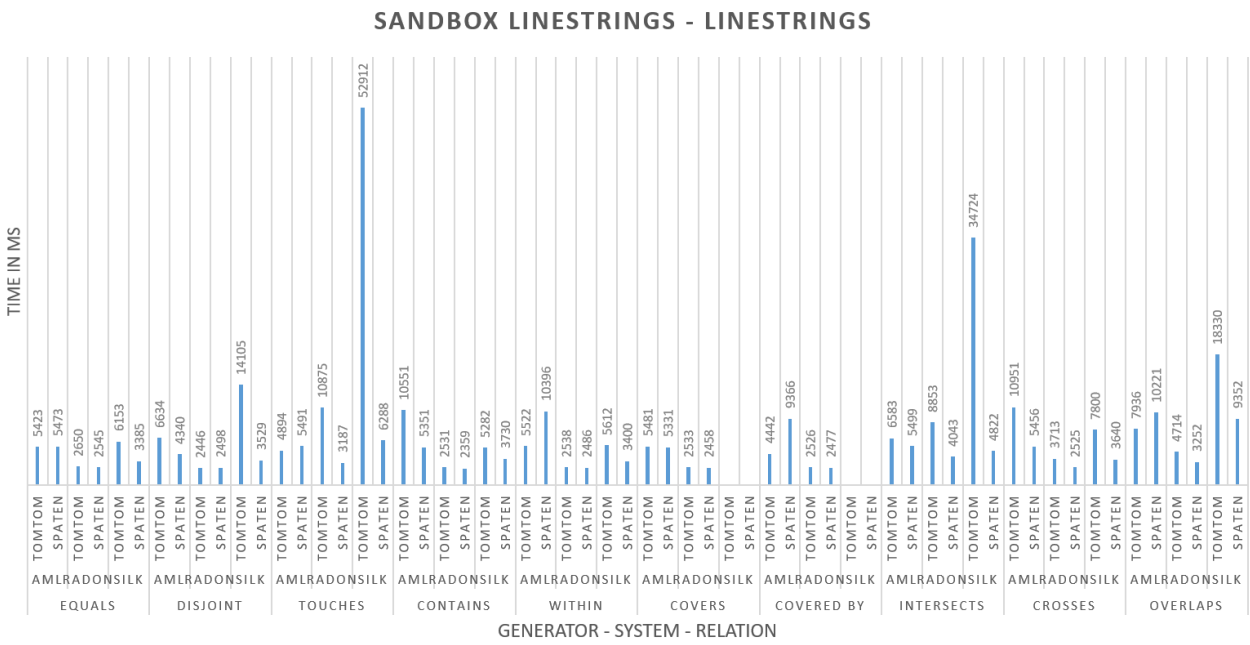


Figure 11: Runtime results of `linestrings-linestrings` Sandbox dataset

the systems were tested against two datasets: (1) the **Sandbox dataset**, with a scale of 10 instances, and (2) the **Mainbox dataset** with a scale of 5K instances. The other participants to this task in addition to RADON2 were **Agreement Maker Light (AML)**[24] and **Silk**[25]. The systems were judged on the basis of precision, recall, F-Measure and run time.

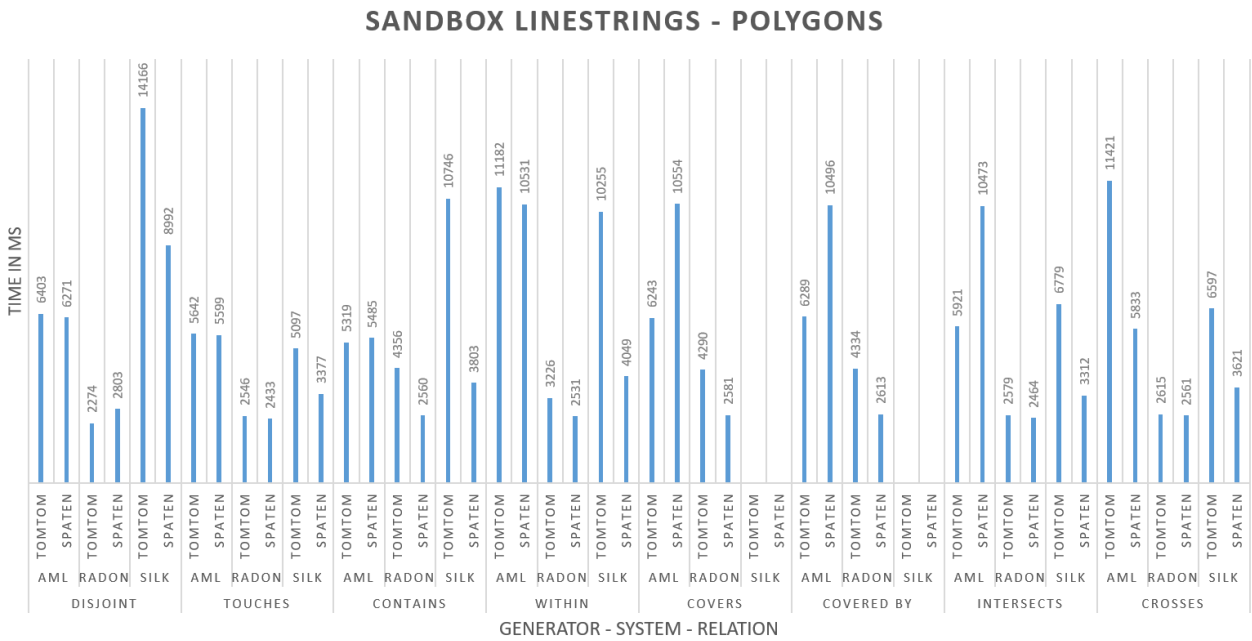


Figure 10: Runtime results of `linestrings-polygons` Sandbox dataset

[24] https://github.com/AgreementMakerLight

[25] http://silkframework.org/

The final results are shown in Figure 9, Figure 11, Figure 10 and Figure 12. Note that we are only presenting the time performance and not precision, recall and F-Measure as all were equal to 1.0. From these results we can see that RADON2 **outperforms** the other systems in all relations for the Sandbox and Mainbox (`linestrings-polygons`) (see Figure 9 and Figure 10) dataset as well as the for the Mainbox dataset (`linestrings-linestrings`) (Figure 2). For the Sandbox dataset (`linestrings-linestrings`) (Figure 11), RADON2 achieves a better performance in most of the relations (e.g., `overlaps`, `crosses`, `covered by`, `covers`, `within`, `contains`, `disjoint` and `equal`. Only for the `touches` and `intersects` AML was able to outperform RADON2 for the TomTom dataset of the sandbox (`linestrings-linestrings`). The differences in performance between `touches` and `intersects`, where AML outperforms RADON2 cannot be explained from an implementation point of view, as the set of relations share the exact optimizations. However, due to the datasets consisting exclusively of `LineStrings`, it is apparent that `touches` and `intersects` are much more likely to hold between any two geometries than other relations. Therefore, the benchmarks on these relations are the hardest in this task.

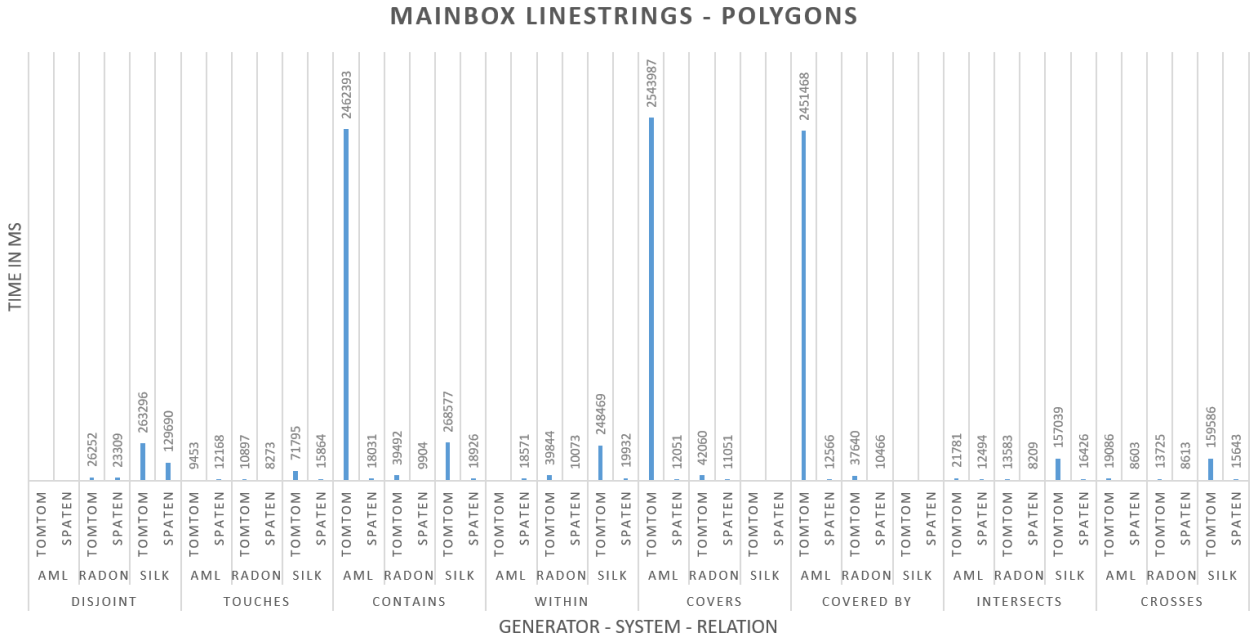


Figure 12: Runtimes results of `linestrings-polygons` Mainbox dataset

3. The DEER Framework

DEER v2.2 is currently, and to the best of our knowledge, the *only framework* that supports a rich set of enrichment operators for enriching geospatial Linked Data. DEER v1.0 was initially developed in the context of the GeoKnow [GeoKnow] project, with the aim to offer a generic purpose functionality for enriching geospatial entities, considering both thematic and geospatial properties. In the context of the SLIPO project, DEER was significantly enhanced and extended to support commercial-level requirements for the efficient, accurate, scalable, and automated enrichment of large POI datasets. Although a number of core enrichment operators was maintained from version 1.0, several modules of the framework were extended, specialized, enhanced, refactored, or even deprecated, in order to produce a new platform that satisfies the requirements of the SLIPO project, and specifically, of our industrial partners.

In the following, we thoroughly present the major enrichment facilities supported by DEER. Next, we present DEER's architecture, including its input and output, its main modules, as well our enrichment operators that allows DEER to enrich POI datasets. Finally, we provide information regarding the utilized software libraries, and software documentation.

3.1. Features and Functionality

DEER is a data enrichment framework that applies enrichment operators to discover implicit or explicit references of entities to external datasets. This way, DEER allows the enrichment of a dataset's entities from several other data sources. DEER provides facilities for manual and automatic enrichment.

DEER is a modular framework which can be easily extended. Currently, DEER provides three main types of artefacts: **dataset readers**, **dataset writers** and **enrichment operators**. Therefore, the user can easily define the set of artefacts that must be used to enrich her dataset. Also, DEER enables the user to fine tune each of its artefacts to meet her needs.

3.1.1. Dataset Readers

As its name implies, a dataset-reader component reads a dataset either from a file or a SPARQL endpoint. Dataset readers are able to read POI datasets in any serialization such as *Turtle, N-Triple, N3, JSON-LD* and *RDF-XML*. The dataset readers are considered as the input points for the DEER configuration process, i.e., the input nodes to the DEER configuration DAG.

3.1.2. Dataset Writers

The dataset-writer components act as the output nodes for the DEER configuration DAG. Dataset writers are able to export the enriched dataset to either a file or a SPARQL end point. As in the case of dataset readers, the dataset writers are able to serialize the output POI dataset using any RDF serialization.

3.1.3. Enrichment Operators

By enrichment operators we mean these artefacts in charge of enriching our POI datasets. The input for such an enrichment operator is a set one or more datasets. The output is also a set of one or more enriched datasets. Formally, a module can thus be regarded as a function $\mu : \mathbb{R} \rightarrow \mathbb{R}$, where $\mathbb{R}$ is the set of all RDF datasets. Currently, DEER implements advanced enrichment operators discussed in the following sections.

3.1.3.1. Filter Enrichment Operator

The idea behind the filter enrichment operator is to extract only a set of desired triples from the input dataset. The filter operator takes a set of triple patterns and a dataset as inputs. Applying the triple patterns against the input dataset, it filters the input dataset and produces the filtered triples as output dataset.

For example, running triple pattern `?s <http://dbpedia.org/ontology/abstract> ?o`

against an input dataset containing the Concise Bounded Description (CBD)[26] of Berlin

`http://dbpedia.org/resource/ Berlin`

will generate an output dataset that contains only Berlin's abstracts.

3.1.3.2. Linking Enrichment Operator

Links to POI resources do not occur in several knowledge bases. Here, we rely on the metrics implemented in the LIMES framework to link the resources in the input datasets. LIMES is a hybrid framework that combines the mathematical characteristics of metric spaces as well prefix-, suffix- and position filtering to compute pessimistic approximations of the similarity of instances. These approximations are then used to filter out a large amount of those instance pairs that do not suffice the mapping conditions. By these means, LIMES can reduce the number of comparisons needed during the mapping process by several orders of magnitude and complexity without losing a single link.

Linking using LIMES can be achieved in three ways:

1. *Manually*, by the means of a link specification, which is an XML-description of (1) the resources in the input and target datasets that are to be linked and (2) of the similarity measure that is to employed to link these datasets.
2. *Semi-automatically,* based on active learning. Here, the idea is that if the user is not an expert and thus unable to create a link specification, she can simply provide the framework with positive and negative examples iteratively. Based on these examples, LIMES can compute links for mapping resources with high accuracy.
3. *Automatically,* based on unsupervised machine learning. Here, the user can simply specify the sets of resources that are to be linked with each other. LIMES implements both a deterministic and non-deterministic machine-learning approaches that optimize a pseudo-F-measure to create a one-to-one mapping.

[26] For more details about CBD see http://www.w3.org/Submission/CBD/

3.1.3.3. Dereferencing Enrichment Operator

For POI datasets which contain similarity proprieties links (e.g., `owl:sameAs`), we deference all links from such dataset to other datasets by using a content negotiation on HTTP as shown in Figure 13. This returns a set of triples that needs to be filtered for relevant POI resources. Here, we use a predefined list of attributes of interest. Amongst others, we look for `geo:lat`, `geo:long`, `geo:lat_long`, `geo:line` and `geo:polygon`. The list of retrieved property values can be configured.

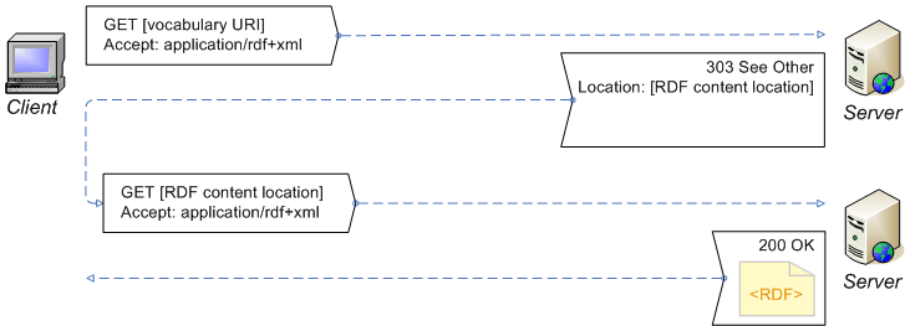


Figure 13: Content Negotiation as used by DEER (courtesy of W3C)

3.1.3.4. NER Enrichment Operator

The enrichment information hidden in datatype properties is retrieved by using Named Entity Recognition (NER) enrichment operator. DEER can inject any framework able to recognize named entities to implement the NER module.

In the current version of DEER, we rely on the *FOX* framework. While several other frameworks could be used to this end, most of the existing solutions rely solely on one of the formalisms developed for NER or simply merge the results of several tools (e.g., by using simple voting). By doing so, current approaches fail to make use of the diversity of current NER algorithms. On the other hand, it is a well-known fact that algorithms with diverse strengths and weaknesses can be aggregated in various ways to create a system that outperforms the best individual algorithms within the system. This learning paradigm is known as ensemble learning. While previous works have already suggested that ensemble learning can be used to improve NER, FOX is one of the first machine-learning approaches for ensemble learning for the NER task.

NER encompasses two main tasks:

(1) The identification of names (Also referred as instances), such as *"Germany"*, *"University of Leipzig"* and *"G. W. Leibniz"* in a given unstructured text and

(2) The classification of these names into predefined entity types (Also referred as classes), such as Location, Organization and Person. In general the NER task can be viewed as the sequential prediction problem of estimating the probabilities $P(y_i|x_{i-k}...x_{i+l}, y_{i-m}...y_{i-1})$, where $x = (x_1, .., x_n)$ is an input sequence (i.e., the preprocessed input text) and $= (y_1, ..., y_n)$ the output sequence (i.e., the entity types)

3.1.3.5. Clone Enrichment Operator

The idea behind the clone operator is to enable parallel execution of different enrichment tasks in the same dataset. The clone operator takes one dataset as input and produces $n \geq 2$ output datasets, which are all identical to the input dataset. Each of the output datasets of the split operator has its own workflow (as to be input to any other artefact). Thus, DEER is able to execute all workflows of output datasets in parallel.

3.1.3.6. Merge Enrichment Operator

The idea behind the merge enrichment operator is to enable combining datasets. The merge operator takes a set of $n \geq 2$ input datasets and simply merge them into one output dataset containing all the input dataset's triples. As in case of clone operator, the merged output dataset has its own workflow (as to be input to any other artefact).

3.1.3.7. Authority Conformation Enrichment Operator

The idea of the authority conformation enrichment operator is to uniform the enriched triples found by DEER under one authority. The conformation module changes a specified source URI authority to a specified target URI authority.

For example, using source URI authority of `http://dbpedia.org` and target URI authority of `http://slipo.eu`, the conformation module conforms a resource like `http://dbpedia.org/Berlin` to `http: //slipo.eu/Berlin`.

3.1.3.8. Predicate Conformation Enrichment Operator

The idea of the predicate conformation operator is to replace all instances of specified source property to a specified target predicated with the same object and subject values.

For example, using the source subject authority of `rdf:label` and the target subject authority of `SKOS:prefLabel`. Such configuration will change all instances of `rdf:label` to `SKOS:prefLabel`.

3.1.3.9. Geo-Fusion Enrichment Operator

The idea of the geo-fusion *enrichment operator* is to merge two or more input POI datasets into one fused output dataset. The geo-fusion operator generates its output dataset by applying a user-defined *fusion action.* By fusion action we mean the set of rules the geo-fusion operator must follow in order to fuse the result POI dataset. For example, if the user wants to fuse two POI datasets containing the geospatial properties `geo:lat` and `geo:long`, the user can select the *"take Most Detailed"* fusion policy to the select most detailed geometry from the input datasets as the one to include in the output fused dataset, i.e., in terms of lexical length of latitude and longitude values.

3.1.3.10. Geo-Distance Enrichment Operator

The idea of the *geo-distance enrichment operator* is to enrich a set of POI pairs with the great elliptic distance between them. These pairs of POIs are identified by a configurable, given relationship between them. For each pair of subject and object in the dataset obtained by filtering for this relationship, the geo coordinates of the two POIs will be taken from the properties `geo:lat` and `geo:long` which will then be

given into the equation for calculating the great elliptic distance, which is the distance that two points on earth are away from each other.

3.2. Architecture

DEER's former execution engine has been replaced by *FARADAY-CAGE*[27], a framework that provides abstractions for nodes in a directed acyclic graph that represent computation steps on more or less homogeneous data structures which are in turn represented by the graph's edges.

This kind of graph is called an execution graph. It is dynamically built from an RDF specification by use of PF4J's plugin[28] mechanism. Graph nodes in FARADAY-CAGE are PF4J extension points, which means that every step in the execution graph is performed by a plugin that is dynamically loaded and automatically parametrized from the RDF configuration. FARADAY-CAGE will also automatically parallelize plugins based on the graphs structure. The current DEER architecture is presented in Figure 14.

DEER itself consists of two modules: **deer-core** and **deer-cli**. The first is a library that implements a domain model for dataset enrichment on top of FARADAY-CAGE while the latter provides a command-line interface (CLI) that can either run a single configuration or spawn a server to submit tasks over a RESTful API. In a future release, the API will be accompanied by a web app to build configurations graphically in the browser.

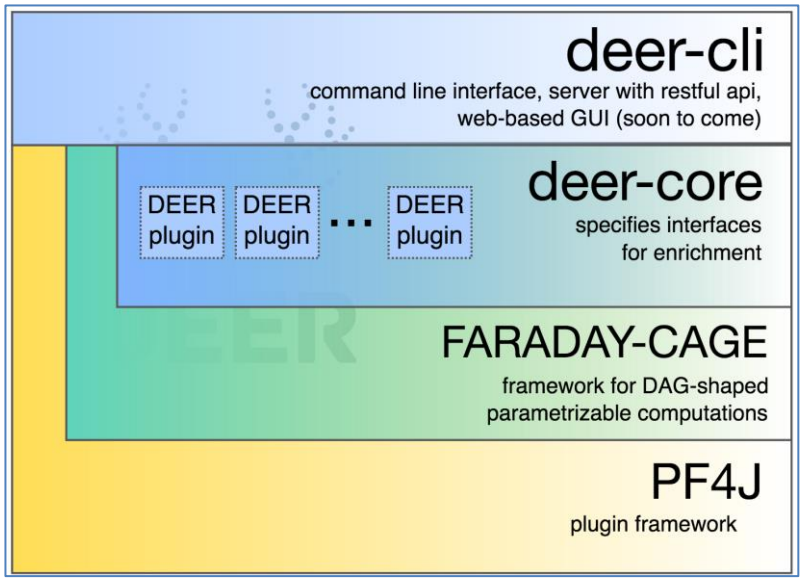


Figure 14: DEER architecture

[27] https://github.com/dice-group/faraday-cage

[28] https://github.com/pf4j/pf4j

3.2.1. FARADAY-CAGE

FARADAY-CAGE provides abstractions for DAG-shaped computations, called execution graphs. These execution graphs can be described in RDF and this description is called the configuration graph. The following four basic abstractions are provided:

- **Execution** – the most basic abstraction. Accepts a number of inputs and emits a number of outputs. The type of the inputs and outputs is homogenous but parametrized. In DEER, all inputs and outputs are RDF models.
- **Plugin** – an Execution that is dynamically loaded using PF4J. Needs to be initialized before execution.
- **Node** – a Plugin that has restrictions on how many inputs and outputs it allows for. These restrictions are called *degree bounds* and have the following notation:
- `(minIn,maxIn,minOut,maxOut)`, where
    - `minIn` is the minimum number of allowed inputs
    - `maxIn` is the maximum number of allowed inputs
    - `minIn` is the minimum number of allowed outputs
    - `maxIn` is the maximum number of allowed outputs
- **Parametrized** – this abstraction can be used together with any of the previous three and allows to customize their behaviour dynamically by providing predefined parameters in the configuration.

Moreover, FARADAY-CAGE implements validation of the configuration graph, generation of the execution graph given a valid configuration graph and parallel execution of a given execution graph.

3.2.2. DEER Plugin Types

All of DEERs Plugins are Nodes and most of them also implement Parametrized. DEER's domain model specifies three basic plugin types:

- A **ModelReader** `(0,0,1,1)` is the only plugin that can be a root node in the execution graph. i.e., an entry point for the execution graph. It is responsible for reading one RDF dataset from a dedicated source to feed it into the execution graph.
- A **ModelWriter** `(1,1,0,1)` can be an intermediary or a leaf node and it will write one RDF dataset to an external channel, e.g. a file or a triple store.
- An **EnrichmentOperator** `(1,N,1,M)` is an intermediary node that takes one or more RDF datasets as input, executes arbitrary transformations on them and outputs the result as one or more RDF datasets.

3.3. Libraries and Frameworks

DEER has dependencies to the following open-source tools/libraries:

- Google Collections[29]: A set of core libraries that includes new collection types. The library has been replaced by Google Guava[30].
- FOX[31]: A framework that integrates the Linked Data Cloud and makes use of the diversity of NLP algorithms to extract RDF triples of high accuracy out of NL.
- JUnit: A simple framework to write repeatable tests.
- LIMES-core[32]: LIMES – Link Discovery Framework for Metric Spaces.
- Apache HttpComponents[33]: A toolset of low level Java components focused on HTTP and associated protocols.
- Apache Log4j: An open-source Java logging library.
- DL-Learner[34]: A tool for supervised Machine Learning in OWL and Description Logics.
- JSON-java[35]: A reference implementation of a JSON package in Java.
- Apache Jena: A Java framework for building Semantic Web applications.
- Google Guava: A set of libraries that includes collection types (such as multimap and multiset), immutable collections, a graph library, functional types, an in-memory cache, and APIs/utilities for concurrency, I/O, hashing, primitives, reflection and string processing.
- Apache Commons Text: A library focused on algorithms working on strings.
- Apache Commons Lang: Provides a host of helper utilities for the java.lang API, notably String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and System properties. Additionally it contains basic enhancements to java.util.Date and a series of utilities dedicated to help with building methods, such as hashCode, toString and equals.
- Apache Lucene: A high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.
- Google code json-simple: A simple Java toolkit for JSON. It is used to encode or decode JSON text.
- Spring Boot Developer Tools: Additional set of tools that facilitate the application development experience. The spring-boot-devtools module can be included in any project to provide additional development-time features.
- Google Code Gson: A Java serialization/deserialization library to convert Java Objects into JSON and back.
- JDOM 2: A complete, Java-based solution for accessing, manipulating, and outputting XML data.

[29] https://code.google.com/archive/p/google-collections/

[30] https://github.com/google/guava

[31] https://github.com/AKSW/FOX

[32] https://github.com/AKSW/LIMES-dev/tree/master/limes-core

[33] https://hc.apache.org/

[34] https://github.com/AKSW/DL-Learner

[35] https://github.com/stleary/JSON-java

- FARADAY-CAGE[36] is the Framework for Acyclic Directed Graphs Yielding Parallel Computations of Great Efficiency.
- PF4J[37] is a way for a third party to extend the functionality of an application. A plugin implements extension points declared by application or other plugins.

3.4. License

DEER is an open source software and is available from GitHub[38] under the terms of the GNU AFFERO GENERAL PUBLIC LICENSE[39].

3.5. Documentation

For details documentation of DEER, please consult the following resources:

- DEER website: http://cs.uni-paderborn.de/ds/research/research-projects/active-projects/DEER/
- User manual: https://dice-group.github.io/deer/
- Javadoc: https://dice-group.github.io/deer/javadoc/
- Source code: https://github.com/SLIPO-EU/deer

3.6. Experimental Evaluation

In this Section, we present a scalability evaluation of DEER on large-scale POI data. The research question thus is, given an increasing amount of POI data, how does the runtime of DEER scale in relation to the number of POIs in the data? To answer this, we produced differently sized subsets of the large-scale synthetic POI data for Europe as described in [SLIPOD14], after it has been processed by LIMES and FAGI.

The datasets were obtained from big to small by iteratively taking 10% of the last datasets triples. In Table 3 we give an overview of the number of POIs and triples in the dataset slices that we produced.

Table 2: Evaluation Dataset Slices

Size	POIs	Triples
Tiny	84	5,272
Small	955	52,720
Regular	9,287	527,200

36 https://github.com/dice-group/faraday-cage

37 https://github.com/pf4j/pf4j

38 https://github.com/dice-group/LIMES/blob/master

39 https://github.com/dice-group/deer/blob/master/LICENSE

Big	92,926	5,272,000
Huge	929,943	52,718,419

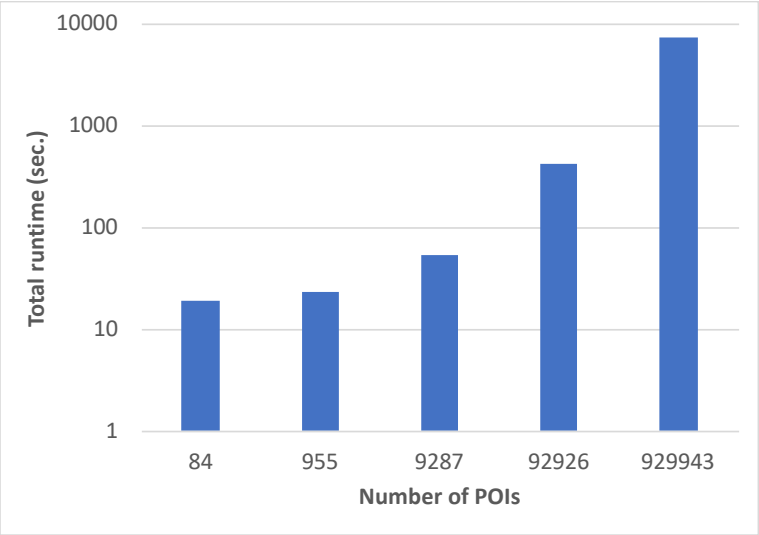


Figure 15: DEER Scaling with increasing number of POIs

The results as displayed in Figure 15 show that for very small dataset slices (tiny and small) the overhead of querying SPARQL endpoints over HTTP dominates the run time. However, with increasing input size, our approach seems to be scaling linearly

4. The FAGI framework

FAGI is currently the *only software framework* that supports a rich set of facilities for fusing geospatial Linked Data. FAGI v1.0 was initially developed in the context of the GeoKnow [GeoKnow] project, with the aim to offer a generic purpose functionality for fusing geospatial entities, considering both thematic and geospatial properties. In the context of the SLIPO project, FAGI was significantly enhanced and extended to support commercial-level requirements for the efficient, accurate, scalable, and automated fusion of large POI datasets. Although a considerable amount of core fusion functionality was maintained from version 1.0, several modules of the framework were extended, specialized, enhanced, refactored, or even deprecated, in order to produce a new platform that satisfies the requirements of the SLIPO project, and specifically, of our industrial partners.

In the following, we thoroughly present the major fusion and quality assessment facilities supported by FAGI. Next, we present FAGI's architecture, including its input and output, its main modules, as well our partitioning scheme that allows FAGI to process large amount of POI data in parallel. Finally, we provide information regarding the utilized software libraries, and software documentation.

4.1. Features and functionality

The major functionality of FAGI is summarized in the form of five (5) high-level feature categories, which are further elaborated in the sub-sections that follow[40].

- **Fusion** of the properties of linked POIs through the specification of fusion rules (Section 4.1.1).
- **Validation of input links** between POIs through the specification of validation rules (Section 4.1.2).
- **Recommendation** of link validation and fusion actions through learning mechanisms (Section 4.1.3).
- Extraction of **quality indicators/statistics** on the input and output datasets (Section 4.1.4).
- **Provenance** of the fusion process. (Section 4.1.5)
- **Configurable** output of fusion results (Section 4.1.6).

[40] In the remainder of the document, the terms '**first dataset/left dataset/dataset A**' and 'second dataset/**right dataset/dataset B**' are used interchangeably to discriminate between the two input POI datasets of FAGI. Similar terminology is used for the POIs and their property values.

4.1.1. POI fusion

4.1.1.1. Fusion actions

Fusion of POI properties is the core task of FAGI. Fusion actions are defined regarding both thematic and geospatial properties of linked POIs. A certain fusion action is atomically applied on a pair of matching properties between two linked POIs. Consequently, different fusion actions are allowed to be defined for different types of POI properties. FAGI v3.0 supports the following fusion actions. Note that, at the end of each feature, we denote whether it regards thematic or spatial properties of POIs.

- **keep-left**. Keeps the property value of the POI from the left input dataset, in the fused POI description. *(Thematic+ Geospatial).*
- **keep-right**. Keeps the property value of the POI from the right input dataset, in the fused POI description. *(Thematic+ Geospatial).*
- **keep-both**. Keeps both property values of the linked POIs, as separate properties in the fused POI description. *(Thematic+ Geospatial).*
- **concatenate**. Keeps both property values of the linked POIs, as a concatenated literal in the same, single property of the fused POI description. *(Thematic).*
- **concatenate-geometries**. Creates and keeps in the fused POI description a *GEOMETRYCOLLECTION* geometry from the two individual geometries of the input linked POIs. *(Geospatial).*
- **keep-longest**. Keeps the longest literal (property value) of the linked POIs in the fused POI description. *(Thematic).*
- **keep-most-complete-name**. Keeps all distinct types of names (such as official international, etc.) and distinct language name property values. *(Thematic).*
- **keep-more-points**. Keeps, from the initial geometries of the linked POIs, the one that is composed by the most points. *(Geospatial).*
- **keep-more-points-and-shift**. Keeps the geometry with the most points and shifts its centroid to the centroid of the other geometry. *(Geospatial).*
- **shift-left-geometry**. Shifts the geometry of the left source entity to the centroid of the right. *(Geospatial).*
- **shift-right-geometry**. Shifts the geometry of the right source entity to the centroid of the left. *(Geospatial).*
- **keep-most-recent**. Keeps the value from the dataset that has been denoted as containing the most recent/up-to-date data. *(Thematic+ Geospatial).*
- **Keep-recommended.** Utilizes an ML model to predict the fusion action. (Thematic + Geospatial).

The result of a fusion action on a pair of matching properties is the replacement of these properties and their values, by the property(-ies) value(-s) that are prescribed by the fusion action.

Most of the above fusion actions were inherited from FAGI v.1.0, since they comprise the core fusion functionality that needs to be performed on geospatial datasets. With the aim to incorporate quality

assessment processes within fusion, FAGI v3.0 further extends this set of fusion actions in order to take into account the potential **ambiguity** in performed fusion actions. Specifically, for each of the above fusion actions a "counterpart" fusion action is defined, that denotes that the specific fusion outcome is ambiguous and needs to be examined by an expert at a later stage of the integration. The output of the application of an ambiguous counterpart of a fusion action is: (a) the exact outcome of the baseline fusion action and (b) an additional RDF triple that denotes that the specific result is ambiguous. For example, if the ambiguity refers to the address property of a POI, the resulting RDF triple would look like this:

```
<http://slipo.eu/def#POI_URI> <http://slipo.eu/def#hasAmbiguous> <http://slipo.eu/def#address>
```

The aforementioned fusion actions are utilized within fusion rule specifications (see Section 4.1.1.3), which examine certain properties of the linked POIs and decide whether to apply the fusion action or not. Thus, it is possible that a certain fusion rule specification does not cover all possible use cases, and consequently, no fusion action from the specification can be applied. To handle such cases, *default* fusion actions and *dataset-level* fusion actions are defined. For each pair of matching properties, FAGI first evaluates all available fusion rules from the specification. If none of the rules applies for a specific pair of properties, then the default fusion action for the specific pair of matching properties is applied. If no fusion specification exists for the pair of matching properties, then the dataset-level fusion is applied. While the supported default fusion actions are drawn from the list above, the supported dataset-level fusion actions are limited to: *keep-left*; *keep-right*; *keep-both*.

4.1.1.2. Condition functions

A *condition* is a function applied on the value of a property, or on the values of a pair of matching properties of two linked POIs. It evaluates to `True` or `False` and may regard several aspects of the property's values. The currently implemented conditions are enumerated below. Note that we use the term "literal" for denoting string values of thematic properties, and the term geometry for the values of geospatial properties.

- **isSameSimpleNormalize:** Checks if the two given literals are same. It normalizes the two literals with some basic steps and uses the provided similarity (default JaroWinkler) to return True if the result is above the provided threshold. The threshold must be in the interval [0,1].
- **isSameCustomNormalize:** Checks if the two given literals are same. It normalizes the two literals with some extra steps in addition to the simple normalization, using FAGISimilarity (see Section 4.1.3.2) and returns True if the result is above the provided threshold. The threshold must be in the interval [0,1].
- **isSameNormalized:** Checks if the two given literals are same. It normalizes the two literals with some basic steps and uses the provided similarity (default JaroWinkler). No threshold provided.
- **isLiteralAbbreviation.** Checks if the given literal is or contains an abbreviation of some form.
- **LiteralContains**. Checks whether a literal contains a specific string (*which might also be the second literal*).
- **isLiteralLonger**. Examines if the first literal is longer than the second one.
- **literalHasLanguageAnnotation**. Examines if the literal includes an annotation on the language used.

- **literalsHaveSameLanguageAnnotation:** Checks if the two literals have the same language annotation (tag).
- **isLiteralsSameLanguage**. Compares the language of two literals, in case both include language annotations.
- **isLiteralNumeric**. Examines if the literal is a number.
- **isNameValueOfficial:** Checks if the value of the name property is tagged as official in the source data.
- **isGeometryMoreComplex:** Checks if the first geometry has more points than the second.
- **isPointGeometry**. Checks whether the geometry is a point geometry.
- **geometriesHaveSameCentroid**. Examines whether two geometries have centroids close enough as to be considered the same. It requires a user provided distance threshold for the centroids.
- **isGeometryContained**. Examines whether the first geometry contains the second geometry.
- **geometriesOverlap**. Examines whether the two geometries overlap.
- **geometriesIntersect:** Checks if the given geometries intersect.
- **geometriesCloserThan**. Examines whether the distance of the two geometries is smaller than a user provided threshold.
- **geometriesHaveSameArea**. Examines whether two geometries have areas close enough as to be considered the same.
- **isGeometryCoveredBy:** Checks if the first geometry is covered by the second geometry. The definition of `coveredBy` follows the DE-9IM model.
- **isPhoneNumberParsable:** Checks if the given phone number consists of only numbers or contains special character and/or exit code.
- **isSamePhoneNumber:** Checks if the given phone numbers are the same having performed some simple normalization steps (*removal of non-numeric characters*).
- **isSamePhoneNumberUsingExitCode:** Checks if the given phone numbers are the same, after having performed some simple normalization steps (*removal of non-numeric characters, normalization of exit code notation*).
- **isSamePhoneNumberCustomNormalize:** Checks if the given phone numbers are the same. It first applies the normalization steps of the two condition functions above. If the equality comparison fails, some custom steps for normalization are executed and the function rechecks for equality (*ignoring country code, area code and trailing digits*).
- **phoneHasMoreDigits**. Examines if the first phone has more digits than the second one.
- **exists:** Checks if the given property exists in the model of the entity.
    - **notExists:** The reverse function of exists. Returns true if the selected property is not found in the model.
- **isDateKnownFormat:** Checks if the given date string complies to a known data format. The known formats are defined at FAGI's specification constants: [FAGISPEC].

- **isDatePrimaryFormat:** Checks if the given date String is written as a primary format as defined at FAGI's specification constants.
- **isValidDate:** Evaluates the given date against the target format.
- **isDateSame**: Evaluates if the given dates are the same using a tolerance value in days.

4.1.1.3. Fusion specifications

This module allows the definition of fusion rules via XML specifications. Fusion rules are comprised by: *expressions* of combined *condition functions* that evaluate several characteristics of the properties of two linked POIs, and *fusion actions* that are performed if the expressions are evaluated to true (as described above). The individual condition functions are combined through the "AND", "OR" and "NOT" logical operators, in order to form an expression. This way, several aspects of two linked POIs can be examined in order to decide a specific fusion action. The currently implemented conditions are enumerated in Section 4.1.1.2. Indicatively, condition functions may examine the similarity of two strings, normalize and compare two phone numbers, examine if a property has an empty value, examine the data type of a property value, etc.

In order to compose such rules, a simple XML syntax is used, where the user is expected to provide the properties to be fused, the conditions to be evaluated (and the respected properties for each condition), the way the conditions will be combined into expressions and the respective fusion actions. A user interface is also provided that allows the graphical construction of such rules. The user is able to define an arbitrary set of fusion rules, considering any pair of matching properties between linked POIs, ranging from a simple condition on one property value, to arbitrarily large combinations of conditions, that form complex rules, on several property values. These rules are evaluated with sequential priority on the whole set of input POI data. This allows the realization of significantly automated fusion workflows, on massive amounts of POI data, with the user input required only at the beginning of the process (rule specification). Except from defining such expressions, the user is also able to define the fusion strategy on POI-ensemble properties in order to apply fusion actions based on distinctive characteristics of specific attributes. For example, when handling address property values, the user might choose keeping a unique value based on a voting strategy of all values that participate in the corresponding POI-ensemble.

Specifically, the **rule specification scheme** of FAGI v3.0 is based on the following concepts:

- **Pair of matching properties**. The pair of properties upon which the fusion action will be performed.
- **Condition function.** A condition is a function applied on the value of a property or on the values of a pair of matching properties of two linked POIs. It evaluates to `True` or `False` and may regard several aspects of the property's values.
- **Condition expression**. A condition expression is constructed by combining several conditions with the logical operators `AND`, `OR` and `NOT`. A condition expression also evaluates to `True` or `False` and its evaluation value decides whether a fusion action will be performed on a pair of matching properties or not.
- **External properties**. These properties are not considered for fusion, but are rather utilized within condition functions to decide on the fusion action of the properties to be fused.

- **Fusion action rules**. The fusion rule specification scheme allows the definition of arbitrary pairs of the form (`condition expression, fusion action`), for the same pair of matching properties. These pairs, denoted as `fusion action rules`, are evaluated sequentially; thus, their priority is defined by their ordering in the fusion rule specification file.
- **Fusion action**. The fusion action is the action that will be applied on the values of the matching properties, and it is always part of an `action rule.`
- **Default fusion action.** A default fusion action is applied to the properties to be fused, in the case where none of the fusion action rules is eventually applied (i.e. all respective condition expressions evaluate to `False`).
- **Fusion rule**. Having defined the above concepts, a fusion rule is comprised by the following elements:
  a. A pair of matching properties to be fused.
  b. A set of external properties to be used in the condition functions.
  c. An ordering of fusion action rules, with each action rule regarding the same pair of properties from (a.) and being composed by a condition expression and a fusion action.
  d. A default fusion action, in case none of the action rules is eventually applied.
- **POI-ensembles**. In several cases, a POI of the left dataset may be linked with several POIs of the right and vice versa. In order to assess this, the rule specifications allow the definition of fusion strategies on these group of POIs. Specifically, we define two categories of property values:
  a. Functional properties.
  b. Non-functional properties.
- As functional properties we refer to property values that uniquely characterize a POI, such as address-street and address-number, geometry, etc. As non-functional properties we refer to properties that may have multiple values for the resulting fused POI. The user can define which properties should be functional and non-functional in the XML specification. This way, FAGI treats these properties with a different fusion strategy. Specifically, the fusion process will keep a unique value for each functional property based on a voting strategy (the most frequent value of these properties will be kept), and for non-functional properties, FAGI constructs separate property paths for all values in order for the final fused POI to retain all these values.

The total set of fusion rules, along with the dataset-level fusion action, that are defined for a specific pair of input POI datasets comprise the **fusion rule specification** for the respective input.

In Figure 16, we present a sample XML snippet corresponding to the configuration of fusion rules for a pair of matching properties.

```
<rule>
   <propertyA>http://slipo.eu/def#name http://slipo.eu/def#nameValue</propertyA>
   <propertyB>http://slipo.eu/def#name http://slipo.eu/def#nameValue</propertyB>
```

```
<externalProperty id="a1">http://slipo.eu/def#phone
http://slipo.eu/def#contactValue</externalProperty>
<externalProperty id="b1">http://slipo.eu/def#phone
http://slipo.eu/def#contactValue</externalProperty>
<actionRuleSet>
    <actionRule>
       <condition>
           <expression>
              <and>
                  <function>isSamePhoneNumberCustomNormalize(a1,b1)</function>
                  <function>isSameCustomNormalize(a,b,0.6)</function>
              </and>
           </expression>
       </condition>
       <action>keep-longest</action>
    </actionRule>
    <actionRule>
       <condition>
           <expression>
              <and>
                  <function>isSamePhoneNumberCustomNormalize(a1,b1)</function>
                  <expression>
                     <not>
                         <function>isSameCustomNormalize(a,b,0.6)</function>
                     </not>
                  </expression>
              </and>
           </expression>
       </condition>
       <action>keep-both</action>
    </actionRuleSet>
  </actionRuleSet>
 <defaultAction>keep-right</defaultAction>
<ensembles>
  <functionalProperties>
    http://slipo.eu/def#address http://slipo.eu/def#street;
    http://slipo.eu/def#hasGeometry http://www.opengis.net/ont/geosparql#asWKT
  </functionalProperties>
  <nonFunctionalProperties>
    http://slipo.eu/def#name http://slipo.eu/def#nameValue;
    http://slipo.eu/def#phone http://slipo.eu/def#contactValue
  </nonFunctionalProperties>
```

```
  </ensembles>
</rule>
```

Figure 16: Sample Fusion Rule Specification

The rule presented in Figure 16 handles the fusion of the *name* property values between linked POIs. It consists of two action rules that are evaluated in the order they are defined. The first action rule examines whether the phone of the two POIs match, and whether the names of the two POIs have a similarity larger than a specific threshold (0.6). If both conditions apply, the two name values are fused by keeping the larger string. That is, the names are considered similar enough to identify them as the same name, but potentially written in a slightly different way. The second action rule also checks whether the phones of the two POIs match, but now checks whether the name similarity is lower than the threshold. In this case, both name values are kept separate attributes of the fused POI, since they are probably different names for the same POI. Finally, in case none of the two action rules applies (i.e., their conditions evaluate to `False`), the default fusion action prescribes to keep the name value from the second (right) dataset.

Finally, inside the *ensembles* tag, the rules indicate that when a POI-ensemble is identified, the address-street and geometry values should result in a unique value in the final fused POI, and all phone numbers and name values should be kept for each group of POIs (detailed description in Section 4.1.1.4).

The mechanism we introduced for constructing fusion rule specifications significantly facilitates the *automation* of the fusion process, since it allows the user to define both **generic rules**, that cover a wide range of trivial fusion cases (*e.g., when one of the properties has an empty value*), as well as an arbitrary number of very **specific and detailed rules**, that capture and handle more complex scenarios (*e.g., when the names of two POIs are not similar enough, but their phones are the same*). As such, a large part of the (*previously manually*) fusion process can be automated. Of course, this does not guarantee an optimal accuracy on the validated links, or on the selected fusion actions. However, it allows the data integrator to process the linked input datasets in batches and define rules of diverse granularity that will potentially relieve her from manually examining a large percentage of the fused POI data, during a later stage of quality assurance/manual validation.

4.1.1.4. Fusion of POI-ensembles

As mentioned previously, there are some scenarios, in which a POI from the left dataset might be linked with multiple POIs from the right dataset and vice versa. This behavior of the interlinking process might indicate falsely linked POIs, but in many cases these groups of POIs (referred as POI-ensembles) are valid and describe the same real-world entity. Several examples from the use cases indicate the POI-ensembles should be handled with a different strategy instead of a -pair-wise fusion based on the rule specifications.

Following this requirement, we developed a separate handler in the core fusion component to assess this kind of input. First, the interlinking results are processed in order to find such groups of links by utilizing the score of each link. This way, cross-linking between different groups is pruned based on the highest score (per-link) and POI-ensembles are modeled in-memory by ensuring that all POIs identified as ensembles will be part of a single group with which they share the strongest connection. After this pre-process step, all POI-ensembles are handled by the fusion process by applying the corresponding ensembles rules specification. The final fused POI will contain unique values for each *functional* property and a complete set

of values for all the *non-functional* properties. The unique property values are selected by the voting strategy. When the voting strategy fails to identify the most frequent value (all property values are different), it selects an arbitrary value from the set. The rule specifications allow to mark these fusion actions as ambiguous by defining proper condition in the rule set.

The link validation (described in Section 4.1.2) of POI-ensembles follows the same validation rules defined for all links, but the validation process is performed after the POI-ensembles that have been identified. This approach seems more appropriate because the POI-ensembles are constructed without the link-validation bias and even if a POI-ensemble falls back as a simple link (all links are rejected during the link-validation, except from a single POI-to-POI link) it will be treated with the default provided rules and not as an ensemble.

4.1.2. POI link Validation

4.1.2.1. Validation actions

An important aspect of quality assurance lies in validating the fusion input and deciding whether the linked entities should be either fused, further examined, or rejected as erroneous. To this end, in FAGI v3.0 we define a set of validation actions, as well as a validation specification scheme, for constructing validation rules and evaluating them on the input POI data. The supported **link validation actions** are the following:

- **accept-link**. The link between two input POIs is accepted and the fusion process is executed as prescribed by the user.
- **reject-link**. The link between two input POIs is rejected. No fusion processes are executed for the specific pair of POIs.
- **accept-mark-ambiguous**. The link between two input POIs is accepted and the fusion process is executed as prescribed by the user. Further, the produced (*fused*) POI is marked as ambiguous, via an additional RDF triple, for later examination. Also, the initial POI descriptions (*from which the fused POI was produced*) are kept in an auxiliary output dataset, in case they are utilized in a later examination/validation stage.
- **reject-mark-ambiguous**. The link between two input POIs is rejected. No fusion processes are executed for the specific pair of POIs. Further, the individual POIs are marked as ambiguous, via additional RDF triples, for later examination. Also, the initial POI descriptions (*whose link was rejected*) are kept in an auxiliary output dataset, in case they are utilized in the later examination of the POI.
- **ml-validation.** Accepts/rejects the link based on the ML model prediction.

4.1.2.2. Condition functions

A *condition* is a function applied on the value of a property, or on the values of a pair of matching properties of two linked POIs. It evaluates to `True` or `False` and may regard several aspects of the property's values. The condition functions that are utilized within validation action specifications are exactly the same with the ones for fusion action specifications, as enumerated in Section 4.1.1.2.

4.1.2.3. Validation specifications

Validation rule specifications follow the same rationale with fusion rule specifications. Specifically, validation rule specifications consider the following concepts:

- **Validation action**. One of the four available link validation actions.
- **Condition function.** A condition is a function applied on the value of a property or on the values of a pair of matching properties of two linked POIs. It evaluates to `True` or `False` and may regard several aspects of the property's values.
- **Condition expression**. A condition expression is constructed by combining several conditions with the logical operators `AND`, `OR` and `NOT`. A condition expression also evaluates to `True` or `False` and its evaluation value decides which validation action will be performed on a pair linked POIs.
- **External properties**. External properties are the only kind of properties that are valid within validation rule specifications, since no properties are fused. The functionality of external properties is identical to the functionality of external properties for fusion rule specifications, i.e. they are utilized within condition functions to decide which validation action to perform.
- **Validation action rules**. The validation rule specification scheme allows the definition of arbitrary pairs of the form `(condition expression, validation action)`, for the same pair of linked POIs. These pairs, denoted as `validation action rules`, are evaluated sequentially; thus, their priority is defined by their ordering in the validation rule specification file.
- **Default validation action.** It has identical functionality to the default fusion action, i.e. it is applied to the pair of linked POIs, in the case where none of the validation action rules is eventually applied (i.e. all respective condition expressions evaluate to `False`).
- **Validation rule**. Having defined the above concepts, a validation rule is comprised by the following elements:
  a. A set of external properties to be used in the condition expressions.
  b. An ordering of validation action rules, with each one being composed by a condition expression and a validation action.
  c. A default validation action, in case none of the action rules is eventually applied.

The total set of validation rules that are defined for a specific pair of input POI datasets comprises the **validation rule specification** for the respective input.

In Figure 17, we present a sample XML snippet corresponding to the configuration of validation rules for a pair of POIs.

```
<validationRule>
    <defaultAction>accept</defaultAction>
        <actionRuleSet>
            <actionRule>
               <action>reject</action>
                <condition>
                   <expression>
                      <not>
```

```
                <function> isSameCustomNormalize(a0, b0, 0.5)</function>
            </not>
        </expression>
    </condition>
</actionRule>
<actionRule>
    <action>reject-mark-ambiguous</action>
    <condition>
        <expression>
            <not>
                <function> isSameCustomNormalize(a0, b0, 0.7)</function>
            </not>
        </expression>
    </condition>
</actionRule>
<actionRule>
    <action>accept-mark-ambiguous</action>
    <condition>
        <expression>
            <not>
                <function> isSameCustomNormalize(a0, b0, 0.9)</function>
            </not>
        </expression>
    </condition>
</actionRule>
</actionRuleSet>
<externalProperty id="a0">http://slipo.eu/def#name
http://slipo.eu/def#nameValue</externalProperty>
<externalProperty id="b0">http://slipo.eu/def#name
http://slipo.eu/def#nameValue</externalProperty>
</validationRule>
```

Figure 17: Sample Link Validation Rule Specification

In the example presented in Figure 17, the validation rule utilizes the *name* property values between linked POIs. It consists of two action rules that are evaluated in the order they are defined. The first validation action rule examines whether the similarity of the names of the two POIs is lower than the given threshold (0.5) by using the 'not' operator in the condition expression. In that case, the names are considered dissimilar enough to reject the link (*the condition will evaluate to* `True`). If the first condition does not evaluate to `True`, the second validation action rule will be checked. If the application reaches the second condition, we know that the similarity of the POI names is over 0.5, so we want to define another condition. Thus, we construct a condition that will return `True` if the similarity of the names is below the given threshold (0.7). Now we know that the name values have a similarity between 0.5 and 0.7, which it can be considered a reject case, but instead of just rejecting it, we will also mark it ambiguous in order to be able to check these examples later. The third condition evaluates to `True` for examples with name similarities between 0.7 and 0.9. We want to keep these links and apply the fusion process, but the similarity score does not indicate an

exact match, so we mark these links as ambiguous. Finally, in case none of the three conditions evaluate to `True`, the default validation action will apply which will accept the link (*name values should be almost identical with similarity above 0.9*).

4.1.3. Recommending POI link validation and fusion

Moving beyond user-defined rule specifications for link validation and fusion, FAGI v3.0 incorporates functionality for *automatic recommendation* of validation and fusion actions. In this subsection, we briefly describe the problem definition and formulation and an overview of the solutions we have developed.

4.1.3.1. Problem formulation

FAGI receives as input a pair of POI datasets, A and B, which contain POI entities accompanied with several properties, as well as a links file L, which contains a set of links connecting POIs from A with POIs from B. Considering a POI *a* in A that is linked with a POI *b* in B, and the sets of properties $\{P_{a,i}\}$ and $\{P_{b,i}\}$ that describe them respectively, FAGI handles two tasks:

- **Decide validation action**. Decide whether POIs *a* and *b* actually correspond to the same real-world entity or they are wrongly interlinked. This can be formulated as a binary classification problem with output classes "accept" and "reject".
- **Decide fusion action**. Decide which is the most fitting fusion action for each pair *i* of matching properties ($P_{a,i}$, $P_{b,i}$) of the two POIs. This can be formulated as a multi-class classification problem, with the different fusion actions being considered as the classes of the problem.

Our goal is to develop the functionality that facilitates the accurate execution of the above two tasks, requiring the minimum effort by the user both in the selection of validation/fusion actions and in the post-processing, manual examination/validation of the fusion results.

4.1.3.2. Overview of solutions

Similarity-based link validation. The first task that needs to be handled by FAGI is link validation. Our first approach is to solve this problem through a similarity-based method, following a wide range of works in the literature. Specifically, our goal has been to design a POI specific meta-similarity function that can be used to accurately identify/distinguish between correctly and wrongly linked POIs. The method we implemented, called *FAGISimilarity*, comprises several string processing steps, a limited set of parameters-weights, an underlying, basic similarity function and a similarity threshold that can be identified at an initial fine-tuning stage. The main objective of FAGISimilarity is to identify and separately compare different parts of the names of two linked POIs, based on the rationale that (a) matching and non-matching parts of the names should be compared differently and (b) matching of frequent terms should be compared differently. FAGISimilarity can be characterized as a "meta-similarity" function, in the sense that it prescribes a series of operations to be performed, utilizing more basic similarity measures in several steps. In our method, we support the utilization of the following basic similarity measures: *Levenshtein, 2-Gram, Longest Common Subsequence, Jaro, Jaro-Winkler*.

Supervised link validation and fusion action recommendation. These two tasks are jointly solved by applying the same learning process and only differentiating between either binary, or multi-class

classification. This approach takes as input training data provided by expert users. This data is comprised by pairs of linked POIs, along with their matching properties. The expert assigns, for each such training instance, a *link label* denoting whether it is a valid link or not, as well as a set of *fusion labels* denoting the proper fusion action per pair of matching properties. The set of training instances with the respective labels train two distinct classifiers in order to learn two models for the two tasks at hand: (i) classification of a new pair of linked POIs as "accept" or "reject"; (ii) Classification of each pair of matching properties to one of the available fusion actions in the system.

4.1.3.3. FAGI v3.0 support

Automatic validation and fusion recommendation is integrated in FAGI v3.0 as an instrument *auxiliary* to the rule specification functionality mechanism. As such, the user performs validation and fusion actions by defining validation and fusion rule specifications and without advising the recommendation mechanism. However, FAGI utilizes ML models for link-validation and fusion action recommendation that can be used in combination with the rule specification. This is done by defining appropriate fusion-actions produced by the models and can be incorporated along with the rule-based scheme. For example, the user can define several rule expressions and if some conditions are met (or not met) the fusion action could be selected by FAGI using the models. The training of these models is performed independently, and the input is given in the configuration of FAGI. The learning/recommendation mechanism when used with conjunction of the ambiguous results (described in Section 4.1.3) further automates the fusion process, by reducing the need for manual validation of fusion results.

The solution we selected was a Random Forest model for its ability to model complex relationships as well as its robustness to overfitting and parameter selection. Specifically, we used the java implementation of Random Forest available in WEKA library. The number of trees for each model was set to 100 and each tree was grown without pruning.

4.1.4. Quality indicators and statistics

FAGI v3.0 supports the extraction of an extended set of quality indicators and statistics, both at the beginning, and the end of the fusion process. The user can review several statistics on the input, linked POI datasets, before performing fusion on them (*pre-fusion statistics*), as well as on the output, fused data (post-fusion statistics). The goal of the former is to allow the integrator to obtain an overview of the data at hand, which may help her properly define and configure the validation/fusion rules. The goal of the latter is to assist the user in the examination/validation of the fusion results, and potentially guide her into re-configuring and re-executing the fusion process.

4.1.4.1. Pre-fusion statistics

FAGI v3.0 supports the extraction of the following statistics/indicators:

- Statistics on individual input datasets:
    - Number of POI entities in each input dataset.
    - Total number of triples in each input dataset, i.e., total number of properties for all POIs.
    - Total numbers of empty and non-empty triples in each input dataset.

    - Average number of properties per POI in each input dataset.
    - Average number of empty and non-empty properties in each input dataset.
    - Average number of categories(tags) per POI in each input dataset.
    - Total number of POIs that have a specific property in each input dataset.
    - Number of empty and non-empty values a specific property in each input dataset.
- Statistics related to linked POIs of the input datasets:
    - Ratio of linked POIs to total number of POIs in each input dataset.
    - Total number of triples in each input dataset (*i.e., total number of properties for all POIs*), corresponding only to linked POIs.
    - Total numbers of empty and non-empty triples in each input dataset, corresponding only to linked POIs.
    - Average number of properties per POI in each input dataset, corresponding only to linked POIs.
    - Average number of empty and non-empty properties in each input dataset, corresponding only to linked POIs.
    - Total number of POIs that have a specific property in each input dataset, regarding only linked POIs.
    - Total numbers of empty and non-empty values for a specific property in each input dataset, regarding only linked POIs.
    - Average number of categories(tags) per POI in each input dataset, regarding only linked POIs.
    - Number of POI name property values from dataset A that are longer (longer literals) than the names of the corresponding (linked) POIs from dataset B (also the inverse indicator).
    - Number of POI phone property values from dataset A that are longer than the names of the corresponding (linked) POIs from dataset B (*longer phone strings imply more proper phone format, e.g., containment of full country/exit codes*) (also the inverse indicator).
    - Number of fully matching address streets between linked POIs in the two datasets.
    - Number of fully matching address numbers between linked POIs in the two datasets.
- Frequent terms statistics. For several important POI properties (*e.g., name, address, categories*) a list of frequent terms on each dataset is produced. These properties may be also custom selected by the user.

4.1.4.2. Post-fusion statistics

FAGI v3.0 supports the extraction of the following statistics/indicators:

- Number of fused POIs vs. initial links, i.e., the number of POI links that were not rejected by FAGI and were actually considered for fusion of their attributes.
- Number of rejected POI links vs. initial links, i.e., the number of POI links that were eventually rejected by FAGI.

- Number of fusion actions that: Kept left value; Kept right value; Concatenated left and right value and kept as one; Kept both values as separate properties; Kept longest value
- For each fusion rule that was defined in the fusion specification, the number of times it was executed and produced a fused POI.
- For each link validation rule that was defined in the validation specification, the number of times it was executed.
- Number/percentage of fused POIs that were marked as ambiguous (and thus require further examination/validation) vs. the number of initial links.
- Number/percentage of rejected POIs that were marked as ambiguous (and thus require further examination/validation) vs. the number of initial links.
-

4.1.4.3. Quality Metrics

FAGI v3.0 output measures that indicate the overall quality of the fused results, as well as measures on each fused entity. In particular:

- **Attribute gain.** Indicates the percentage of extra properties compared to the original (e.g., a gain of 0.4 on a given POI means it was complemented with 40% additional attribute values).
- **Fusion confidence**. Indicates the degree of similarity (in names, geometry, phone number, etc.) between the original features that were fused into a unified one, with values close to 1 indicating almost perfect match.
- **Average attribute gain.** This is the average *attribute gain* calculated from all the POIs that participated in the fusion process.
- **Average fusion confidence.** This is the average *fusion confidence* calculated from all the POIs that were fused during the whole process.
- These indicators are utilized both internally in FAGI (similarity measures, learning mechanisms) and as output for the end user, for further inspection and manual validation of the fused results.

4.1.5. Tracking changes in FAGI

As mentioned previously, in order to allow for tracking changes and their provenance, FAGI has been extended with a mechanism that records all such relevant value transformations. This is then provided as additional output, alongside the final merged dataset. Specifically, for each pair of interlinked POIs Pi in A and Pj in B, this additional output comprises statements including the following information:

- the identifiers of P_i and P_j;
- the name of the property on which the fusion action was applied;
- the fusion action that was applied;
- the original values of this property in each POI;
- the fused value that was produced as result;
- a confidence score for the action performed (if available).

Towards this, FAGI implements a logging mechanism that keeps track of the fusion actions it performs. This mechanism is optional and can be turned on or off by the user by setting the value of the parameter *verbose* to true or false, respectively, in FAGI's configuration file. This fusion log keeps track of the fusion actions applied on *each* attribute of *each* pair of linked POIs, as well as a fusion confidence score for each pair. The fusion log is written as a separate output file.

The fusion log contains detailed information about all the actions that were applied based on the rule specifications. Each rule defined for a specific attribute has its own record containing the initial values, the fusion action that was applied, and the final, fused value. The attributes without rule definitions also appear at the fused POI record, as they follow the default dataset action that was defined by the user.

4.1.6. Dataset output modes

FAGI v3.0 supports the modes enumerated in the following list, for outputting fused POI datasets. Note that the input of FAGI v3.0 remains fixed and is composed from two source POI datasets and a set of links that link POIs between the two datasets. Further, in all but one of the output modes, the user is required to select a "base" dataset, from the two input POI datasets (*the remaining input dataset is then denoted as "secondary"*). In most cases, this base dataset will be used as a basis for outputting the final, fused POI dataset.

1. **Handle only linked POIs – Include base dataset**. The pairs of linked POIs are fused into unified descriptions, and the resulting fused POI entities are written on the fused dataset. All unlinked POIs from the base dataset are also written on the fused dataset. All unlinked POIs from the secondary input dataset are discarded.
2. **Handle only linked POIs – Use a new dataset**. Only the results of the fusion process are written to the fused dataset. The new dataset contains unified descriptions only for pairs of POIs that where linked. Unlinked POIs from both input datasets are discarded.
3. **Handle both linked and unlinked POIs – Produce a single dataset**. The pairs of linked POIs are fused into unified descriptions, and the resulting fused POI entities are written on the fused dataset. All unlinked POIs from the base dataset are also written on the fused dataset. In contrast to the first mode, the unlinked triples of the secondary dataset are also written to the fused dataset. This way, the final, fused dataset contains fused POIs as prescribed from the links between the initial two input datasets, as well as **all** unlinked POIs from both input datasets.
4. **Handle both linked and unlinked POIs – Produce two datasets**. In this mode, two output datasets are produced. The first output dataset contains exactly the same data as the output dataset of mode 1. The second output dataset is produced by subtracting from the secondary dataset all the POIs (*with all their properties*) that participated in the fusion process. This way, the first output dataset represents the result of the fusion, while the second output dataset contains all the POIs that are not eventually included in the first output dataset.

4.2. Architecture

FAGI v3.0 is intended to be used for the efficient and accurate fusion of large volumes of linked POI entities. It can be executed as an autonomous service, or as part of an integration workflow within the SLIPO Workbench. FAGI comprises several modules that isolate different functionalities and allow their modular extension. Figure 18 provides an overview of FAGI's architecture.

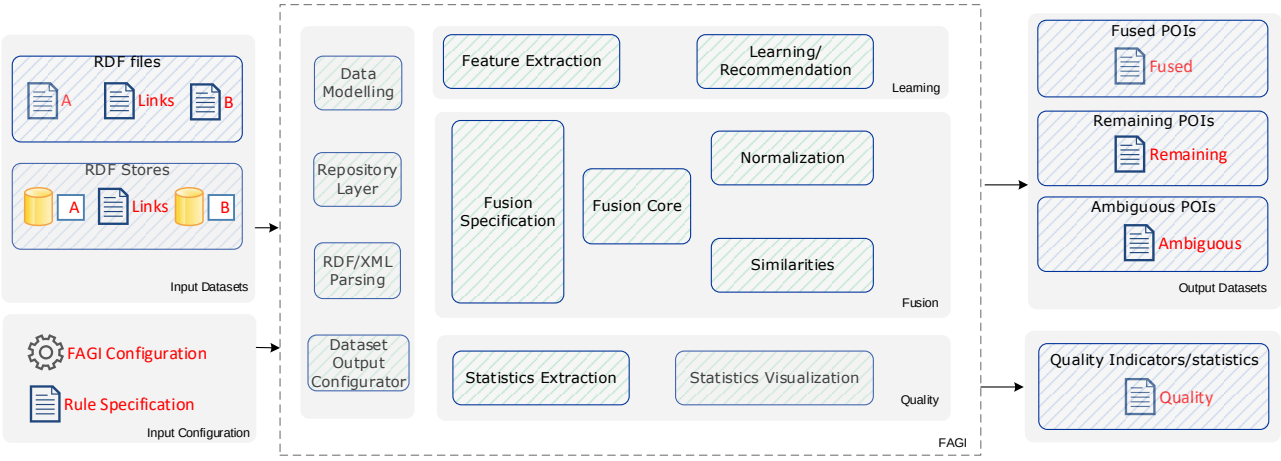


Figure 18: FAGI v3.0 Architecture

Based on the above architecture, we next describe in detail the input/output of FAGI, as well as the major components and modules it consists of.

4.2.1. Input and output

FAGI v3.0 accepts as input two POI datasets (A and B) in RDF format and a file containing *sameAs* links that connect POIs between the two datasets. Two input source types are supported, that essentially cover most usual input cases: RDF files and SPARQL endpoints. Regarding RDF files, the following RDF formats are supported:

- N-Triples (NT)
- Turtle (TTL)
- RDF/XML (RDF)
- RDF/XML (OWL)
- JSON-LD (JSONLD)
- RDF/JSON (RJ)
- TriG (TRIG)
- N-Quads (NQ)
- TriX (TRIX)

Additionally, FAGI receives as input an execution configuration and a rule specification containing link validation and fusion action rules. The structure of the configuration and rule specification files is described in Section 7.2. Examples of validation and fusion rule specifications are described in Figure 16 and Figure 17.

FAGI produces four different types of output files, depending on the configuration and the selected dataset output mode (see Section 4.1.6).

- **Fused POIs dataset**. It is the core output of FAGI and contains the fused POIs, as produced by the executed validation and fusion rule specifications. Depending on the configuration, the fused POI dataset may be produced in three variations:
    - **Plain**. In this version, the fused dataset does not include any additional POIs, apart from the fused POIs.
    - **Fused and unlinked from base**. In this version, the fused dataset also includes the remaining, unlinked POIs from one of the two input POI datasets, that serves as the "base" dataset.
    - **Fused and all unlinked**. In this version, the fused dataset also includes the remaining, unlinked POIs from both input POI datasets.
- **Remaining POIs dataset**. This dataset is optionally produced only if it is prescribed in the FAGI configuration by the user and mainly regards the case where the fused POIs dataset also contains the unlinked POIs from the base input POI dataset. In this scenario, the remaining, unlinked POIs from the secondary input POI dataset comprise the remaining POIs dataset. That is, this output dataset contains all the POIs from the secondary input POI dataset, except for the ones that where eventually fused.
- **Ambiguous POIs dataset**. This dataset is created in order to retain ambiguous POIs from the initial datasets, in case they are required for additional processing, after the examination/validation of the fused POIs. We note that the whole set of POI properties are retained, so that all information is available at a later integration stage.
- **Quality indicators/statistics dataset**. This dataset contains all the statistics and quality indicators extracted by FAGI, as prescribed in Section 4.1.4.

4.2.2. Core modules

FAGI v3.0 consists of four (4) high level components:

- **I/O Component**. This component consists of three main sub-modules. The module for reading input data and writing the output results, the module for parsing and validating the configuration and input files, and the repository layer. The repository layer serves as an in-memory object collection and encapsulates the functionality for querying and retrieving any necessary data during the fusion process.
- **Quality Component**. This component implements all the functionality for extracting statistics and indicators from both the input and the output data of FAGI as well as throughout the validation and fusion process.
- **Fusion Component**. This component handles the core, as well as auxiliary functionality related to fusion. This includes the definition and processing of rule specifications, the evaluation of conditions and expressions, the application of fusion actions, similarity calculations, normalization and comparison of property values, as well as the link validation process.

- **Learning Component**. This component is responsible for the learning mechanisms that allow training on previous actions and recommending fusion and validation actions for new pairs of linked POIs.

In what follows, the basic modules of FAGI v3.0 are briefly presented:

- **Fusion core**. It is the module that gathers all the core functionality for fusing POIs. It implements all available fusion and link validation actions, as well as all the functions that implement conditions on POI properties. The core is responsible for carrying out all the fusion processes based on the dataset output mode and producing the final output results.
- **Fusion Specification**. This module implements the functionality for defining rule specifications for link validation and fusion. The fusion specification supplies all necessary methods for validating input (either in XML format, or in the format produced by FAGI's software interface, which is consequently transformed to FAGI's XML syntax for rule specifications) and manages the process of parsing the XML configuration files as well as mapping and modelling them into in-memory data structures suitable for guiding the validation and fusion process.
- **Normalization**. This module gathers all pre-processing functionality that is executed on the values of several POI property types, with the aim to clean the data to some extent, as well as to align them in terms of format. It consists of several submodules, each one responsible for generic or more specific data types, depending on the objective of the specification.
- **Similarities**. This module implements a series of similarity functions, rules and thresholds that allow the comparison of POI property values of several types (e.g., addresses, names, phones). The functions that are implemented in this module are utilized in several others, such as in Fusion Specification, Normalization and Learning/Recommendation.
- **Feature Extraction**. This module realizes a series of functions for extracting features that represent pairs of linked POIs. These features include characteristics of property values of the POIs, as well as relations between these property values.
- **Learning/Recommendation**. This module implements several learning (classification) algorithms that are able to recommend: (a) whether a pair of linked POIs should be accepted (fused) or rejected; (b) which is the proper fusion action for a pair of matching properties between two linked POIs.
- **Statistics Extraction**. This module extracts statistics from the input and output data of FAGI. These statistics are destined both for exploration by the end user and for utilization in other modules of FAGI (e.g., Feature Extraction, Similarities).
- **Provenance Handler**. This module is essentially part of the fusion core and keeps track of all changes during the fusion process.
- **RDF Parsing**. This module handles parsing input RDF data (either from RDF files, or from a SPARQL endpoint) and loading them into RDF models in memory. The supported RDF formats are described in Section 4.2.1.
- **Data Modelling**. This module creates models and mappings for the input POIs and their links and encapsulates all the necessary functionality for accessing and handling the specification logic in order to support the actual validation and fusion of POI data.

- **Repository Layer**. Essentially, it provides an abstraction of data, so that the application can work with a simple abstraction that has an interface approximating that of a collection. Adding, removing, updating, and selecting items from this collection is done through a series of straightforward methods, without the need to deal with the data sources, e.g., files, SPARQL endpoints or in memory graphs.
- **Dataset Output Configurator**. This module handles the formatting of the output, fused POIs into a proper RDF format, according to SLIPO's ontology. Further, it realizes several options for the number and types of output datasets, as described in Sections 4.1.4.3 and 4.2.1.

4.2.3. Partitioning scheme

One of our main goals in the SLIPO project is to produce a commercial-level fusion software that can scale to tens of millions of POIs and efficiently execute batch fusion of large POI datasets. FAGI v3.0 implements a partitioning scheme for dividing the input POI datasets into a configurable number of subsets and distributing their processes into multiple *parallel* instances of FAGI.

The goal is to divide the two input POI datasets A and B into ($N_A \times N_B$) partitions that match a corresponding partition of the set of links between POIs. Specifically, we want to end up with a number of directories (*partition-directories*) of specific size (*calculated so that a single moderately size in terms of memory FAGI instance can handle them*). Each partition-directory should contain a set of FAGI input files (*dataset-partitions*): (i) a partition of the initial links file (*links-sublist*), (ii) a partition of dataset A containing all the POIs contained in the *links-sublist,* and (iii) a partition of dataset B containing all the POIs contained in the *links-sublist*.

The partitioning process consists in four main steps:

(a) **Partitioning the links file**. This step comes before the actual dataset partitioning and is straightforward, since the links file consists of independent triples (n-triples format) in the form of "<POI_a> <owl:sameAs> <POI_b>", where *a* and *b* are POIs belonging to datasets A and B respectively. The number of the initial partitions of the links file is calculated with a rough estimation of the initial file size and the available memory of each *FAGI-partitioner*. Then, each of these *links-lists* should be able to be managed by each *FAGI-partinioner*.

(b) **Creating *links-sublists* and mapping them to *partition-directories***. Each of the *links-lists* are loaded into different instances of *FAGI-partitioners* that are now able to load them in memory. Then, each *FAGI-partitioner* is responsible for further dividing the *links-list* to produce groups of subsets of the input datasets (dataset A, dataset B, links file) in a way that each group can fit in the provided memory of the machine that runs a *FAGI-instance*. Following an estimation of the partition size, the *FAGI-partitioner* divides the *links-list* into as many *links-sublists* as required and produces a *partition-directory* for each *links-sublist*, while serializing each *links-sublist* to an RDF file (n-triples format) inside that directory.

(c) **Mapping POI URIs of each link to a *dataset-partition* filename inside a *partition-directory*.** This process creates three mappings for each *links-sublist*: (i) the first one maps the URIs from dataset A to a corresponding *dataset-partition* filename using the *links-sublist* hashcode in order to keep track of which URIs belong to which *dataset-partition* and (ii) similarly, the second one maps the URIs

from the dataset B to the corresponding *dataset-partition* filename. These mappings are constructed for later use in the actual partitioning process. This process uses multimaps, as there is no guarantee that the URIs are unique between links (a URI from the dataset A could be linked with several different URIs from dataset B).

(d) **Partitioning of the source datasets**. Finally, the *FAGI-partitioner* is ready to partition the input RDF data. This step creates two threads, one for each input RDF file and the corresponding mappings from (c). Each thread starts streaming the input RDF file and, by utilizing the URI mappings, is able to match (and write) every triple in its corresponding *dataset-partition*.

The result of this process is a group of independent partitions of linked RDF data that are ready to be passed in multiple FAGI instances separately. At the end of the fusion process, the output of all FAGI instances are combined to a single (or more, depending on the fusion mode) output result.

4.3. Libraries and Frameworks

FAGI has dependencies to the following open-source tools/libraries:

- Apache Jena[41]: A Java framework for building Semantic Web applications.
- Java Topology Suite[42]: An API of 2D spatial predicates and functions conforming to the OGC Simple Features Specification for SQL.
- Google Guava[43]: A set of libraries that includes collection types (such as multimap and multiset), immutable collections, a graph library, functional types, an in-memory cache, and APIs/utilities for concurrency, I/O, hashing, primitives, reflection and string processing.
- Apache Commons Text[44]: A library focused on algorithms working on strings.
- Apache Commons Lang[45]: Provides a host of helper utilities for the java.lang API, notably String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and System properties. Additionally it contains basic enhancements to java.util.Date and a series of utilities dedicated to help with building methods, such as hashCode, toString and equals.
- Apache Lucene[46]: A high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.
- Google code json-simple[47]: A simple Java toolkit for JSON. It is used to encode or decode JSON text.

[41] https://jena.apache.org/

[42] https://github.com/locationtech/jts

[43] https://github.com/google/guava

[44] https://commons.apache.org/proper/commons-text/

[45] https://commons.apache.org/proper/commons-lang/

[46] https://lucene.apache.org/

[47] https://code.google.com/archive/p/json-simple/

- Apache Log4j 2[48]: A Java-based logging utility.
- Additionally, the UI component of FAGI is a Spring Boot application and uses the following libraries:
- Spring Boot[49]: Starter for building web, including RESTful, applications using Spring MVC. It uses Tomcat as the default embedded container.
- Spring Boot Developer Tools[50]: Additional set of tools that facilitate the application development experience. The spring-boot-devtools module can be included in any project to provide additional development-time features.
- Google Code Gson[51]: A Java serialization/deserialization library to convert Java Objects into JSON and back.
- JDOM 2[52]: A complete, Java-based solution for accessing, manipulating, and outputting XML data.
- Hamcrest[53]: A java library of Hamcrest matchers.
- WEKA[54]: The Waikato Environment for Knowledge Analysis (WEKA), a machine learning workbench.

4.4. License

FAGI is an open source software and is available from GitHub at [FAGI] (the core software) and at [FAGI-web] (the UI layer). It can be redistributed and/or modified under the terms of the Apache version 2.0 License as published by the Apache Software Foundation.

4.5. Documentation

A JavaDoc API documentation has been prepared in HTML format is publicly available at [FAGI-api]. Full documentation for the FAGI software is provided in [FAGI-doc].

4.6. Experimental Evaluation

In this section, we present the experiments that assess the efficiency and scalability of FAGI v3.0. In our experiments, we evaluate both the centralized version of FAGI, as well as its deployment using the distributed execution scheme (Section 4.2.3). First, we briefly present the POI datasets we used in our evaluation, that span from 100K links of POIs (~1.65M triples per dataset) to 90M links of POIs (~1.5 billion triples per dataset). Next, we compare the final version of FAGI (v3.0) with its initial (v1.0) and interim (v2.0)

[48] https://logging.apache.org/log4j/2.x/

[49] https://spring.io/projects/spring-boot

[50] https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-devtools/1.3.0.RELEASE

[51] https://github.com/google/gson

[52] https://mvnrepository.com/artifact/org.jdom/jdom2/2.0.6

[53] https://mvnrepository.com/artifact/org.hamcrest/hamcrest-core

[54] https://mvnrepository.com/artifact/nz.ac.waikato.cms.weka/weka-stable

versions, assessing the performance of the software in a centralized execution setting. Finally, we assess the scalability of FAGI v3.0 against v2.0, by measuring execution times when handling 1M and 10M links of POIs, using our custom distributed execution scheme. This experiment demonstrates that FAGI v3.0 can support world-scale fusion scenarios.

4.6.1. Datasets and measures

4.6.1.1. Datasets

In our experiments, we use the TomTom (dataset A) and Herold (dataset B) POI datasets for Austria, that are described in the Annex (Section 9.1). These initially include 312,385 and 350,053 POIs respectively. Since in these experiments we need to evaluate efficiency and scalability, we produced a subset (Version 1, 100K POIs) and then synthetically expanded these initial datasets (Version 2, 1.2M POIs; Version 3, ~10M POIs), creating in total three versions of each, presented in Table 3 (Versions 1, 2, and 3).

In addition, we used two synthetic planet dumps of the OpenStreetMap POIs (dataset A, dataset B) to examine the distributed execution of FAGI v3.0 created as described in Deliverable D1.4 'Final SLIPO System'. This was deemed necessary, since further synthetically scaling the TomTom and Herold datasets by two orders of magnitude was infeasible. The OSM dataset includes 90 million POIs for each version of the dataset and the interlinking process discovered up to 20 million unique links between the two. The links between the two datasets were created using LIMES (with very loose similarity thresholds, so that many POIs end up interlinked)[55].

Overall, the four produced versions allow us to evaluate the performance of FAGI in different fusion settings (*i.e., city-level, country-level, continent-level, global-level fusion respectively*). Version 1 of the datasets was used to compare the centralized version of FAGI v3.0, FAGI v2.0 and FAGI v1.0. Versions 2, 3 and 4 of the datasets were applied to evaluate the scalability of the distributed deployment of FAGI, since the centralized versions were (as expected) not able to handle the respective dataset sizes.

		Dataset A		Dataset B	
	Links	POIs	Triples	POIs	Triples
Version 1	100,000	100,000	1,507,659	100,000	1,881,912
Version 2	1,000,000	1,244,312	18,750,588	1,400,212	26,354,932
Version 3	10,000,000	10,888,737	164,082,950	12,251,855	230,605,655
Version 4 (OSM)	20,000,000	91,842,846	1,537,766,754	91,842,846	1,533,742,530

Table 3: Evaluation datasets

4.6.1.2. Measures

In the performed experiments, we measure execution times (*in seconds*), for several parts of the fusion process. Specifically, we report execution times for the following quantities-subtasks of the fusion process:

[55] These experiments regard efficiency and scalability; hence, ensuring that the utilized links between POIs are correct/accurate was not a concern.

- **Loading time**: Measures the time required by each version of FAGI in order to load the input datasets (two input POI dataset files and the corresponding links file).
- **Fusion time**: Measures the time required for FAGI to perform the actual fusion task, including (*depending on the version of the software*), rule specification parsing, transformation of the data to the proper internal formats, and writing the produced fused RDF triples into the output file.
- **Partitioning time** (distributed FAGI v2.0 and v.3.0): Reports the time required by our partitioning scheme to transform the initial input datasets into separate partitions to be parallelly processed by multiple FAGI instances.
- **Data transfer time** (distributed FAGI v2.0 and v3.0): Reports times measuring the data transfer overhead between nodes.
- **Total time**: The total execution time, including all applicable subtask times.

4.6.2. Results

The following Table 4 demonstrates the execution times of the different versions of FAGI on the four different versions of the input datasets, referenced by the corresponding number of links they contain (100K, 1M, 10M and 20M links respectively). We note that:

- When one of the above measures is not applicable in a specific execution combination, the measure value is marked as "-".
- When a version of FAGI did not manage to complete execution of a specific subtask, it is marked as *DNF*.

The first comparison regards the scenario of POI data fusion in a city-level (100K links). In this setting, we compare the centralized current version of FAGI (v3.0) with the initial version of the software and the 2.0 version. We can observe that, although FAGI v3.0 incorporates *much richer functionality*, it achieves fusion in *less than one third of the time* required by FAGI v1.0. These numbers demonstrate the effectiveness of the optimizations and code restructuring/enhancements we performed. The slight increase in execution time between FAGI v2.0 and v3.0 is negligible and anticipated, due to the further facilities incorporated into the newer version. Overall, FAGI v3.0 requires *only a bit more than two minutes* to perform fusion on 100K linked POIs, making it rather efficient for commercial dataset sizes.

The second comparison examines the setting of 1M links (*~45M triples in both input datasets in total*). In this setting, the distributed deployment mode of FAGI v2.0 and v3.0 is compared to the initial version of the software (v1.0). We note that FAGI v1.0 did not support a distributed execution mode and was depended on the Virtuoso RDF store, which was used for loading and querying the input data. In this experiment, we aim to assess the gains of developing and applying a custom partitioning and distributed processing scheme, against the initial, centralized version. Indeed, Table 4 shows that the distributed deployment of FAGI v3.0 is *more than 100x faster* than FAGI v1.0 in fusing 1M POIs. In the specific setting, FAGI occupies only ten nodes, each one running an individual FAGI instance. Further, in this setting FAGI v3.0 requires *less than five minutes* to perform fusion on 1M linked POIs, which corresponds to a country-level fusion process.

The third comparison examines the setting of 10M links *(~394M triples in both input datasets in total*). Again, the distributed deployment mode of FAGI v2.0 and v3.0 is compared to the initial version of the software

(v1.0). As expected, FAGI v1.0 cannot complete its execution after several hours of running. On the contrary, the distributed deployment of FAGI v2.0 and v3.0 requires ~24 min to fuse 10M POIs over 10 nodes. This is an acceptable runtime for the considered dataset sizes, taking also into account that fusion is a more fine-grained process, and thus, is usually performed individually in smaller scale datasets.

Finally, we used the OSM datasets to evaluate the scalability of FAGI on a world-scale setting. As seen in the results, FAGI v2.0 *fails* to complete this task after several hours of running, but FAGI v3.0 finishes the fusion task of 20 million links in an acceptable time (both over 32 nodes). The partitioning process takes much more time to execute than the previous examples, as the initial datasets are very much larger (~250Gb each).

Dataset and software versions	Partitioning time (sec)	Data transfer time (sec)	Loading time (sec)	Fusion time (sec)	Total time (sec)
100,000 links					
FAGI v1.0	-	-	19.6	455.3	474.9
FAGI v2.0 centralized	-	-	29.6	102.4	132.0
FAGI v3.0 centralized	-	-	33.1	107.3	140,4
1,000,000 links					
FAGI v1.0	-	-	899.2	26479.0	27378.2
FAGI v2.0 distributed	109.0**	18.3	30.8*	104.0*	262.1
FAGI v3.0 distributed	114.7**	21.2	32.2*	115.4*	283.5
10,000,000 links					
FAGI v1.0	-	-	DNF	DNF	DNF
FAGI v2.0 distributed	966.2**	141.8	69.8*	244.2*	1422.0
FAGI v3.0 distributed	952.8**	149.3	74.3*	262.4*	1438.8
20,000,000 links					
FAGI v1.0	-	-	-	-	-
FAGI v2.0 distributed	DNF	DNF	-	-	-
FAGI v3.0 distributed	12569.2**	410.6	112.4*	1678.9*	14771.1

Table 4: Runtimes (in sec.) of FAGI versions

*Max value from all nodes reported to represent the slowest processing node/FAGI instance, in a parallel processing setting.

** Partitioning times also include the runtimes for the inverse process, i.e. merging of the individual, fused files

5. LIMES Usage Manual

In this Section, we provide the usage manual for LIMES v1.7.0. First, we give details on building the application from the Java source code. Next, we provide instructions on both the manual- and machine-learning-based- configuration of LIMES. Finally, we present a short demonstration example on configuring and running LIMES.

5.1. Building Installation

LIMES v1.7 is publicly available (see https://github.com/SLIPO-EU/limes), offering the entire Java source code as well as indicative configurations. Java SDK 1.8[56] (or later) as well as Maven 3.3.3[57] (or later) must be installed and properly configured in order to compile and execute DEER. The `pom.xml` file contains the project's configuration in Maven and has been successfully tested in Mac OS, Microsoft Windows and Linux environments. The following building instructions assume that Git is also installed.

5.1.1. Generating Jar File (Headless)

In order to build the command line version from source, first the master branch of LIMES must be cloned to a preferred location by running:

```
git clone -b master --single-branch https://github.com/SLIPO-EU/LIMES.git LIMES
```

It is recommended to use the `--single-branch` parameter to save some time and avoid pulling the whole history of the project.

Then, from the root directory of the project (LIMES) the following command needs to be executed:

```
mvn clean install
```

Creating the runnable jar file including the dependencies use:

```
mvn clean package shade:shade -Dcheckstyle.skip=true -Dmaven.test.skip=true
```

After a successful installation, a target directory should have been created containing the `LIMES-VERSION-SNAPSHOT.jar` (version depending on POM configuration).

5.1.2. Generating Jar File (GUI)

Optionally, LIMES provides a graphical user interface (GUI) which helps the user build complicated link specifications without having to manually define rules in XML files. The GUI installation assumes that the command line version is already installed and uses the command-line version as a library.

Similarly, the LIMES-web repository needs to be cloned to a preferred location on your system by running:

```
git clone -b master --single-branch https://github.com/SLIPO-EU/LIMES.git LIMES
```

[56] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

[57] https://maven.apache.org/docs/3.3.3/release-notes.html

Switch to `limes-gui` and use:

```
mvn jfx:jar -Dcheckstyle.skip=true -Dmaven.test.skip=true
```

The `.jar` will be placed in `limes-gui/target/jfx/app/limes-GUI-VERSION-SNAPSHOT-jfx.jar`

The `limes-gui/target/jfx/app/lib` folder needs to be in the same folder as the `.jar` for the `.jar` to work.

5.2. Configuration Settings

A LIMES configuration file consists of ten parts, some of which are optional.

5.2.1. Metadata

The `metadata` tag always consists of the following bits of XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE LIMES SYSTEM "limes.dtd">
<LIMES>
```

5.2.2. Prefixes

Defining a prefix in a LIMES file demands setting two values: The `namespace` that will be addressed by the prefix's `label`

```
<PREFIX>
    <NAMESPACE>http://www.w3.org/1999/02/22-rdf-syntax-ns#</NAMESPACE>
    <LABEL>rdf</LABEL>
</PREFIX>
```

Here, we set the prefix `rdf` to correspond to `http://www.w3.org/1999/02/22-rdf-syntax-ns\.` A LIMES link specification can contain as many prefixes as required.

5.2.3. Data Sources

LIMES computes links between items contained in two Linked Data sources, dubbed source and target. Both source and target need to be configured using the same tags. An example of a configuration for both data sources is shown below.

```
<SOURCE>
    <ID>mesh</ID>
    <ENDPOINT>http://mesh.bio2rdf.org/sparql</ENDPOINT>
    <VAR>?y</VAR>
    <PAGESIZE>5000</PAGESIZE>
    <RESTRICTION>?y rdf:type meshr:Concept</RESTRICTION>
    <PROPERTY>dc:title</PROPERTY>
    <TYPE>sparql</TYPE>
</SOURCE>
```

```
<TARGET>
    <ID>linkedct</ID>
    <ENDPOINT>http://data.linkedct.org/sparql</ENDPOINT>
    <VAR>?x</VAR>
    <PAGESIZE>5000</PAGESIZE>
    <RESTRICTION>?x rdf:type linkedct:condition</RESTRICTION>
    <PROPERTY>linkedct:condition_name</PROPERTY>
</TARGET>
```

Six properties need to be set.

Each data source must be given an ID via the tag `ID`.

The endpoint of the data source needs to be explicated via the `ENDPOINT` tag.

If the data is to be queried from a SPARQL end point, the `ENDPOINT` tag must be set to the corresponding SPARQL endpoint URI.

In case the data is stored in a local file (CSV, N3, TURTLE, etc.), `ENDPOINT` tag must be set to the absolute path of the file containing the data.

The `VAR` tag describes the variable associated with the aforementioned endpoint. This variable is also used later, when specifying the metric used to link the entities retrieved from the source and target endpoints.

The fourth property is set via the `PAGESIZE` tag. This property must be set to the maximal number of triples returned by the SPARQL endpoint. For example, the *DBpedia endpoint*[58] returns a maximum of 1000 triples for each query. If `PAGESIZE` tag is set, the SPARQL query module in LIMES will be able to retrieve all relevant instances by iteratively asking the SPARQL end point for the relevant instances using the *offset* and *limit* features of the SPARQL query language. If the SPARQL endpoint does not limit the number of triples it returns or if the input is a file, the value of `PAGESIZE` should be set to -1.

The restrictions on the queried data can be set via the `RESTRICTION` tag. This tag allows to constrain the entries that are retrieved by the LIMES' query module. In this particular example, we only use instances of MESH concepts. Additionally, multiple `RESTRICTION` tags are allowed per data source.

The `PROPERTY` tag allows to specify the properties that will be used during the linking. It is important to note that the property tag can also be used to specify the pre-processing on the input data. For example, setting `rdfs:label AS nolang`, one can ensure that the language tags get removed from each `rdfs:label` before it is written in the cache. Pre-processing functions can be piped into one another by using `->`. For example, `rdfs:label AS nolang->lowercase` will compute `lowercase(nolang(rdfs:label))`. If you are not sure if all your entities have a certain property you can use the `OPTIONAL_PROPERTY` tag instead of `PROPERTY`. Additionally, multiple `PROPERTY` tags are allowed per data source.

Additionally, optional properties can be set to segment the requested dataset:

The graph of the endpoint can be specified directly after the `ENDPOINT` tag using the `GRAPH` tag.

The limits of the query can be set with the `MINOFFSET` and `MAXOFFSET` tags directly after the `PAGESIZE` tag. The resulting query will ask about the statements in the interval [`MINOFFSET`, `MAXOFFSET`]. Note that

[58] http://dbpedia.org/sparql

`MINOFFSET` must be smaller than `MAXOFFSET`! If both `SOURCE` and `TARGET` are restricted, a warning is generated.

5.2.3.1. Pre-processing Functions

Currently, LIMES supports the following set of pre-processing functions:

- `noLang` for removing language tags
- `lowerCase` for converting the input string into lower case
- `upperCase` for converting the input string into upper case
- `number` for ensuring that only the numeric characters, "." and "," are contained in the input string
- `replace(String a,String b)` for replacing each occurrence of `a` with `b`
- `regexReplace(String x,String b)` for replacing each occurrence the regular expression `x` with `b`
- `cleanIri` for removing all the prefixes from IRIs
- `celsius` for converting Fahrenheit to Celsius
- `fahrenheit` for converting Celsius to Fahrenheit
- `removeBraces` for removing the braces
- `regularAlphabet` for removing nun-alphanumeric characters
- `uriAsString` returns the last part of an URI as a String.

Sometimes, generating the right link specification might either require merging property values (for example, the `dc:title` and `foaf:name` of MESH concepts) or splitting property values (for example, comparing the label and `foaf:homepage` of source instances and the `foaf:homepage` of target instances as well as `foaf:homepage AS cleanIri` of the target instances with the `rdfs:label` of target instances. To enable this, LIMES provides the `RENAME` operator which simply stores either the values of a property or the results of a pre-processing function into a different property field. For example, `foaf:homepage AS cleanIri RENAME label` would store the homepage of an object without all the prefixes in the name property. The user could then access this value during the specification of the similarity measure for comparing source and target instances. Note that the same property value can be used several times. Thus, the following specification fragment is valid and leads to the `dc:title` and `foaf:name` of individuals) of MESH concepts being first cast down to the lowercase and then merged to a single property.

```
<SOURCE>
    <ID>mesh</ID>
    <ENDPOINT>http://mesh.bio2rdf.org/sparql</ENDPOINT>
    <VAR>?y</VAR>
    <PAGESIZE>5000</PAGESIZE>
    <RESTRICTION>?y rdf:type meshr:Concept</RESTRICTION>
    <PROPERTY>dc:title AS lowercase RENAME name</PROPERTY>
    <PROPERTY>foaf:name AS lowercase RENAME name</PROPERTY>
    <TYPE>sparql</TYPE>
</SOURCE>
```

In addition, the following allows splitting the values of `foaf:homepage` into the property values `name` and `homepage`.

```
<SOURCE>
```

```
    <ID>mesh</ID>
    <ENDPOINT>http://mesh.bio2rdf.org/sparql</ENDPOINT>
    <VAR>?y</VAR>
    <PAGESIZE>5000</PAGESIZE>
    <RESTRICTION>?y rdf:type meshr:Concept</RESTRICTION>
    <PROPERTY>foaf:homepage AS lowercase RENAME homepage</PROPERTY>
    <PROPERTY>foaf:homepage AS cleaniri->lowercase RENAME name</PROPERTY>
    <TYPE>sparql</TYPE>
</SOURCE>
```

In addition, a source type can be set via `TYPE`. The default type is set to `SPARQL` (for a SPARQL endpoint) but LIMES also supports reading files directly from the hard-drive. The supported data formats are:

- `CSV:` Character-separated file can be loaded directly into LIMES. Note that the separation character is set to `TAB` as a default. The user can alter this setting programmatically.
- `N3` (which also reads `NT` files) reads files in the `N3` language.
- `N-TRIPLE` reads files in W3C's core N-Triples format[59].
- `TURTLE` allows reading files in the `Turtle` syntax[60].

Moreover, if you want to download data from a SPARQL endpoint, there is no need to set the `<TYPE>` tag. Instead, if you want to read the source (or target) data from a file, you should fill `<ENDPOINT>` tag with the absolute path of the input file, e.g. `<ENDPOINT>C:/Files/dbpedia.nt</ENDPOINT>`, and you should also set the `<TYPE>` tag with the type of the input data, for example `<TYPE>NT</TYPE>`.

5.2.4. Metric Expression for Similarity Measurement

One of the core improvements of the newest LIMES kernels is the provision of a highly flexible language for the specification of complex metrics for linking (set by using the `METRIC` tag as exemplified below).

```
<METRIC>
    trigrams(y.dc:title, x.linkedct:condition_name)
</METRIC>
```

In this example, we use the `trigrams` metric to compare the `dc:title` of the instances retrieved from the *source* data source (with which the variable `y` is associated) with the `linkedct:condition` (with which the variable `x` is associated). While such simple metrics can be used in many cases, complex metrics are necessary in complex linking cases. LIMES includes a formal grammar for specifying complex configurations of arbitrary complexity. For this purpose, two categories of binary operations are supported: Metric operations and Boolean operations.

5.2.4.1. Metric operations

Metric operations allow to combine metric values. They include the operators `MIN`, `MAX` and `ADD` e.g. as follows:

```
MAX(trigrams(x.rdfs:label,y.dc:title)|0.3,
    euclidean(x.lat|long,y.latitude|longitude)|0.5).
```

[59] http://www.w3.org/TR/rdf-testcases//#ntriples

[60] http://www.w3.org/TR/turtle

This specification computes the maximum of:

1. The trigram similarity of x's `rdfs:label` and y's `dc:title` filtered by 0.3. i.e. LIMES only returns resources with trigram similarities above or equal 0.3.
2. The 2-dimension Euclidean distance of `x`'s `lat` and `long` with `y`'s `latitude` and `longitude`, i.e., `((x.lat-y.latitude)2+(x.long-y.longitude)2) \sqrt{((x.lat- y.latitude)^2 + (x.long - y.longitude)^2)} √((x.lat-y.latitude)2+(x.long-y.longitude)2.` Note that, Like in the case of trigrams, LIMES filter the results for Euclidian similarities greater or equal to 0.5.

Note that the Euclidean distance supports arbitrarily many dimensions. In addition, note that ADD allows to define weighted sums as follows:

```
ADD(0.3*trigrams(x.rdfs:label,y.dc:title)|0.3,
    0.7*euclidean(x.lat|x.long,y.latitude|y.longitude)|0.5).
```

We call `trigrams(x.rdfs:label,y.dc:title)|0.3` the left child of the specification and `euclidean(x.lat|long, y.latitude|longitude)|0.5` the right child of the specification. Both children specifications are simple specifications and combined with a metric operator, they create a complex specification. LIMES gives the user the opportunity to combine **exactly two LSs** (complex or simple) in order to create a new complex LS.

5.2.4.2. Boolean operations

Boolean operations allow to combine and filter the results of metric operations and include `AND`, `OR`, `DIFF`, e.g. as

```
AND(trigrams(x.rdfs:label,y.dc:title)|0.9,
    euclidean(x.lat|x.long,y.latitude|y.longitude)|0.7).
```

This specification returns all links such that:

1. the trigram similarity of `x`'s `rdfs:label` and `y`'s `dc:title` is greater or equal to 0.9 and
2. the 2-dimension Euclidean distance of `x`'s `lat` and `long` mit y's `latitude` and `longitude` is greater or equal to `0.7`.

We call `trigrams(x.rdfs:label,y.dc:title)|0.9` the left child of the specification and `euclidean(x.lat|x.long, y.latitude|y.longitude)|0.7` the right child of the specification. Both children specifications are simple specifications and combined with a Boolean operator, they create a complex specification. LIMES gives the user the opportunity to combine **exactly two** complex or simple speciation's to create a new complex specification. Note that each child specification must be accompanied by its own threshold.

5.2.4.3. Measure Packages

Measures are organized in packages, based on the type of resource they are designed to operate with. Several measure packages ship with LIMES, while and it is easy to extend it with custom packages from third parties.

The current version of LIMES ships with the following measure packages include:

1. *String measures* for computing the similarity values among string representations of POI resources. For example, string measures can be used to compare name of POI resources modelled using `rdf:label`.
2. *Vector space measures* for computing the proximity among point representations of POI resources. For example, find the nearest bus station for each hospital.
3. Point-set *measures* for computing the distance among point-set representations of POI resources. For example, it can be used to find the nearest hospital for each kindergarten.
4. Topological *measures* for computing the topological relation among geospatial representations of POI resources. For example, it can be used to find car park placed which located within shopping malls.
5. Temporal *measures* for computing the temporal relations among time stamps of POI resources. For example, find restaurants with the same opening hours.

More complex distance measures are being added continuously. We give more details about each of the measure type in the following sections.

5.2.4.3.1. String Measures

The string measures package consists of the following measures:

`Cosine:` Cosine string similarity is a measure of similarity between two non-zero vectors representations of the two input strings of an inner product space that measures the cosine of the angle between them. The outcome of the Cosine string similarity is neatly bounded in [0,1]

`ExactMatch:` Exact match string similarity is a measure of similarity between two input strings that returns one in case the two input strings were identical, zero otherwise.

- `Jaccard:` The *Jaccard index*, also known as Intersection over Union and the *Jaccard similarity coefficient* (originally coined coefficient de communauté by Paul Jaccard), is a statistic used for comparing the similarity and diversity of sample sets. The Jaccard coefficient measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets. In LIMES, we use trigrams of the input strings to generate our sample sets.
- `Overlap:` The *overlap coefficient* or *Szymkiewicz–Simpson coefficient*, is a similarity measure that measures the overlap between two sets. It is related to the *Jaccard index* and is defined as the size of the intersection divided by the smaller of the size of the two sets.
- `Jaro:` The *Jaro distance* between two strings is the minimum number of single-character transpositions required to change one string into the other.
- `JaroWinkler:` The *Jaro–Winkler distance* is a string metric for measuring the edit distance between two sequences. It is a variant proposed in 1990 by William E. Winkler of the Jaro distance metric. The Jaro–Winkler distance uses a prefix scale which gives more favourable ratings to strings that match from the beginning for a set prefix length. The lower the Jaro–Winkler distance for two strings is, the more similar the strings are. The score is normalized such that 0 equates to no similarity and 1 is an exact match. The *Jaro–Winkler similarity* in LIMES is given by 1 / (1 – Jaro–Winkler distance).

- `Levenshtein:` The *Levenshtein distance* is a string metric for measuring the difference between two strings. Informally, the *Levenshtein distance* between two strings is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. It is named after the Soviet mathematician Vladimir Levenshtein, who considered this distance in 1965. Normalized *Levenshtein distance* is computed by dividing the *Levenshtein distance* by the length of the input string. The *Levenshtein similarity* in LIMES is given by 1 / (1 – normalized-*Levenshtein distance*).
- `MongeElkan:` The *Monge-Elkan similarity* measure is a type of hybrid similarity measure that combines the benefits of sequence-based and set-based methods. This can be effective for domains in which more control is needed over the similarity measure. In LIMES, we use trigrams of the input strings to generate our sample string subsets. In LIMES, we use trigrams of the input strings to generate our sample sets.
- `RatcliffObershelp` In Ratcliff/Obershelp, we compute the similarity of the two input strings as the number of matching characters divided by the total number of characters in the two strings. Matching characters are those in the longest common subsequence plus, recursively, matching characters in the unmatched region on either side of the longest common subsequence.
- `Soundex:` Soundex is a phonetic algorithm for indexing names by sound, as pronounced in English. The goal is for homophones to be encoded to the same representation so that they can be matched despite minor differences in spelling. The algorithm mainly encodes consonants, a vowel will not be encoded unless it is the first letter. In LIMES, we compute the Soundex distance as the reverse of the distance between the encoding of the two input strings.
- `Trigram`: A tri-gram is a group of three consecutive characters taken from a string. In LIMES, we measure the similarity of two input strings by counting the number of trigrams they share. Formally, we compute the trigram similarity as the normalized sum of absolute differences between tri-gram vectors of both the input strings.
- `Qgrams:` Same as trigram but using a group of four three consecutive characters for generating the q-gram vectors of the input strings.

Below, an example of an atomic LS that consists of the string measure `Trigram` and a threshold `theta = 0.8` is given:

```
trigram(x.label, y.title) | 0.8
```

where `label` and `title` are properties of the source and target KB respectively, whose values are strings.

5.2.4.3.2. Vector Space Measures

LIMES supports comparing numeric vectors representations of POI resources by using the vector space measures package consisting of the following measures:

- `Euclidean` *Euclidean* metric is the straight-line distance between two points in *Euclidean* space. With this distance, Euclidean space becomes a metric space. For example:

```
euclidean(a.wgs84:lat|wgs84:long,b.wgs84:lat|wgs84:long)
```

will compute the *Euclidean distance* between the point representations of each resource from the source and target datasets.

- `Geo_Orthodromic` The *great-circle distance* or *orthodromic distance* is the shortest distance between two points on the surface of a sphere, measured along the surface of the sphere (as opposed to a straight line through the sphere's interior). The distance between two points in *Euclidean* space is the length of a straight line between them, but on the sphere, there are no straight lines. In spaces with curvature, straight lines are replaced by geodesics. Geodesics on the sphere are circles on the sphere whose centres coincide with the canter of the sphere and are called great circles.
- `Geo_Great_Elliptic` The great ellipse distance is the length of the ellipse passing through two points on a spheroid and having the same centre as that of the spheroid. Equivalently, it is distance of the ellipse on the surface of a spheroid and cantered on the origin, or the curve formed by intersecting the spheroid by a plane through its centre. The great ellipse distance is confedered the most accurate distance between two point In the surface of the earth.

5.2.4.3.3. Point-Set Measures

The similarity between POI geometries can be measured by using the following point-set distances:

- `Geo_Hausdorff` The *Hausdorff* distance is a measure of the maximum of the minimum distances between pairwise points in the two input geometries.
- `Geo_Max` The idea behind this measure is to compute the overall maximal distance between pairwise points of the two input geometries.
- `Geo_Min` The idea of `Geo_Min` is akin to that of `Geo_Max` and is defined as minimal distance between pairwise points of the two input geometries.
- `Geo_Mean` The mean distance is one of the most efficient distance measures for point sets. First, a mean point is computed for each point set. Then, the distance be- tween the two means is computed by using the orthodromic distance.
- `Geo_Avg` For computing the average point sets distance function, the orthodromic distance measures between all the source-target geometries' points pairs is cumulated and divided by the number of points in the source-target geometries' point pairs.
- `Geo_Frechet` The Fréchet distance is a measure of similarity between curves (in our case geometries representations of the POI resources) that takes into account the location and ordering of the points along the curves.
- `Geo_Sum_Of_Min` First, the closest point from the source geometry to each point to the target geometry is computed. The same operation is carried out with source and target reversed. Finally, the average of the two values is then the distance value.
- `Geo_Naive_Surjection` The surjection distance function introduced defines the distance between two geometries as the minimum distance between the sum of distances of the surjection of the larger set to the smaller one. A main drawback of the surjection is being biased toward some points ignoring some others in calculations.
- `Geo_Fair_Surjection` In order to fix the bias of the `Geo_Naive_Surjection`, the fair-surjection distance maps the elements of source geometry as evenly as possible to the elements of the target geometry.

- `Geo_Link The Link` distance is defined as the minimum orthodromic distance between pairwise points of the source and target geometries that satisfy the bijection relation.

5.2.4.3.4. Topological Measures

The topological relations between spatial representations of POI resources can be found by using the following relations. In the following relations we assume that the first POI resource has a geospatial representation in a form of geometry *a* and the target POI resource has a geospatial representation in a form of geometry *b*:

- `Top_Contains` A geometry *a* contains geometry *b* if and only if no points of b lie in the exterior of a, and at least one point of the interior of b lies in the interior of a.
- `Top_Covers` A geometry *a* covers geometry *b* if and only if the geometry *b* lies in *a*. i.e. No points of *b* lie in the exterior of *a*, or Every point of *b* is a point of (the interior or boundary of) *a*.
- `Top_Covered_By` A geometry *a* is covered by a geometry *b* if and only if every point of *a* is a point of *b*, and the interiors of the two geometries have at least one point in common. Note that `Top_Covered_By` is the reverse relation of `Top_Covers`.
- `Top_Crosses` A geometry *a* crosses a geometry *b* if and only if they have some but not all interior points in common, and the dimension of the intersection is less than that of at least one of them.
- `Top_Disjoint` Two geometries *a* and *b* are disjoint if and only if they have no point in common. They form a set of disconnected geometries.
- `Top_Equals` Two geometries *a* and *b* are topologically equal if their interiors intersect and no part of the interior or boundary of one geometry intersects the exterior of the other.
- `Top_Intersects` A geometry *a* intersects A geometry *b* if and only if geometries a and b have at least one point in common.
- `Top_Overlaps` A geometry *a* overlaps a geometry *b* if and only if they have some but not all points in common, they have the same dimension and the intersection of the interiors of the two geometries has the same dimension as the geometries themselves
- `Top_Touches` Two geometries *a* and touched if they have at least one boundary point in common, but no interior points.
- `Top_Within` A geometry *a* is within a geometry *b* if and only if a lies in the interior of b.

5.2.4.3.5. Temporal Measures

The temporal relations between POI resources can be found by using the following relations:

- `Tmp_Concurrent`: given a source and a target KB, `Tmp_Concurrent` links the source and the target events that have the same begin date and were produced by the same machine. For example: `Tmp_Concurrent(x.beginDate1|machine1,y.beginDate2|machine2)|1.0`
- `Tmp_Predecessor:` given a source and a target KB, `vmp_Predecessor` links the source events to the set of target events that happen exactly before them. For example: `Tmp_Predecessor (x.beginDate1, y.beginDate2)|1.0`. If the `Tmp_Predecessor` measure is used in a complex LS, the `CANONICAL` planner should be used.

- `Tmp_Successor`: given a source and a target KB, `Tmp_Successor` links the source events to the set of target events that happen exactly after them. For example: Tmp_Successor `(x.beginDate1, y.beginDate2)|1.0`. If the `Tmp_Successor` measure is used in a complex LS, the `CANONICAL` planner should be used.

Moreover, LIMES support the following temporal relations between POI resources based on *Allen's* algebra:

- `Tmp_After` The first POI takes place *after* the second POI takes place.
- `Tmp_Before` The first POI takes place *before* the second POI takes place.
- `Tmp_During` The first POI take place *during* the second POI takes place.
- `Tmp_During_Reverse` The second POI take place *during* the first POI takes place. `Tmp_During_Reverse` is the reverse of `Tmp_During`
- `Tmp_Equals` Both first POI and the second take place concurrently. i.e. both POIs have *equal* timestamp.
- `Tmp_Finishes` The first POI *finishes in the same time* as the second POI finishes.
- `Tmp_Is_Finished_By` reverse of `Tmp_Finishes`
- `Tmp_Overlaps` Part of the first POI timestamp *overlaps* with the second POI time stamp.
- `Tmp_Is_Overlapped_By` reverse of `Tmp_Overlaps`
- `Tmp_Starts` The start first POI timestamp is tha same as the start of the second POI time stamp.
- `Tmp_Is_Started_By` reverse of `Tmp_Starts`
- `Tmp_Meets` The end first POI timestamp meets the start of the second POI time stamp.
- `Tmp_Is_xBy` reverse of `Tmp_Meets`

Below, an example of an atomic LS that consists of the temporal measure `Tmp_Finishes` and a threshold `theta = 1.0` is given:

```
Tmp_Finishes(x.beginDate1|endDate1, y.beginDate2|endDate2) | 0.8
```

where `beginDate1` and `beginDate2` are properties of the source and target KB respectively, whose values indicate the begin of a temporal event instance and `endDate1` and `endDate2` are properties of the source and target KB respectively, whose values indicate the end of a temporal event instance. Both begin and end properties for both source and target MUST be included in an atomic LS whose measure is temporal. Also, the acceptable values for all properties are in the format: `2015-04-22T11:29:51+02:00`.

5.2.5. Execution (optional)

Additional fine-tuning parameters can be set using the `Execution` tag. We recommend LIMES users to use the default parameters for each of the execution parameters. An advanced user needs to consult the developer's manual[61] for correct setting of the optional parameters. Three LIMES execution parameters can be set here:

- `REWRITER`: LIMES 1.0.0 implements the `DEFAULT` rewriter. **(recommended)**

[61] https://dice-group.github.io/LIMES/developer_manual/

- `PLANNER:` the user can choose between:
  - `CANONICAL:` It generates an immutable plan in a static manner.
  - `HELIOS:` It generates an immutable plan in a dynamic manner.
  - `DYNAMIC:` It generates a mutable plan in a dynamic manner.
  - `DEFAULT:` same as `CANONICAL`. **(recommended)**
- `ENGINE:` the user can choose between:
  - `SIMPLE:` It executes each independent part of the plan sequentially.
  - `DEFAULT:` same as `SIMPLE`. **(recommended)**
  - if not set, the `DEFAULT` value for each parameter will used be will

5.2.6. Machine Learning

In most cases, finding a good metric expression (i.e. one that achieves high F-Measure in interlinking POI entities) is not a trivial task. Therefore, in LIMES we implemented a number of machine learning algorithms for auto-generation of mappings among POI resources. For using a machine learning algorithm in your configuration file use the `MLALGORITHM` tag instead of the `METRIC` tag. For example:

```
<MLALGORITHM>
    <NAME>wombat simple</NAME>
    <TYPE>supervised batch</TYPE>
    <TRAINING>trainingData.nt</TRAINING>
    <PARAMETER>
        <NAME>max execution time in minutes</NAME>
        <VALUE>60</VALUE>
    </PARAMETER>
</MLALGORITHM>
```

In the above example, the following apply:

- The tag `NAME` contains the name of the machine learning algorithm. Currently, we have implemented the following algorithms:
  - `wombat simple`: First the `wombat simple` algorithm generates a set of initial atomic LSs. Then, it uses the three logical connectors ⊔, ⊓ and \ to append further atomic LS. `wombat simple` uses a naïve operator for LS generation based of concept of the generalisation via an upward refinement operator to traverse the space of link specification. The naïve refinement operator used by `wombat simple` is not complete (i.e., not able to reach all possible LSs). Our experiments show that `wombat simple` is faster and as accurate as the `wombat complete`.
  - `wombat complete`: Same as wombat simple but uses a more sophisticated complete refinement operator to be able to reach all the possible LSs. Our experiments show that `wombat complete` is slower and as accurate as the `wombat simple`.
  - `Eagle`: EAGLE is a genetic-algorithm-based approach for link specification learning. `EAGLE` is non-deterministic (i.e. running EAGLE for the same setting many times generates different results).

- The tag `TYPE` contains the type of the machine learning algorithm, which could take one of the values:
  - `supervised batch`: The user provides a file contains number of labelled examples for the different ML algorithms to learn from.
  - `supervised active`: The user iteratively will be asked by the ML algorithm for labelling feedback (i.e., user will be asked to mark one of more example link(s) as positive or negative at each iteration run of the ML algorithm).
  - `Unsupervised`: No training data needed from the user. Note that unsupervised linking is currently supported only for `owl:sameAs` links.
- The tag `TRAINING` contains the full path to the training data file. Note that this tag is not required in case of the supervised active and unsupervised learning algorithms. Usually a `*.ttl` file is used where the URIs of the entities are linked via `owl:sameAs` e.g.: `<http://sourceexample.org/entity1> <http://www.w3.org/2002/07/owl#sameAs><http://targetexample.org/entity1>`
- The tag `PARAMETER` contains the name (using the sub-tag `NAME`) and the value (using the sub-tag `VALUE`) of the used machine learning algorithm parameter. User can use as many `PARAMETER` tags as it is required. Note that LIMES uses the default values of all unspecified parameters.

Table 5 contains a list of implemented algorithms together with supported implementations and parameters.

Table 5: ML algorithms in LIMES

ML Algorithm	Supported types	Parameter	Default Value	Note
WOMBAT Simple	`supervised batch, supervised active and unsupervised`	`max refinement tree size`	2000	
		`max iterations number`	3	
		`max iteration time in minutes`	20	
		`max execution time in minutes`	600	
		`max fitness threshold`	1	Range 0 to 1
		`minimum property coverage`	0.4	Range 0 to 1
		`property learning rate`	0.9	Range 0 to 1
		`overall penalty weight`	0.5	Range 0 to 1

		children penalty weight	1	Range 0 to 1
		complexity penalty weight	1	Range 0 to 1
		verbose	false	
		atomic measures	jaccard, trigrams, cosine, qgrams	
		save mapping	true	
WOMBAT Complete	supervised batch, supervised active and unsupervised	Same as WOMBAT Simple		
EAGLE	supervised batch, supervised active and unsupervised	generations	10	Integer
		preserve_fittest	true	
		max_duration	60	[1,Inf)
		inquiry_size	10	[1,Inf)
		max_iterations	500	[1,Inf)
		max_quality	0.5	[0.0,1.0]
		termination_criteria	iteration	enum
		termination_criteria_value	0.0	[0.0,Inf)
		beta	1.0	[0.0,1.0]
		population	20	[1,Inf)
		mutation_rate	0.4	[0.0,1.0]
		reproduction_rate	0.4	[0.0,1.0]
		crossover_rate	0.3	[0.0,1.0]

5.2.7. Granularity (optional)

The user can choose positive integers to set the granularity of hyperspace tiling approaches used by LIMES by setting

```
<GRANULARITY>2</GRANULARITY>.
```

Our evaluations show that the default granularity of 2 achieves the best performance for most of the datasets. Therefore we recommend LIMES' users not to change the default value. An advanced user needs to consult the developer's manual[62] for correct setting of the granularity parameters.

5.2.8. Acceptance Condition

Filling the acceptance condition consists of setting the threshold value to the minimum value that two instances must have in order to satisfy a relation. This can be carried out as exemplified below.

```
<ACCEPTANCE>
    <THRESHOLD>0.98</THRESHOLD>
    <FILE>accepted.nt</FILE>
    <RELATION>owl:sameAs</RELATION>
</ACCEPTANCE>
```

By using the `THRESHOLD` tag, the user can set the minimum value that two instances must have in order to satisfy the relation specified in the `RELATION` tag, i.e., `owl:sameAs` in our example. Setting the tag `<FILE>` allows to specify where the links should be written. Currently, LIMES produces output files in the N3 format.Future versions of LIMES will allow to write the output to other streams and in other data formats.

5.2.9. Review Condition

Setting the condition upon which links must be reviewed manually is very similar to setting the acceptance condition as shown below.

```
<REVIEW>
    <THRESHOLD>0.95</THRESHOLD>
    <FILE>reviewme.nt</FILE>
    <RELATION>owl:sameAs</RELATION>
</REVIEW>
```

All instances that have a similarity between the threshold set in `REVIEW` (0.95 in our example) and the threshold set in `ACCEPTANCE` (0.98 in our example) will be written in the review file and linked via the relation set in `REVIEW`.

The LIMES configuration file should be concluded with `</LIMES>`

5.2.10. Output Format

The user can choose between `TAB` (for CSV records) and `N3` (for RDF triples) as output format by setting:

```
<OUTPUT>N3</OUTPUT>
```

5.3. Configuration File Examples

The following listing shows the whole configuration file for LIMES explicated in the sections above.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE LIMES SYSTEM "limes.dtd">
<LIMES>
    <PREFIX>
```

[62] https://dice-group.github.io/LIMES/developer_manual/

```
        <NAMESPACE>http://geovocab.org/geometry#</NAMESPACE>
        <LABEL>geom</LABEL>
    </PREFIX>
    <PREFIX>
        <NAMESPACE>http://www.opengis.net/ont/geosparql#</NAMESPACE>
        <LABEL>geos</LABEL>
    </PREFIX>
    <PREFIX>
        <NAMESPACE>http://linkedgeodata.org/ontology/</NAMESPACE>
        <LABEL>lgdo</LABEL>
    </PREFIX>
    <SOURCE>
        <ID>linkedgeodata</ID>
        <ENDPOINT>http://linkedgeodata.org/sparql</ENDPOINT>
        <VAR>?x</VAR>
        <PAGESIZE>2000</PAGESIZE>
        <RESTRICTION>?x a lgdo:RelayBox</RESTRICTION>
        <PROPERTY>geom:geometry/geos:asWKT RENAME polygon</PROPERTY>
    </SOURCE>
    <TARGET>
        <ID>linkedgeodata</ID>
        <ENDPOINT>http://linkedgeodata.org/sparql</ENDPOINT>
        <VAR>?y</VAR>
        <PAGESIZE>2000</PAGESIZE>
        <RESTRICTION>?y a lgdo:RelayBox</RESTRICTION>
        <PROPERTY>geom:geometry/geos:asWKT RENAME polygon</PROPERTY>
    </TARGET>
    <METRIC>geo_hausdorff(x.polygon, y.polygon)</METRIC>
    <ACCEPTANCE>
        <THRESHOLD>0.9</THRESHOLD>
        <FILE>lgd_relaybox_verynear.nt</FILE>
        <RELATION>lgdo:near</RELATION>
    </ACCEPTANCE>
    <REVIEW>
        <THRESHOLD>0.5</THRESHOLD>
        <FILE>lgd_relaybox_near.nt</FILE>
        <RELATION>lgdo:near</RELATION>
    </REVIEW>

    <EXECUTION>
        <REWRITER>default</REWRITER>
        <PLANNER>default</PLANNER>
        <ENGINE>default</ENGINE>
    </EXECUTION>

    <OUTPUT>TAB</OUTPUT>
</LIMES>
```

The following configuration file uses the machine learning algorithm of the Wombat simple to find the metric expression for the same example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE LIMES SYSTEM "limes.dtd">
<LIMES>
    <PREFIX>
        <NAMESPACE>http://geovocab.org/geometry#</NAMESPACE>
        <LABEL>geom</LABEL>
    </PREFIX>

    <PREFIX>
        <NAMESPACE>http://www.opengis.net/ont/geosparql#</NAMESPACE>
```

```
        <LABEL>geos</LABEL>
    </PREFIX>

    <PREFIX>
        <NAMESPACE>http://linkedgeodata.org/ontology/</NAMESPACE>
        <LABEL>lgdo</LABEL>
    </PREFIX>

    <SOURCE>
        <ID>linkedgeodata</ID>
        <ENDPOINT>http://linkedgeodata.org/sparql</ENDPOINT>
        <VAR>?x</VAR>
        <PAGESIZE>2000</PAGESIZE>
        <RESTRICTION>?x a lgdo:RelayBox</RESTRICTION>
        <PROPERTY>geom:geometry/geos:asWKT RENAME polygon</PROPERTY>
    </SOURCE>

    <TARGET>
        <ID>linkedgeodata</ID>
        <ENDPOINT>http://linkedgeodata.org/sparql</ENDPOINT>
        <VAR>?y</VAR>
        <PAGESIZE>2000</PAGESIZE>
        <RESTRICTION>?y a lgdo:RelayBox</RESTRICTION>
        <PROPERTY>geom:geometry/geos:asWKT RENAME polygon</PROPERTY>
    </TARGET>

    <MLALGORITHM>
        <NAME>wombat simple</NAME>
        <TYPE>supervised batch</TYPE>
        <TRAINING>trainingData.nt</TRAINING>
        <PARAMETER>
            <NAME>max execution time in minutes</NAME>
            <VALUE>60</VALUE>
        </PARAMETER>
    </MLALGORITHM>

    <ACCEPTANCE>
        <THRESHOLD>0.9</THRESHOLD>
        <FILE>lgd_relaybox_verynear.nt</FILE>
        <RELATION>lgdo:near</RELATION>
    </ACCEPTANCE>

    <REVIEW>
        <THRESHOLD>0.5</THRESHOLD>
        <FILE>lgd_relaybox_near.nt</FILE>
        <RELATION>lgdo:near</RELATION>
    </REVIEW>

    <EXECUTION>
        <REWRITER>default</REWRITER>
        <PLANNER>default</PLANNER>
        <ENGINE>default</ENGINE>
    </EXECUTION>

    <OUTPUT>TAB</OUTPUT>
</LIMES>
```

5.4. Running LIMES

Once the configuration file containing all the 10 elements detailed in the previous section is written, the last step consists in actually running the LIMES framework. Here, we detail how to run LIMES with an arbitrary configuration file (dubbed here as `config.xml`).

5.4.1. Running LIMES core

For running LIMES from command line, the following command needs to be executed:

```
java -jar LIMES.jar config.xml [OPTIONS...].
```

The following optional command line flags and options are available:

- `-f $format` sets the format of configuration file. Possible values for `$format` are `"XML"` (default) or `"RDF"`
- `-s` runs the LIMES server
- `-p $port` used to specify port of LIMES server; defaults to port 8080
- `-l $limit` limits the number of resources processed by LIMES server to `$limit`; defaults to -1 (no limit). **CAUTION:** Setting this option will compromise the correctness of LIMES and is only encouraged to reduce server load for demo purposes.
- `-h` prints out a help message
- `-o $file_path` sets the path of the logging file

In case your system runs out of memory, please use the `-Xmx` option (must appear before the -jar option) to allocate more memory to the Java Virtual Machine.

5.4.2. Running LIMES GUI

For running LIMES GUI, one should switch to the `limes-gui/` folder and run:

```
mvn jfx:jar -Dcheckstyle.skip=true -Dmaven.test.skip=true
```

The jar will be placed in `limes-gui/limes-gui/target/jfx/app/`

Note that, the `limes-gui/target/jfx/app/lib` folder needs to be in the same folder as the `.jar` for the `.jar` to work.

5.4.3. Running LIMES from Java

For running LIMES from java, please consult the developer manual at:

```
https://dice-group.github.io/LIMES/developer_manual
```

5.5. LIMES GUI

The main purpose of the LIMES GUI is to provide the users of LIMES with an easy interface for configuring LIMES, without the need to write the configuration file in the XML or RDF serialization. The LIMES GUI (see

Figure 19) consists of two main components the *menu bar* and the *tool box*. We will discuss each of the two components in the following subsections.

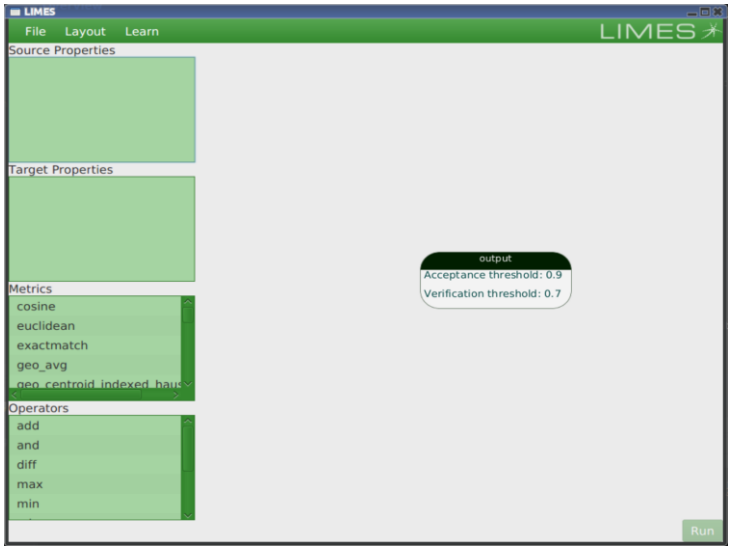


Figure 19: LIMES GUI

5.5.1. Menu Bar

The menu bar (as shown in top of Figure 19) contains three drop-down menus:

- **File:** The file drop-down menu gives the possibility to:
  - New: Create a new configuration
  - Load Config: Load a configuration file.
  - Save Config: Save a configuration to a file (only possible, after loading a configuration or creating a new configuration)
  - Exit
- **Layout:** Handles the layout of the current metric
  - Refresh Layout: Rearranges the nodes of the metric in a tree-like structure
  - Delete Graph: Delete the current metric leaving only an output node
- **Learn:** All the machine learning functionality of the GUI can be accessed through this drop-down menu (These features are only available when a configuration is loaded):
  - *Active Learning*
  - *Batch Learning*
  - *Unsupervised Learning*

5.5.2. Toolbox

On the panel of the GUI, one can find the toolbox (see Figure 20) containing all options of LIMES. The user needs to build his own metric after (s)he loaded/made a configuration, the components of the toolbox include:

- Source/Target Properties: The properties you want to link (if you did not load or create a configuration these are empty)
- Measures: All the measures you can use to link properties
- Operators: All the operators you can use to combine measures

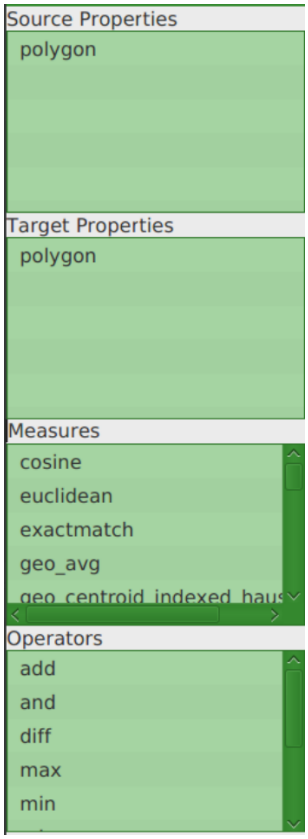


Figure 20: Toolbox

5.5.3. Metric Builder

The metric builder (see Figure 21) eases the process of complex link specification creation, especially for end users with limited programming experience. In particular, the user can visually link the various atomic link specification nodes to create the complex ones (s)he needs.

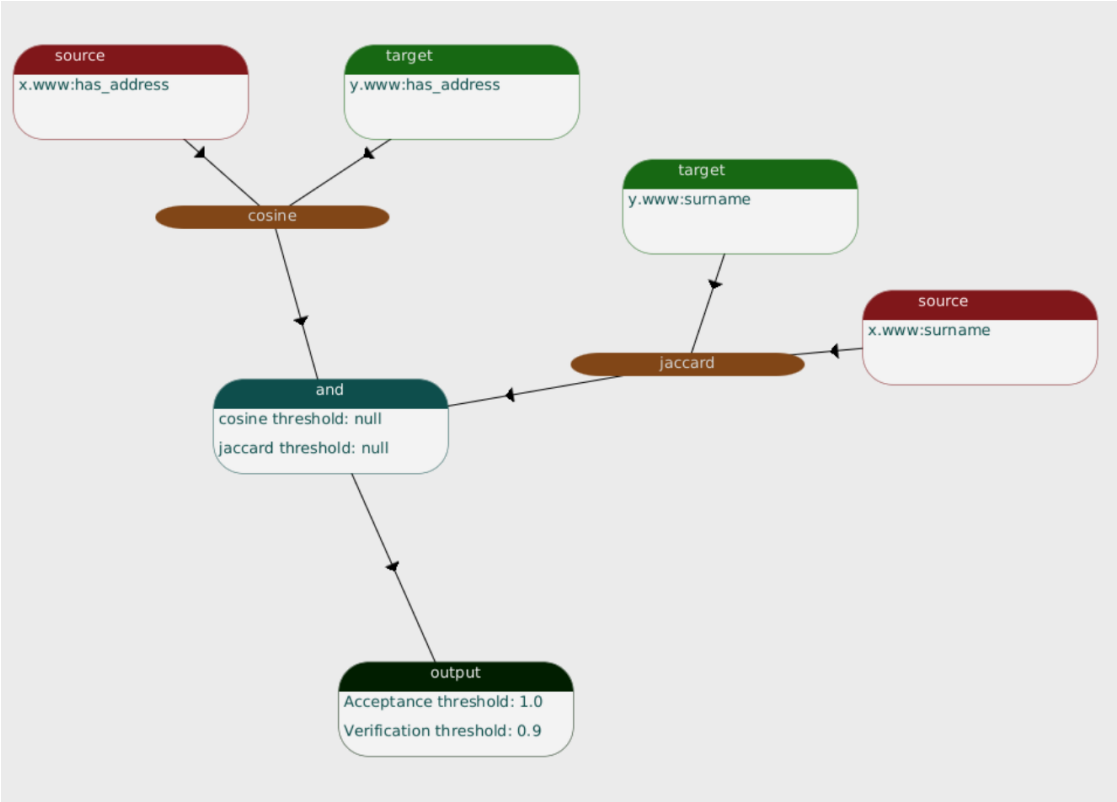


Figure 21: Metric builder

5.5.4. Creating New Configuration via GUI

The GUI user can configure LIMES to **read data from an external SPARQL endpoint** by clicking on File -> New. A window will pop up (as the one shown in Figure 22), in which the source and target endpoints of the new configuration can be configured.

Figure 22: SPARQL configuration window

The user can use the following fields for configuring his/her LIMES run:

- *EndpointURL:* Either a URL of a SPARQL Endpoint is entered here or the *filepath* to a local endpoint. Files can also be entered more easily by pressing the little green button with the file symbol which opens a file chooser dialog.
- *ID/Namespace*: Source/Target Endpoint can be given a name (optional).
- *Graph*: Specify the graph. If this is left empty, the default graph will be used.
- *Page size*: How many pages of the endpoint should be fetched? (-1 = all)

Let's use `http://dbpedia.org/sparql` as source endpoint and `http://linkedgeodata.org/sparql` as target endpoint URL. We enter `DBpedia` as *source ID* and `LGD` as target ID.

Pressing the *Next* button gets you to the next step of **class matching** presented in Figure 23. A source and target class must be selected by clicking on it to continue. Some classes have subclasses which can be accessed by clicking on the arrow besides them. We click on `HistoricPlace` from DBpedia and `HistoricThing` from *LGD*.

The Next step is **property matching** is shown in Figure 24. Clicking on the available properties moves them to the bottom container, where the already added properties can be seen. If you change your mind, clicking on added properties moves them back up. Alternatively, all available properties can be added with the button Add All. At least one source and one target property have to be added.

Let's take `rdfs:label` for both. Since the properties are alphabetically sorted you can find those towards the bottom of each list.

Press Finish and you are now ready to build a metric!

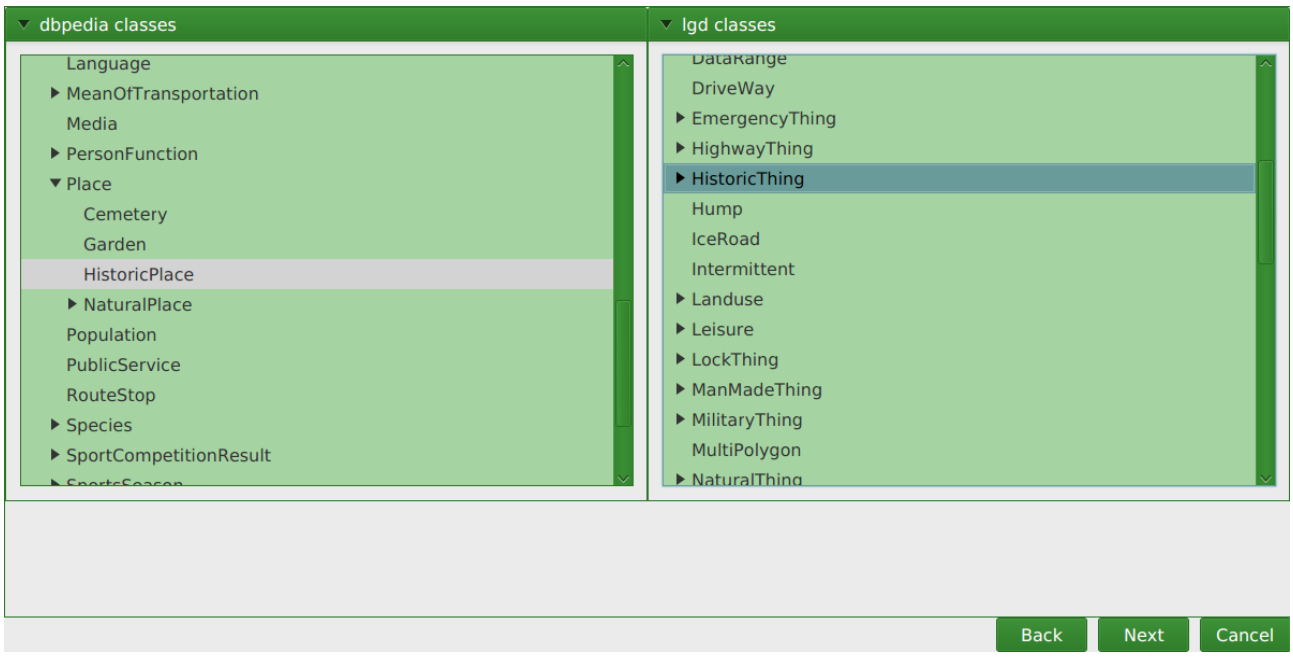


Figure 23: Class matcher

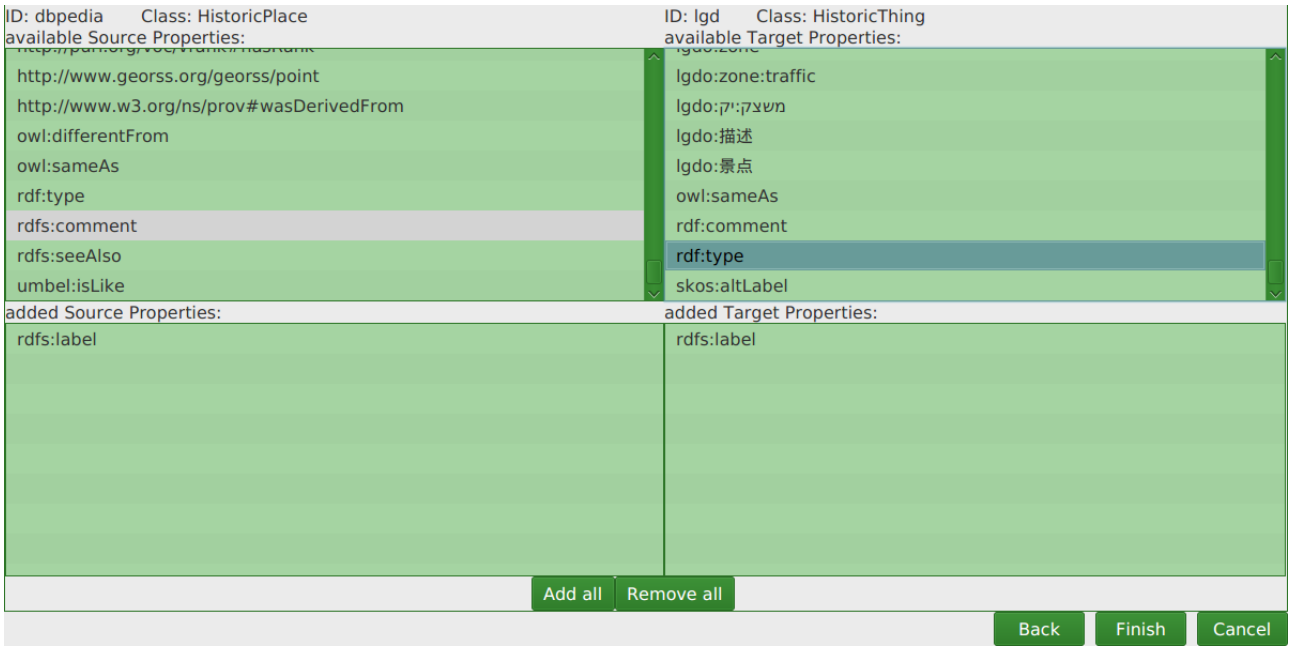


Figure 24: Property matcher

5.5.5. Creating a new Link Specification via GUI

By clicking on elements from the toolbox, the user can make the nodes he/she needs to appear in the *metric builder* (Like the ones in Figure 21). Let us assume that the user clicked on both `rdfs:label` properties he/she has. Now we need a measure to check the similarity between those properties. Let's choose `cosine` for example.

Right clicking the nodes will create a small context menu for the current node. If the user clicks *Link To*, the user can link the node with an appropriate other node. The following links are permitted:

- property -> measure
- measure -> output
- measure -> operator
- operator -> output

Also, operator and measure need two nodes that link to them. The context menu also gives the user the possibility to *Close* it or *Delete* the node. If the user wants to delete a link, he/she just needs to right-click the arrow. Let's link our properties with `cosine` and the measure with `output`. If the user wants, he/she can define a Acceptance Threshold and Verification Threshold.

5.5.6. Link Specification Running

If the user followed the steps, his/her *link specification* should look something like the one presented in Figure 25. For running the current *link specification*, the user just click on the button in the bottom right corner.

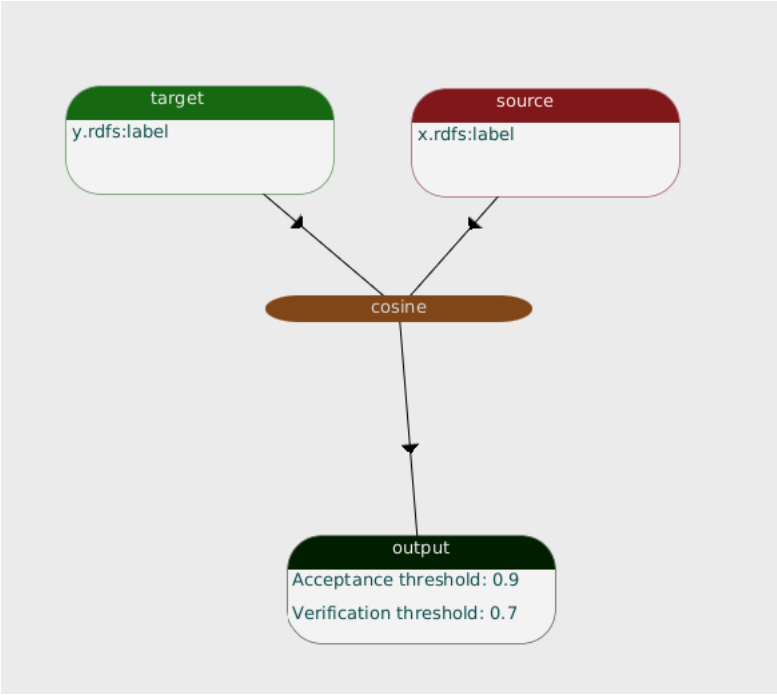
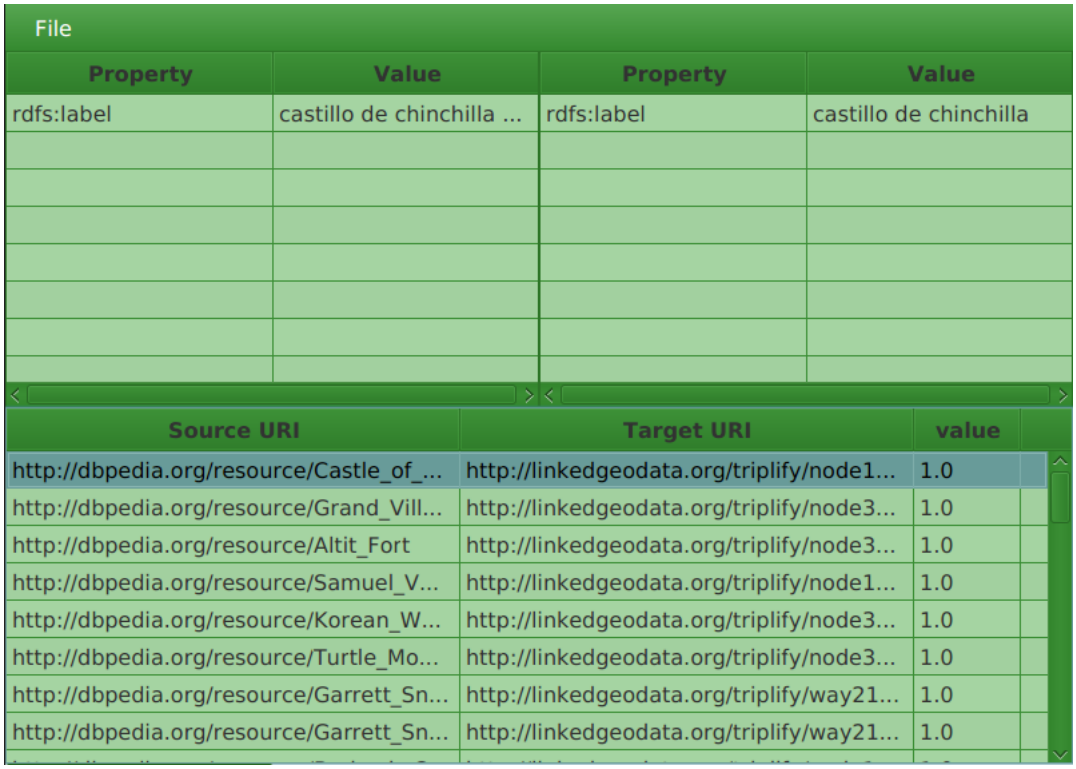


Figure 25: LS example in the metric builder

After the progress popup vanished the user should see his/her results in a new window such as the one presented in Figure 26. In the top left the user has the possibility to save the resulted set of links into a file.



File

Property	Value	Property	Value
rdfs:label	castillo de chinchilla ...	rdfs:label	castillo de chinchilla

Source URI	Target URI	value
http://dbpedia.org/resource/Castle_of_...	http://linkedgeodata.org/triplify/node1...	1.0
http://dbpedia.org/resource/Grand_Vill...	http://linkedgeodata.org/triplify/node3...	1.0
http://dbpedia.org/resource/Altit_Fort	http://linkedgeodata.org/triplify/node3...	1.0
http://dbpedia.org/resource/Samuel_V...	http://linkedgeodata.org/triplify/node1...	1.0
http://dbpedia.org/resource/Korean_W...	http://linkedgeodata.org/triplify/node3...	1.0
http://dbpedia.org/resource/Turtle_Mo...	http://linkedgeodata.org/triplify/node3...	1.0
http://dbpedia.org/resource/Garrett_Sn...	http://linkedgeodata.org/triplify/way21...	1.0
http://dbpedia.org/resource/Garrett_Sn...	http://linkedgeodata.org/triplify/way21...	1.0

Figure 26: Result window

5.5.7. Machine Learning via GUI

Since finding a good metric expression can be hard, we also have implemented machine learning algorithms in the GUI. There are three different types of algorithms you can use:

- Active Learning
- Batch Learning
- Unsupervised Learning

To use any of these algorithms, the user has to either create a new LIMES configuration or load one from file. In the menu bar, the user should click on *Learn* and choose the type he/she wants to use. A new window will pop up. In the top left corner. The user will find a drop-down menu, showing him/her which algorithms implement the chosen learning type. After he/she clicks on his/her desired algorithm, the window will be filled with the elements he/she can use to set the configuration parameters. Figure 27 shows an example of setting the parameters of the WOMBAT simple ML algorithm from the GUI.

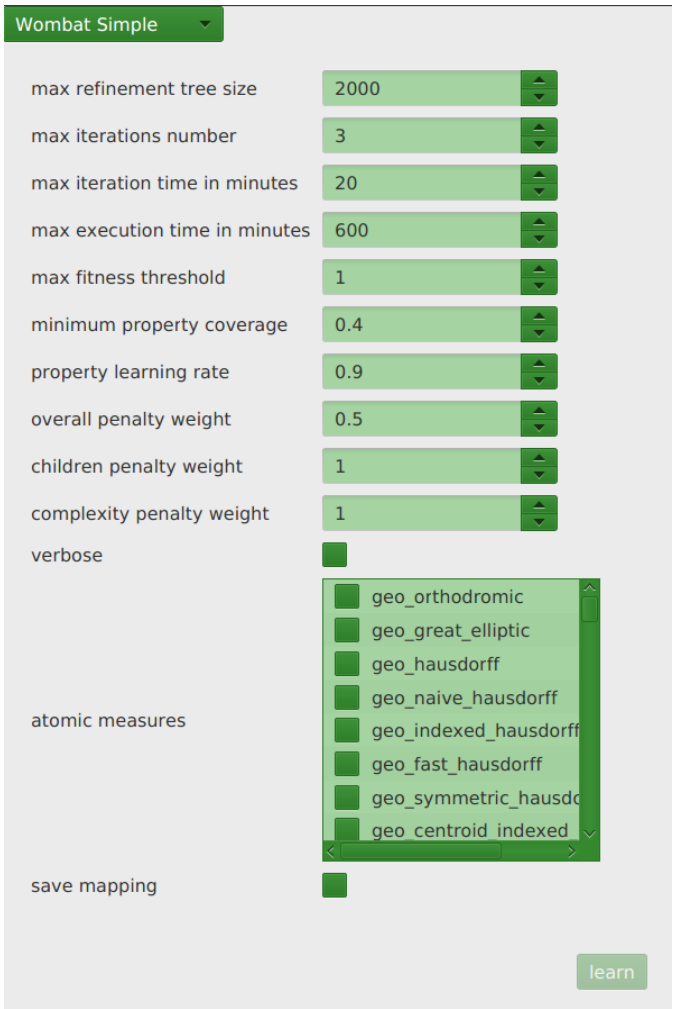


Figure 27: WOMBAT simple parameters in the GUI

5.5.7.1. Active Learning via GUI

In case the user is happy with the parameters he/she set, the user must click on *Learn* in the bottom right corner. After the progress popup vanishes, he/she will see a new window, where the algorithm wants you to label link candidates as matches or non-matches as shown in Figure 28.

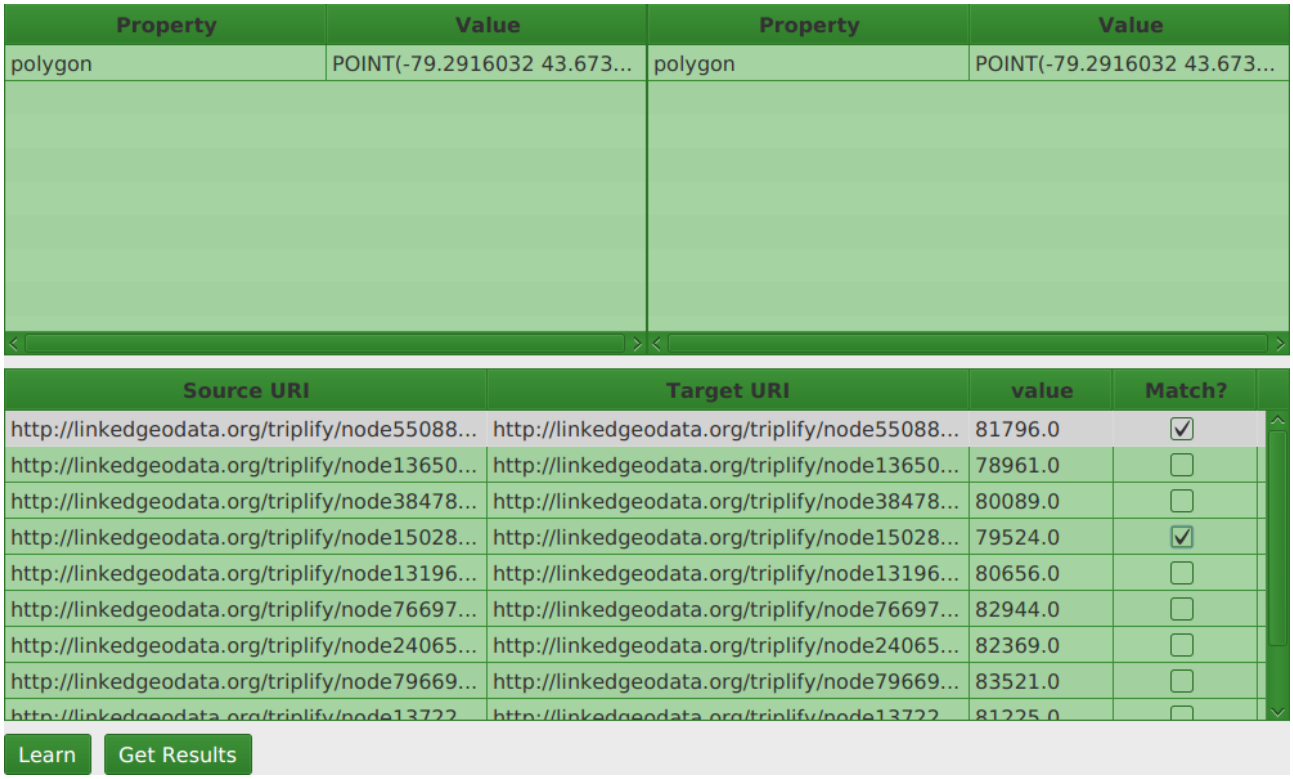


Figure 28: active learning via GUI

The user can click on *Learn* again and another iteration starts. If she doesn't want another iteration, the user can click on *Get Results* and a new view with results will pop up. This time he/she also has the possibility to *Save the resulted link specification* in the bottom left corner.

5.5.8. Batch Learning via GUI

This learning type only takes one iteration and the user has to provide a file containing the training mapping. The file can be either CSV or RDF in any serialization. For CSV, the first line contains the properties on which the user wants to match, and the following lines the respective properties' values of the instance. For example:

```
id1,id2
```

```
http://www.okkam.org/oaie/person1-Person2190,http://www.okkam.org/oaie/person2-Person2191
```

For example, if the user uses an RDF file in n-triple serialization, a mapping for `owl:sameAs` predicate will look like:

```
<http://linkedgeodata.org/triplify/node3635863841>
<http://www.w3.org/2002/07/owl#sameAs>
<http://linkedgeodata.org/triplify/node3635863841>
```

Of course, the more training data the user provides the better the algorithm can learn. After the user click on *Save* the learning will start.

5.5.9. Unsupervised Learning via GUI

This is the type of machine learning algorithms that needs the least effort from the user. The user has just to click on *Learn* and after the algorithm is finished, the results will be presented. Note that, LIMES only support the unsupervised machine learning for `owl:sameAs` relations.

6. DEER Usage Manual

In this Section, we provide the usage manual for DEER v2.2. First, we give details on building the application from the Java source code. Next, we provide instructions for the configuration of DEER. Finally, we present a short demonstration example on configuring and running DEER.

6.1. Building Installation

DEER v2.2 is publicly available (see https://github.com/SLIPO-EU/deer), offering the entire Java source code as well as indicative configurations. Java SDK 1.8[63] (or later) as well as Maven 3.3.3[64] (or later) must be installed and properly configured in order to compile and execute DEER. The `pom.xml` file contains the project's configuration in Maven and has been successfully tested in Mac OS, Microsoft Windows and Linux environments. The following building instructions assume that Git is also installed.

DEER is split into two maven submodules: **deer-core** and **deer-cli**. While deer-core is intended to be used programmatically from other Java applications, deer-cli provides a CLI to either run a single configuration or start the DEER server.

6.1.1. Generating Jar File (Headless)

```
git clone -b master --single-branch https://github.com/SLIPO-EU/DEER.git DEER
```

It is recommended to use the `--single-branch` parameter to save some time and avoid pulling the whole history of the project.

Then, from the root directory of the project (DEER) the following command needs to be executed:

```
mvn clean install
```

For creating the runnable jar file including the dependencies use:

```
mvn clean package shade:shade -Dmaven.test.skip=true
```

The runnable jar file will be generated into `deer-cli/target/deer-cli-${version}.jar`.

6.1.2. Import DEER using Maven

Using Maven, DEER can be imported to another project using:

```
<dependencies>
  <dependency>
    <groupId>org.aksw.deer</groupId>
    <artifactId>deer-core</artifactId>
    <version>{insert version here}</version>
  </dependency>
</dependencies>

<repositories>
```

[63] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

[64] https://maven.apache.org/docs/3.3.3/release-notes.html

```
 <repository>
      <id>maven.aksw.internal</id>
      <name>University Leipzig, AKSW Maven2 Internal Repository</name>
      <url>http://maven.aksw.org/repository/internal/</url>
    </repository>

    <repository>
      <id>maven.aksw.snapshots</id>
      <name>University Leipzig, AKSW Maven2 Snapshot Repository</name>
      <url>http://maven.aksw.org/repository/snapshots/</url>
    </repository>
</repositories>
```

6.2. Configuring DEER

6.2.1. FARADAY-CAGE Core Vocabulary

The RDF graph consists of execution nodes which can optionally be plugins and/or parametrized. An execution node can have multiple incoming edges (each carrying different data) and produce multiple outgoing edges (each carrying different data). As a result, it is important to assign indices to edges for each execution node. We call these indices *ports*.

FARADAY-CAGE specifies the following vocabulary for any given execution node (here labelled `:e1`):

- `:hasInput` (required) declares the incoming edges to this execution node. Allowed values are:
  - another execution `node :e2` (if `:e2`s has an edge from its first port to `:e1`s first port)
  - a list of other execution nodes (if :e1 has incoming edges from each execution nodes first port)
  - a list of blank nodes (or resources) representing outgoing edges with the following properties:
    - `:fromNode` the execution node this edge points to
    - `:fromPort` the port of the execution node this edge points to
- `rdf:type (a)` (for plugin nodes) declares the resource identifying the implementation of the plugin.

6.2.1.1. Example

The following abstract configuration graph serves as an example for how the syntax elements we have just introduced can be combined in order to assemble a configuration for FARADAY-CAGE based projects. It is serialized in the Turtle format for brevity and ease of reading.

```
:e1 a :somePluginClassIdentifier .

:e2 a :somePluginClassIdentifier ;
    :hasInput :e1 .

:e3 a :somePluginClassIdentifier ;
    :hasInput :e2 .
```

```
:e4 a :somePluginClassIdentifier ;
    :hasInput ( [ :fromNode :e2 ; :fromPort 0 ]
                [ :fromNode :e3 ; :fromPort 0 ] ) ;
:e5 a :somePluginClassIdentifier ;
    :hasInput ( [ :fromNode :e3 ; :fromPort 2 ]
                [ :fromNode :e3 ; :fromPort 1 ]
                [ :fromNode :e4 ; :fromPort 0 ] ) ;
```

The above example generates the following graph:

```
e1 ----> e2 ----> e3 ----------------> e5
            \           \             /
             \-----------> e4 ----/
```

Plugins define their own configuration vocabulary. In the next two sections, we provide an accurate description of the available parameters for each plugin in the current DEER release.

6.2.2. DEER IO Plugins

6.2.2.1. DefaultModelReader

The *DefaultModelReader* can read in RDF from any file format supported by Apache Jena as well as from querying SPARQL endpoints. Its supported configuration parameters are:

- `:useEndpoint` if this is given, *DefaultModelReader* will operate in endpoint mode and the endpoint will be set to the value of this parameter.
- `:fromUri` in endpoint mode, issue a *DESCRIBE* query for the given URI; in normal mode read in RDF directly from the given uri (can also be a file path for local files).
- `:useSparqlConstruct` issue the *CONSTRUCT* query given as value of this parameter against either the model obtained by reading in the RDF in normal mode or the SPARQL endpoint in endpoint mode. In endpoint mode, this has precedence over `:fromUri`, i.e., if both are specified, only the *CONSTRUCT* query is issued.

6.2.2.2. DefaultModelWriter

The DefaultModelWriter can write RDF to any file format supported by Apache Jena. Its supported configuration parameters are:

- `:outputFile` (required) specifies the name of the emitted file
- `:outputFormat` specifies the file format to be used, defaults to Turtle. Supports all formats of Apache Jena.

6.2.3. DEER Enrichment Operator Plugins

In the following, we provide a comprehensive list of the enrichment operators included in DEER, their degree bounds, parameters and modes of operation. If not specified otherwise, the degree bounds on enrichment operators are `(1,1,1,1)`.

6.2.3.1. Filter Enrichment Operator

The idea of the filter enrichment operator is to select a specific set of the input dataset triples. Then, the selected set of triples are generated by the filter operator as its output. The filter enrichment operator defines the set of triples to be selected using the single parameter `:selector`. The `:selector` parameter accepts a selector. By selector we mean the triple patterns to be applied to the input dataset. The filter enrichment operator has three basic selector types, i.e. the `:subject`, `:predicate` and `:object` selectors.

The following example configuration demonstrates the configuration of the filter enrichment operator for filtering only the predicates `geo:lat`, `geo:long`, `rdfs:label` and `owl:sameAs`.

```
@prefix : <http://w3id.org/deer/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

:node_filter
              a :FilterEnrichmentOperator ;
              :hasInput          ( :node_in ) ;
              :selector
                    [ :predicate geo:lat ] ,
                    [ :predicate geo:long ] ,
                    [ :predicate rdfs:label ] ,
                    [ :predicate owl:sameAs ] ,
              .
```

6.2.3.2. Linking Enrichment Operator

The idea of the linking enrichment operator is to enrich models with links discovered using LIMES. It can be used in two modes:

1. enriching a dataset with links from an external linking process (1 input, 1-2 outputs)
    - This mode will be selected if the linking enrichment operator node has exactly one input dataset.
    - If it has exactly one output dataset, it will be the original input dataset enriched with the links.
    - If it has exactly two output datasets, the first will be just the original input dataset and the second will be a dataset of just the generated links.
    - In this mode, the following parameters are accepted:
        - `:specFile` which is a path for LIMES specification file
        - `:linksPart` one of `"source"`, `"target"`. If `"source"` is selected (default), nothing happens. If `"target"` is selected, we swap subjects and objects in the generated links.
        - `:selectMode` determines the strategy for selecting which links to keep from the original full limes mapping
            - `"all"` (default) all links are kept

- `"best1toN"` enforces 1-to-N mapping. For each target resource, only keep the best link to the source.
- `"best1to1"` enforces 1-to-1 mapping. No resource in either target or source dataset will appear in more than one link.
- `"best"` keep just the best link. If there are ties, it is unspecified which will be selected.

2. enriching a dataset with links from an internal linking process (2 inputs, 1-3 outputs)
    - This mode will be selected if the linking enrichment operator node has exactly two input datasets.
    - If it has exactly one output dataset, it will be the result of merging the two input datasets and the generated links
    - If it has exactly two output datasets, they will correspond to the two input datasets in order and depending on the parameter `:linksPart`, either the first or the second will be enriched with the generated links.
    - If it has exactly three output datasets, they will correspond to the two input datasets in order and the third one will be a dataset of just the generated links.
    - In this mode, the following parameters are accepted:
        - `:linksPart` one of `"source"`, `"target"`. If `"source"` is selected (default), nothing happens. If `"target"` is selected, we swap subjects and objects in the generated links.

- Moreover, if the number of output datasets is exactly two, this parameter will decide which of the two datasets will be enriched with the generated links. If it is "source" it will be the first, if it is "target" it will be the second.
    - `:selectMode` determines the strategy for selecting which links to keep from the original full limes mapping
        - `"all"` (default) all links are kept
        - `"best1toN"` enforces 1-to-N mapping. For each target resource, only keep the best link to the source.
        - `"best1to1"` enforces 1-to-1 mapping. No resource in either target or source dataset will appear in more than one link.
        - `"best"` keep just the best link. If there are ties, it is unspecified which will be selected.
        - `:linkSpecification` the link specification to execute (as string literal)
    - `:linkingPredicate` the predicate with which links will be built
    - `:threshold` the similarity threshold. All links in the resulted mapping will have a similarity value greater than or equal to the threshold.

In the following example, the linking enrichment operator is used based on the LIMES configuration file "limes_specs.xml" and the source dataset is the one to be enriched.

```
@prefix : <http://deer.aksw.org/vocabulary/#> .

:node_linking
              a :LinkingEnrichmentOperator ;
              :hasInput          ( :node_in ) ;
              :specFile          "limes_specs.xml" ;
              :linksPart         "source" .
```

6.2.3.3. Dereferencing Enrichment Operator

For datasets which contain links to similar resources (e.g., `owl:sameAs`), the goal of the dereferencing enrichment operator is to deference all links from an externally linked dataset to our input dataset by using a content negotiation on HTTP. This process returns a set of triples that need to be filtered for relevant information. In addition to the common parameters, the dereferencing operator uses the parameter `:operation`, which takes a node with the following configurations:

- `:lookUpPrefix` interesting resources from the external dataset
- `:dereferencingProperty` interesting property to extract from the external dataset
- `:importProperty` for renaming of the `:dereferencingProperty` in the output dataset

In the following example, the dereferencing enrichment operator is used to find `dbo:abstract` from the external dataset of DBpedia and export them using the `dcterms:description`.

```
 @prefix : <http://w3id.org/deer/> .
 @prefix dbo: <http://dbpedia.org/property/> .
 @prefix dcterms: <http://purl.org/dc/terms/> .

:dereferencing_dbp
              a    :DereferencingEnrichmentOperator ;
              :hasInput         ( :node_in ) ;
              :operation
                    [ :lookUpPrefix "http://de.dbpedia.org/resource" ;
                      :dereferencingProperty dbo:abstract ;
                      :importProperty dcterms:description ]
                .
```

6.2.3.4. NER Enrichment Operator

The enrichment information hidden in datatype properties is retrieved by using Named Entity Recognition (NER) enrichment operator. In the current version of DEER, we rely on the FOX framework. In the following, we provide details about the NER operator parameters:

- `:literalProperty` Literal property used by FOX for NER. If not set, the top ranked literal property will be pecked automatically by DEER, which ranks the lateral properties of a model according to the average size of each literal property divided by the number of instances of such property.
- `:importProperty` Property of interest from the dereferenced resources to be added into the input dataset.

- `:neType` Force FOX to look for a specific NE's types only. Available types are: `:location` (default value), `:person`, `:organization`, and `:all` to retrieve all the previous three types.
- `:askEndpoint` Ask the *DBpedia* endpoint for each location returned by FOX (setting it generates slower execution time but more accurate results). By default, this parameter is set to false.

6.2.3.5. Merge Enrichment Operator

The idea behind the merge operator is to enable combining datasets. The merge operator takes a set of n ≥ 2 input datasets and merges them into one output dataset containing all the triples from all the input datasets.

In the following example, the merge operator is used to combine the 2 input datasets of `:node_1` and `:node_2` into `:node_3`

```
@prefix : <http://w3id.org/deer/> .

:node_merge
              a     :MergeEnrichmentOperator ;
              :hasInput          ( :node_in_1 :node_in_2).
```

6.2.3.6. Geo-Fusion Enrichment Operator

The idea of the geo-fusion enrichment operator is to merge two or more input datasets into one fused output dataset. In addition to the common parameters, the geo-fusion operator has the following additional parameter:

The `:fusionAction` is used to specify the how to fuse geo-spatial properties (by default the `geo:lat` and `geo:long`), the available fusion actions are:

- `"takeA"` always use geometry from first dataset
- `"takeB"` always use geometry from second dataset
- `"takeAll"` merge all geometries
- `"takeMostDetailed"` use most detailed geometry from any model, e.g., in terms of lexical length of latitude and longitude values
- The `:mergeOtherStatements` parameter is used to enable the merge of all other non geo-spacial properties from all input dataset to the output dataset

```
@prefix : <http://w3id.org/deer/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

:node_geofusion
              a     :GeoFusionEnrichmentOperator ;
              :hasInput          ( :node_in_1 :node_in_2 ) ;
              :fusionAction      "takeAll" ;
              :mergeOtherStatements
                                 "true".
```

6.2.3.7. Authority Conformation Enrichment Operator

The idea of the authority conformation operator is to change a specified source URI authority to a specified target URI authority. In addition to the common parameters, the authority conformation enrichment operator accepts the `:operation` parameter, which in turn expects its objects to have the two basic parameters of `:sourceAuthority` and `:targetAuthority` for specifying the source and target authorities respectively.

In the following example, we use the `:sourceSubjectAuthority` of `http://dbpedia.org` and the `:targetSubjectAuthority` of `http://deer.org`. Such configuration will change a resource like `http://dbpedia.org/Berlin` to `http://slipo.eu/Berlin`.

```
@prefix : <http://w3id.org/deer/> .

:node_a_conf
              a     :AuthorityConformationEnrichmentOperator ;
              :hasInput          ( :node_in ) ;
              :operation [
                  :sourceSubjectAuthority
                                 "http://dbpedia.org" ;
                  :targetSubjectAuthority
                                 "http://slipo.eu"
              ].
```

6.2.3.8. Predicate Conformation Enrichment Operator

The idea of the predicate conformation operator is to replace all instances of specified source property to a specified target predicated with the same object and subject values. In addition to the common parameters, the predicate conformation enrichment operator accepts the basic parameters of `:operation` for specifying the `:sourcePredicate` and `:targetPredicate` predicates.

In the following example, we use the predicate conformation enrichment operator to change all instances of `rdf:label` to `SKOS:prefLabel`. For example, a triple as `dbp:Berlin rdf:label "Berlin"` to `dbp:Berlin skos:prefLabel "Berlin"`.

```
@prefix : <http://w3id.org/deer/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

:node_p_conf
              a     :PredicateConformationEnrichmentOperator ;
              :hasInput          ( :node_in ) ;
              :operation
                    [ :sourcePredicate rdf:label ;
                      :targetPredicate skos:prefLabel ]
              .
```

6.2.3.9. GeoDistance Enrichment Operator

The idea of the geo-distance operator is to enrich resources that are connected via some predicate (configured with parameter `:selectPredicate`) with the distance between their respective geo-coordinates. The distance in kilometres will be added as a string literal to the subject via the predicate specified in parameter `:distancePredicate`.

In the following example, we use the geo-distance enrichment operator to enrich cities from DBPedia with the distance to their western adjacent cities.

```
@prefix : <http://deer.aksw.org/vocabulary/#> .
@prefix dbp : <http://dbpedia.org/property/> .

:node_geodistance
              a     :GeoDistanceEnrichmentOperator ;
              :hasInput          ( :node_in ) ;
              :selectPredicate   dbp:west ;
              :distancePredicate :distanceToNextCityInTheWest.
```

6.3. Running DEER

Once the configuration file containing all the configuration elements detailed in the previous section is written, the last step consists of actually running the DEER framework. For running DEER from command line, simply run:

```
java -jar DEER.jar config.ttl
```

Note that we presented the DEER configuration in *RDF Turtle* format because of its simplicity and compactness. In general, *DEER accepts its configuration file in any RDF serialization* including but not limited to: N3, N-Triples, Turtle, JSON-LD and RDF-XML.

In case your system runs out of memory, please use the `-Xmx` option (must appear before the -jar option) to allocate more memory to the Java Virtual Machine.

6.4. Extending DEER

DEER is built on top of FARADAY-CAGE, which uses the plugin system PF4J. This is helpful for developers willing to extend DEER with their custom enrichment operators, as it does require to fork and edit the DEER source directly. Instead, a developer may just follow a few guidelines and implement an easy interface to plug her new enrichment operator into DEER.

We define three base interfaces:

```
1. org.aksw.deer.DeerPlugin
2. org.aksw.deer.ParametrizedDeerPlugin
3. org.aksw.deer.learning.SelfConfigurator
```

and four abstract classes:

```
1. org.aksw.deer.enrichments.AbstractEnrichmentOperator
2. org.aksw.deer.enrichments.AbstractParametrizedEnrichmentOperator
3. org.aksw.deer.io.AbstractModelReader
4. org.aksw.deer.io.AbstractModelWriter
```

One should always extend the abstract classes and never need to implement the interfaces directly, but they could come in handy when trying to obtain instances programmatically using `org.aksw.faraday-cage.PluginFactory`. For more information on how to extend these classes, please read the Javadoc.

An example plugin using a parameter-less enrichment operator can be found in

```
https://github.com/dice-group/deer/tree/master/examples/simple-plugin-example/
```

7. FAGI Usage Manual

In this section, we provide the usage manual for FAGI v3.0. First, we give details on building the application from the Java source code, along with information on its dependencies. Next, we provide instructions on constructing link validation and fusion action rule specifications, and for configuring the execution of FAGI. Finally, we present a short example demonstrating the configuration and execution of FAGI.

7.1. Building Installation

FAGI v3.0 is available as an open source software, with its source code as well as indicative configurations available for download at [FAGI][FAGI-web]. Java SDK 1.8 (or later) as well as Maven 3.3.3 (or later) [MAVEN] must be installed and properly configured in order to compile and execute FAGI. The `pom.xml` file contains the project's configuration in Maven and has been successfully tested in both Microsoft Windows and Linux environments. The following building instructions assume that Git is also installed.

7.1.1. Command line version

In order to build the command line version from source, first the master branch of FAGI must be cloned to a preferred location by running:

```
git clone -b master --single-branch https://github.com/SLIPO-EU/FAGI.git fagi
```

It is recommended to use the `--single-branch` parameter to save some time and avoid pulling the whole history of the project.

Then, from the root directory of the project (fagi) the following command needs to be executed:

```
mvn clean install
```

After a successful installation, a target directory should have been created containing the fagi-2.0-SNAPSHOT.jar (version depending on POM configuration).

In order to deploy FAGI, the following command needs to be executed under the target directory:

```
java -jar fagi-1.2.3-SNAPSHOT.jar -spec /path/to/conf.xml
```

where “conf.xml” is the FAGI v2.0 configuration file (described in Section 7.2.1).

7.1.2. Web interface version

Optionally, FAGI provides a web interface which helps the user build complicated rule specifications without having to manually define rules in XML files. The web interface is a Spring Boot application which makes it very easy to install and use. The web interface installation assumes that the command line version is already installed and uses the command-line version as a library.

Similarly, the FAGI-web repository needs to be cloned to a preferred location on your system by running:

```
git clone https://github.com/SLIPO-EU/fagi-web.git fagi-web
```

In order to build the project, the following command needs to be executed from the root directory of the project (fagi-web):

```
mvn clean install
```

After a successful installation, the project can be easily started by running the following command in the same directory:

```
mvn spring-boot:run
```

Upon the application has been initialized and started, the web interface is accessible through a browser at by visiting

```
http://localhost:8080
```

7.2. Configuration Settings

FAGI requires two XML files as configuration input. These files contain some basic input/output information, as well as the fusion modes and the rule specification. In this section, we describe the structure and syntax of these files and we provide some basic examples.

7.2.1. Configuration specification

The first of these files is provided through the command line with the "-co" parameter followed by the path of the configuration file. The configuration specification follows an XML syntax and holds general configuration for the fusion process, which is filled with text values between an opening and a closing tag. A template of this file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<specification>
        <inputFormat></inputFormat>
        <outputFormat></outputFormat>
        <locale></locale>
        <similarity></similarity>
        <verbose></verbose>
        <rules></rules>
        <left>
                <id></id>
                <endpoint></endpoint>
                <file></file>
                <categories></categories>
        </left>
        <right>
                <id></id>
                <endpoint></endpoint>
                <file></file>
                <categories></categories>
        </right>
        <links>
                <id>links</id>
                <endpoint></endpoint>
                <linksFormat></linksFormat>
                <file></file>
        </links>
        <target>
```

```
                <id>fused</id>
                <mode></mode>
                <outputDir></outputDir>
                <fused></fused>
                <remaining></remaining>
                <ambiguous></ambiguous>
                <statistics></statistics>
                <fusionLog></fusionLog>
        </target>
</specification>
```

The *inputFormat* refers to the RDF format of the input dataset and the *outputFormat* holds the value of the desired output format. The valid RDF formats are:

- N-Triples (NT)
- Turtle (TTL)
- RDF/XML (RDF)
- RDF/XML (OWL)
- JSON-LD (JSONLD)
- RDF/JSON (RJ)
- TriG (TRIG)
- N-Quads (NQ)
- TriX (TRIX)

In order to fill the *inputFormat* and *outputFormat*, the values in the corresponding parentheses are used within the file.

The *locale* is an optional parameter, in case a dataset contains entities from regions with different locales, but it is strongly recommended to choose one when applicable, because it is used on several steps of the normalization process. The available locales are:

- EN
- EN-GB
- EN-US
- DE
- DE-DE
- DE-AT
- EL

The *similarity* parameter is used as a part of the customized similarity function we implement within FAGI. The available values (case-insensitive) are the following (default is *JaroWinkler*):

- levenshtein
- 2Gram

- longestcommonsubsequence
- jaro
- jarowinkler

The *rules* field expects the absolute path of the rule specification XML file, which is discussed in Section 7.2.2.

The *left*, *right*, *links* and *target* parameters refer to the source and target datasets. Each of these XML tags contain additional tags that describe each of the datasets. Specifically:

- *id*. An ID to identify the dataset.
- *file*. The filepath of the dataset.
- *endpoint* (optional parameter). Instead of using files, add a SPARQL endpoint and leave the file tag empty.
- *categories* (optional parameter). It is used to extract statistics about the categories of the entities. It requires the filepath of a file in N-Triples format that contains the categorization.

Concerning the links, there are two supported formats. N-triples like `<poiA> <owl:sameAs> <poiB>` format and CSV like `poiA poiB score` (space separated and [0-1] value for the score). In the case of a CSV file, the user can define three different modes. The first takes into account all the provided links and executes the fusion process accordingly, the second keeps unique links with the highest confidence score, and the third takes into account POI-ensembles by handling cases that a POI from one dataset is linked with multiple POIs from the other (the rules applied in this case are different and described at the rule specification below). The values are the following:

- NT
- CSV
- CSV-unique-links
- CSV-ensembles

The *mode* parameter is used in order to specify the dataset output mode. The supported parameter values are the following:

- *aa_mode*: Only linked triples are handled – include base dataset. Fused triples along with unlinked triples only from dataset A are written in the output, fused dataset.
- *bb_mode*: Only linked triples are handled – include base dataset. Fused triples along with unlinked triples only from dataset B are written in the output, fused dataset.
- *ab_mode*: All triples are handled: Fused triples along with unlinked triples from both datasets are written in the output, fused dataset, using dataset A as base dataset.
- *ba_mode*: All triples are handled: Fused triples along with unlinked triples from both datasets are written in the output, fused dataset, using dataset B as base dataset.

- *a_mode*: All triples are handled. Fused triples along with unlinked triples only from dataset A are written in the output, fused dataset. Triples corresponding to fused POIs are removed from dataset B, which only maintains the remaining, unlinked triples.
- *b_mode*: All triples are handled. Fused triples along with unlinked triples only from dataset B are written in the output, fused dataset. Triples corresponding to fused POIs are removed from dataset A, which only maintains the remaining, unlinked triples.
- *l_mode*: Only linked triples are handled: Only fused triples are written in the output, fused dataset.

The *outputDir* parameter holds the directory path under which all produced files will be written. The produced files are described below:

- *fused* (optional parameter). Specifies the output filepath of the fused dataset (based on fusion mode). If no value is specified, the default name will be "fused.nt" under the output directory defined above.
- *remaining* (optional parameter). Specifies the output filepath of the non-fused dataset, containing the remaining, unlinked triples (based on fusion mode). If no value is specified, the default name will be "remaining.nt" under the output directory defined above.
- *ambiguous* (optional parameter). Specifies the output filepath of the dataset containing ambiguous linked entities. If no value is specified, the default name will be "ambiguous.nt" under the output directory defined above.
- *statistics* (optional parameter). Specifies the path of the statistics file. By default, a file with name "statistics.txt" will be written under the output directory defined above.
- *fusionLog.* Specifies the path of the fusion log file. By default a file with name "fusionLog.txt" will be written under the output directory defined above.

FAGI supports the prediction of validation and fusion actions with the use of ML models. These models are defined in the `ML` group tag.

- name: the path of the ML-model for name resources.
- address: the path of the ML-model for address resources.
- website: the path of the ML-model for website resources.
- phone: the path of the ML-model for phone number resources.
- email: the path of the ML-model for e-mail resources.
- validation: the path of the ML-model for link validation.

ML-predicted actions on a property cannot be used if the corresponding ML model is not defined in the above tags.

7.2.2. Rule Specification

The second configuration input for FAGI is the rule specification file. This file holds all the rules that the validation and fusion processes are to follow and requires that certain text values are filled between an

opening and a closing tag, similarly to the configuration specification file. The file starts with the root element <*rules*> and each fusion rule on a property is set using a <*rule*> element as a child of the root tag. Each <*rule*> element consists of the following main childs: <propertyA>, <propertyB>, <externalProperty>, <actionRuleSet>, <defaultAction>.

- <propertyA> and <propertyB> define the two matching RDF properties that the rule will apply the fusion action upon.
- <externalProperty> is used to define properties that are utilized inside condition functions. The fusion action does not affect the value of this property. The external property requires an id attribute as a parameter in the XML and the id must start with the letter *a* or *b* that refers to the corresponding value (from the corresponding input POI dataset) and followed by an incrementing integer for each different property used in the same rule.
- <actionRuleSet> element: This element consists of one or more <actionRule> child elements. Each is a pair of a condition expression and a fusion action, namely (`condition expression, fusion action`), denotes by tags <condition> and <action> respectively. If the condition expression evaluates to `True`, then the respective fusion action of the specific action rule will be applied and all the next action rules will be ignored.
- <defaultAction> is the default fusion action to apply if no condition from the is met.

The condition expression (<condition>) along with the fusion action (<action>) are the most essential part of the configuration of the fusion process. In order to construct a condition expression, we assemble a group of logical operations that contain condition functions to apply on the RDF properties defined above. We can define a logical operation by using the <expression> tag as a child of a condition. Then, inside the expression we can put together a combination of <and>, <or> and <not> operations. As operands we can use <function> elements containing a condition function or a nested expression containing more logical operations. The depth of the nested expressions supported currently is 2 levels.

Apart from fusion rules which are defined using the <rule> tag, similar functionality is supported for the definition of validation rules using the <validationRule> tag. With a validation rule we can accept/reject and/or mark a link as ambiguous in the model. The validation rules follow the exact same logic described above with the only difference being that the fusion actions are replaced with the validation actions. Examples of rule configurations are provided in Sections 4.1.1.3, 4.1.2.3 and 7.3.

7.3. Demonstration

Next, we demonstrate the usage of FAGI v3.0 through an exemplary execution on real world POI datasets. FAGI aims at facilitating both expert and non-expert stakeholders of the POI value chain into fusing batch amounts of linked POIs. The core usage of the software consists in a very simple workflow: (i) the user provides a minimal input configuration for the execution of FAGI; (ii) the user provides link validation and fusion rules within the rule specification of FAGI; (iii) the user deploys FAGI with input the two aforementioned configuration files and obtains the fusion results, which might consist in four different files, depending on the selected dataset output mode, as prescribed in Sections 4.2.1 and 7.2.1. In what follows, this workflow is demonstrated in detail.

7.3.1. FAGI input configuration

The first step for the user is to provide the configuration of FAGI, following the instructions of Section 7.2.1. As an example, the following configuration can be produced:

```
<?xml version="1.0" encoding="UTF-8"?>
<specification>
        <inputFormat>NT</inputFormat>
        <outputFormat>NT</outputFormat>
        <locale>EN</locale>
        <similarity>jarowinkler</similarity>
        <verbose>false</verbose>
        <rules>./rulespecs.xml</rules>
        <left>
                <id>datasetA</id>
                <endpoint></endpoint>
                <file>G:\DATA\RDF\TomTom_MultiNet_Austria_RDF_v1.4\TomTom_mnpoi_Austria.nt</file>
                <categories></categories>
        </left>
        <right>
                <id>datasetB</id>
                <endpoint></endpoint>
                <file>G:\DATA\RDF\Herold_RDF_v.0.4\public.herold.nt</file>
                <categories></categories>
        </right>
        <links>
                <id>links</id>
                <endpoint></endpoint>
                <linksFormat>csv-unique-links</linksFormat>
                <file>G:\DATA\RDF\links.csv</file>
        </links>
        <target>
                <id>fused</id>
                <mode>aa_mode</mode>
                <outputDir> G:\DATA\RDF\output</outputDir>
                <fused></fused>
                <remaining></remaining>
                <ambiguous></ambiguous>
                <statistics></statistics>
                <fusionLog></fusionLog>
        </target>
        <ML>
                <name></name>
                 <address></address>
                 <website></website>
                 <phone></phone>
                 <email></email>
        </ML>
</specification>
```

In this example, we have set the paths of the RDF input files, the path for the rule specification file, the output directory of the results, the locale of the input datasets and the dataset output mode. The latter prescribes that only the linked/fused POIs along with the unlinked POIs from the left datasets are maintained in the fused, output dataset. This configuration can either be provided directly in the configuration specification file, or be filled in through the graphical user interface of FAGI. Also, the links will be processed in order to result only unique links between the POIs, based on the link scores, and the ML-component is not utilized for the simplicity of the example.

7.3.2. FAGI validation and fusion rule specification

The second step is to define validation and fusion rules in the rule specification XML file. In this scenario, we are interested in validating links by checking similarities on their name property values. We wish to define two validation conditions with the following actions: (i) reject links with POI name similarities (normalized) less than 0.5, and (ii) accept and mark as ambiguous all links that have POI name similarities between [0.5,0.7) and their phone number values are the same. The priority of the validation rules (according to their order of definition in the file) will ensure that validation action (ii) will be applied only to POIs with name similarities above 0.5. The prescribed validation specification is presented below:

```
<?xml version="1.0" encoding="UTF-8"?>
<rules>
  <validationRule>
    <defaultAction>accept</defaultAction>
    <actionRuleSet>
      <actionRule>
        <action>reject</action>
        <condition>
          <expression>
            <not>
              <function>isSameCustomNormalize(a0, b0, 0.5)</function>
            </not>
          </expression>
        </condition>
      </actionRule>
      <actionRule>
        <action>accept-mark-ambiguous</action>
        <condition>
          <expression>
            <and>
                        <function>isSamePhoneNumber(a0, b0)</function>
              <expression>
                <not>
                  <function>isSameCustomNormalize(a1, b1, 0.7)</function>
                </not>
              </expression>
            </and>
          </expression>
        </condition>
      </actionRule>
    </actionRuleSet>
    <externalProperty id="a0">http://slipo.eu/def#phone http://slipo.eu/def#contactValue
    </externalProperty>
    <externalProperty id="b0">http://slipo.eu/def#phone http://slipo.eu/def#contactValue
    </externalProperty>
    <externalProperty id="a1">http://slipo.eu/def#name http://slipo.eu/def#nameValue
    </externalProperty>
    <externalProperty id="b1">http://slipo.eu/def#name http://slipo.eu/def#nameValue
    </externalProperty>
  </validationRule>
```

The above validation specification structure can be provided directly in the configuration specification file; however, the syntax may become quite complicated depending on the rules we define. For this purpose, we have developed a web interface for constructing more complicated rule specifications. The above XML file can be easily constructed using the UI and defining rules, through the selection of properties, conditions functions, condition expression operators and actions, as seen in Figure 29:

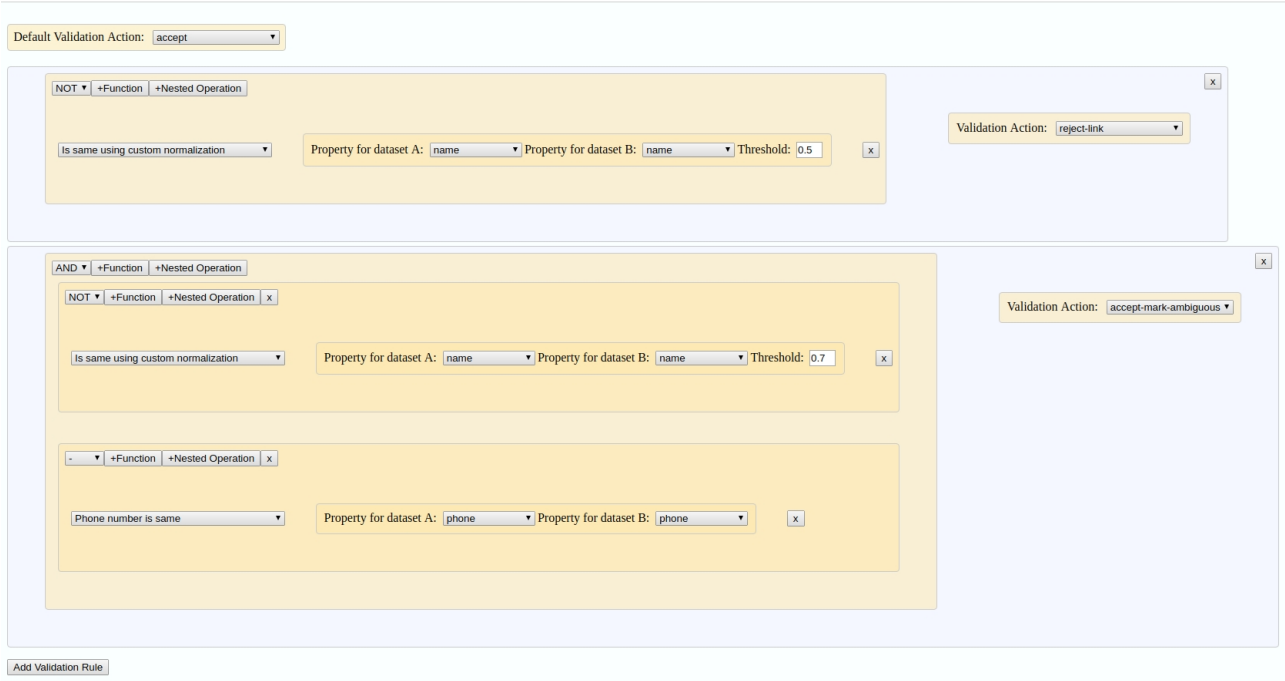


Figure 29: UI for validation rule specification

Regarding fusion, we wish to define a rule prescribing the fusion of the POI name property values, as well as a rule prescribing the fusion of phone number values. All other properties should be kept from the left dataset, thus we define "keep-left" as the default dataset-level action. For the first action rule (fusing names), we want to *keep the longer value* when the POI name values are at least 0.8 similar, their address's streets are almost identical (0.9 and larger) and their street numbers are equal. If this first action rule is not met, we define a secondary action rule that does not check the address values of the POIs, but instead it *keeps the left value* if the names are almost exactly the same (using normalization) and the left value does not contain an abbreviation. For all the remaining cases (when none of the defined action rules apply), we *keep both* name values (default action).

The above rule is realized in the XML specification syntax as presented in the listing below. In this, all the external properties that are used in the condition functions are defined at the end of the rule. Each of the two action rules includes a different fusion action and a different condition expression according to what was described in the previous paragraph. The rule also includes a default fusion action, in case no action rule is satisfied.

```
<rule>
    <propertyA>http://slipo.eu/def#name http://slipo.eu/def#nameValue</propertyA>
    <propertyB>http://slipo.eu/def#name http://slipo.eu/def#nameValue</propertyB>
    <defaultAction>keep-both</defaultAction>
    <actionRuleSet>
      <actionRule>
        <action>keep-left</action>
        <condition>
          <expression>
            <and>
              <function>isSameCustomNormalize(a2, b2, 0.8)</function>
              <function>isSameCustomNormalize(a3, b3, 0.9)</function>
              <function>isSameCustomNormalize(a4, b4, 1.0)</function>
            </and>
          </expression>
```

```
      </condition>
    </actionRule>
    <actionRule>
      <action>keep-right</action>
      <condition>
        <expression>
          <and>
                      <function>isSameCustomNormalize(a5, b5, 0.9)</function>
            <expression>
              <not>
                <function>isLiteralAbbreviation(b2)</function>
              </not>
            </expression>
          </and>
        </expression>
      </condition>
    </actionRule>
  </actionRuleSet>
  <externalProperty id="a2">http://slipo.eu/def#name http://slipo.eu/def#nameValue
  </externalProperty>
  <externalProperty id="b2">http://slipo.eu/def#name http://slipo.eu/def#nameValue
  </externalProperty>
  <externalProperty id="a3">http://slipo.eu/def#address http://slipo.eu/def#street
  </externalProperty>
  <externalProperty id="b3">http://slipo.eu/def#address http://slipo.eu/def#street
  </externalProperty>
  <externalProperty id="a4">http://slipo.eu/def#address http://slipo.eu/def#number
  </externalProperty>
  <externalProperty id="b4">http://slipo.eu/def#address http://slipo.eu/def#number
  </externalProperty>
  <externalProperty id="a5">http://slipo.eu/def#name http://slipo.eu/def#nameValue
  </externalProperty>
  <externalProperty id="b5">http://slipo.eu/def#name http://slipo.eu/def#nameValue
  </externalProperty>
</rule>
```

Similarly to the validation specifications, the user may use the graphical web interface of FAGI to easily construct the fusion rule of the above listing. Specifically, the user only needs to select the pair of matching properties and a default fusion action, and then start the construction the fusion action rules by selecting condition functions, operators and fusion actions from drop-down lists, that contain all supported functionality of FAGI. Figure 30 and Figure 31 (one for each of the two action rules) present the equivalent rule specification of the above listing, constructed through the web interface.

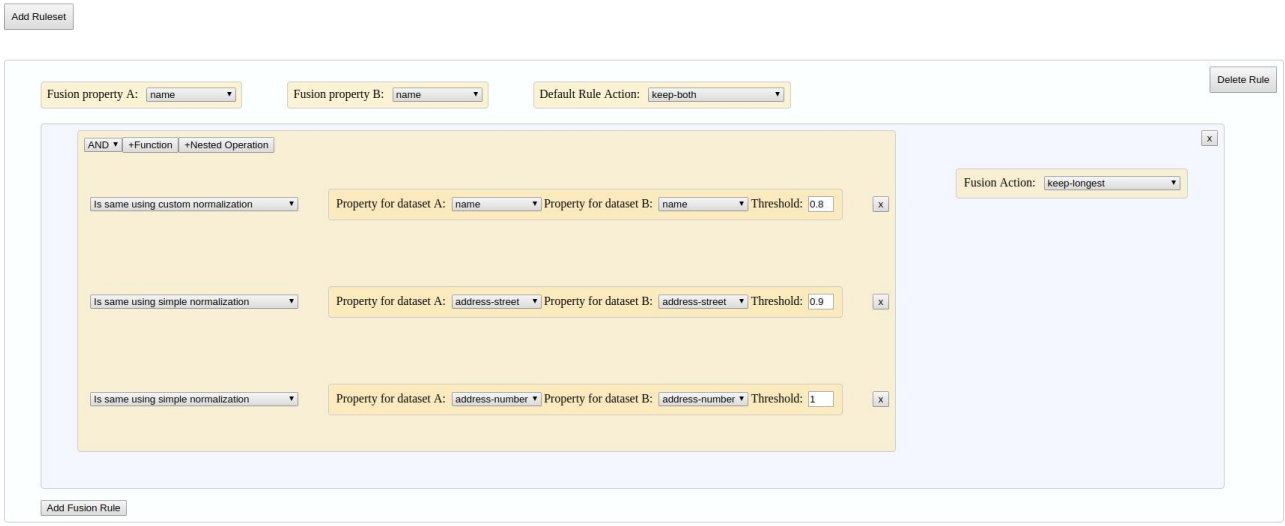


Figure 30: UI for fusion rule specification on POI names (first action rule)

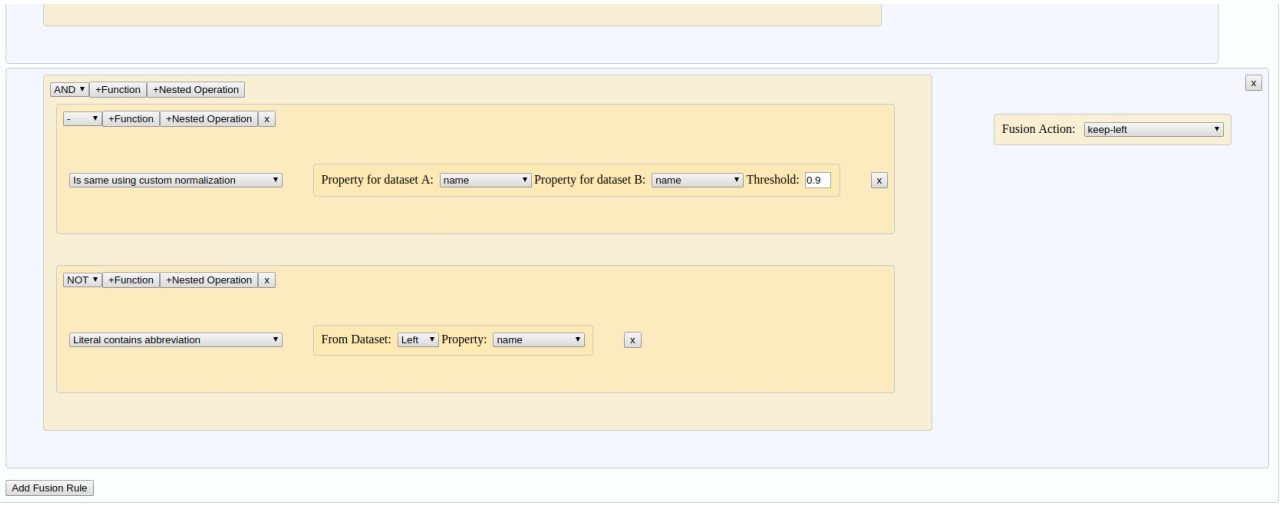


Figure 31: UI for fusion rule specification on POI names (second action rule)

Regarding the fusion of phone number values, we wish to *keep both* phone numbers as a default action. However, in case the phones are the same and their respective POI names are quite similar we only *keep the left* phone number value. This specification can be defined as follows in the rule specification file:

```
<rule>
  <propertyA>http://slipo.eu/def#phone http://slipo.eu/def#contactValue</propertyA>
  <propertyB>http://slipo.eu/def#phone http://slipo.eu/def#contactValue</propertyB>
  <defaultAction>keep-both</defaultAction>
  <actionRuleSet>
    <actionRule>
      <action>keep-left</action>
      <condition>
        <expression>
          <and>
            <function>isSameSimpleNormalize(a6, b6, 0.9)</function>
            <function>isSamePhoneNumber(a7, b7)</function>
          </and>
        </expression>
      </condition>
    </actionRule>
```

```
    </actionRuleSet>
    <externalProperty id="a6">http://slipo.eu/def#name http://slipo.eu/def#nameValue
    </externalProperty>
    <externalProperty id="b6">http://slipo.eu/def#name http://slipo.eu/def#nameValue
    </externalProperty>
    <externalProperty id="a7">http://slipo.eu/def#phone http://slipo.eu/def#contactValue
    </externalProperty>
    <externalProperty id="b7">http://slipo.eu/def#phone http://slipo.eu/def#contactValue
    </externalProperty>
  </rule>
  <defaultDatasetAction>keep-left</defaultDatasetAction>
</rules>
```

Finally, the equivalent fusion rule for phone properties can be constructed through the web interface as presented in Figure 32.

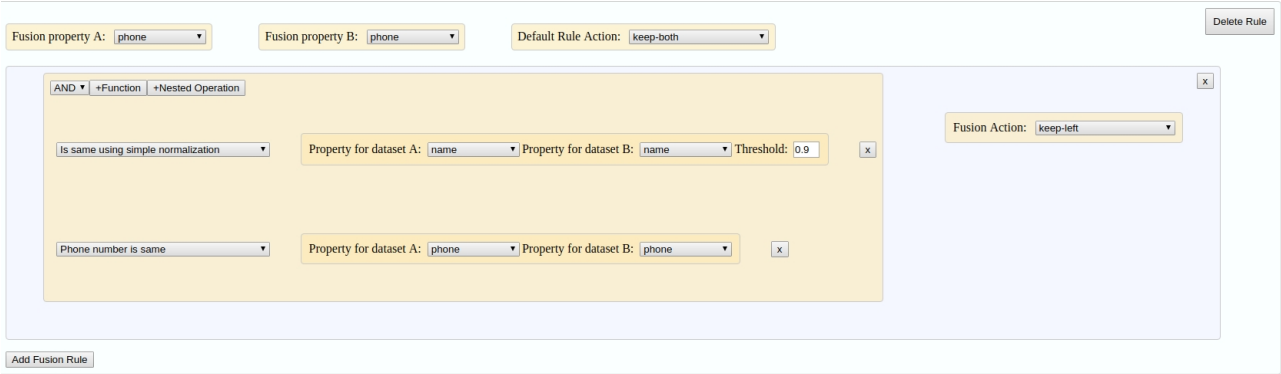


Figure 32: UI for fusion rule specification on POI phones

7.3.3. FAGI execution

Upon the configuration and rule specification files are prepared, the user can deploy FAGI. The deployment can be performed either through the SLIPO Workbench, within a designed integration workflow, or using the instructions of Section 7.1.1 for the command line version or 7.1.2 for the web interface. The output is produced in the directory prescribed in the configuration file and consists of an RDF file containing the fusion results as described in Section 4.2.1.

8. References

[AEGLE] K. Georgala, M. A. Sherif, A. N. Ngomo, An efficient approach for the generation of allen relations, in: ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016), 2016, pp. 948–956.

[AGREM] I. F. Cruz, F. P. Antonelli, C. Stroe, Agreement maker: Efficient matching for large real-world schemas and ontologies, PVLDB 2 (2) (2009) 1586–1589.

[BN08] Bleiholder, Jens and Naumann, Felix. Data fusion. In ACM Comput. Surv., v. 41:1, pp. 1-41, 2008.

[CCM+98] Cobb, Maria A., and Chung, Miyi J., and Miller, Vincent, and Foley, Harold III, and Petry, Frederick E., and Shaw, Kevin B. A Rule-Based Approach for the Conflation of Attributed Vector Data. GeoInformatica, v. 2:1, pp. 7–35, 1998.

[CK08] Chen Ching-Chien, Knoblock Craig A. Conflation of Geospatial Data. Encyclopedia of GIS, pp. 133-140, 2008.

[COALA] A. N. Ngomo, K. Lyko, V. Christen, COALA - correlation-aware active learning of link specifications, in: The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC 2013, Montpellier, France, May 26-30, 2013. Proceedings, 2013, pp. 442–456.

[CODI] J. Tang, B. Liang, J. Li, K. Wang, Risk minimization-based ontology mapping, in: Content Computing, Advanced Workshop on Content Computing, AWCC 2004, ZhenJiang, JiangSu, China, November 15-17, 2004, Proceedings, 2004, pp. 469–480.

[CSK+06] Chen, Ching-Chien, and Shahabi, Cyrus, and Knoblock, Craig A., and Kolahdouzan, Mohammad. Automatically and Efficiently Matching Road Networks with Spatial Attributes in Unknown Geometry Systems. In Proceedings of the Third Workshop on Spatio-temporal Database Management (co-located with VLDB2006), 2006.

[DEER] M. A. Sherif, A. N. Ngomo, J. Lehmann, Automating RDF dataset transformation and enrichment, in: The Semantic Web. Latest Advances andNew Domains - 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Proceedings, 2015, pp. 371–387.

[DPSO] M. A. Sherif, A. N. Ngomo, An optimization approach for load balancing in parallel link discovery, in: Proceedings of the 11th International Conference on Semantic Systems, SEMANTICS 2015, Vienna, Austria, September 15-17, 2015, 2015, pp. 161–168.

[EAGLE] A. N. Ngomo, K. Lyko, EAGLE: efficient active learning of link specifications using genetic programming, in: The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings, 2012,

[FAGI-api] Athena R.C. FAGI API: https://slipo-eu.github.io/FAGI/apidocs/index.html

[FAGI-doc] Athena R.C. FAGI documentation: https://slipo-eu.github.io/FAGI/dependencies.html

[FAGI-web] Athena R.C. FAGI-web. Java source code: https://github.com/SLIPO-EU/fagi-web

[FAGI] Athena R.C. FAGI. Java source code: https://github.com/SLIPO-EU/FAGI

[FAGISPEC] Athena R.C. FAGI specification constants: https://github.com/SLIPO-EU/FAGI/blob/master/src/main/java/gr/athena/innovation/fagi/specification/SpecificationConstants.java

[GeoKnow] GeoKnow EU/FP7 project. http://geoknow.eu/

[GeoKnowD323] GeoKnow EU/FP7 project. D3.2.3 Fusing of geospatial relations.

[GRR+12] A. N. Ngomo, Link discovery with guaranteed reduction ratio in affine spaces with Minkowski measures, in: The Semantic Web - ISWC2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I, 2012, pp. 378–393.

[GUI+16] A.-C. Ngonga Ngomo, D. Obraczka, K. Georgala, Using machine learning for link discovery on the web of data, in: Demos at the European Conference on Artificial Intelligence, 2016.

[GVK+15] Giorgos Giannopoulos, Nick Vitsas, Nikos Karagiannakis, Dimitrios Skoutas, Spiros Athanasiou FAGI-gis: A Tool for Fusing Geospatial RDF Data. In The Semantic Web: ESWC 2015 Satellite Events, 2015.

[Knofuss] A. Nikolov, V. S. Uren, E. Motta, Knofuss: a comprehensive architecture for knowledge fusion, in: Proceedings of the 4th International Conference on Knowledge Capture (K-CAP 2007), October 28-31, 2007, Whistler, BC, Canada, 2007, pp. 185–186.

[LBB17] Oliver Lehmberg, Christian Bizer, Alexander Brinkmann: WInte.r - A Web Data Integration Framework. International Semantic Web Conference (Posters, Demos & Industry Tracks), 2017.

[LCG+13] M. Saleem, S. S. Padmanabhuni, A. N. Ngomo, J. S. Almeida, S. Decker, H. F. Deus, Linked cancer genome atlas database, in: M. Sabou, E. Blomqvist, T. D. Noia, H. Sack, T. Pellegrini (Eds.), I-SEMANTICS 2013 -9th International Conference on Semantic Systems, ISEM '13, Graz, Austria, September 4-6, 2013, ACM, 2013, pp. 129–134.

[LDS+17] M. Nentwig, M. Hartung, A. N. Ngomo, E. Rahm, A survey of current link discovery frameworks, Semantic Web 8 (3) (2017) 419–436.

[LGD+12]C. Stadler, J. Lehmann, K. Höffner, S. Auer, Linkedgeodata: A core for a web of spatial open data, Semantic Web 3 (4) (2012) 333–354.

[LIMES+11]A. N. Ngomo, S. Auer, LIMES - A time-efficient approach for large scale link discovery on the web of data, in: T. Walsh (Ed.), IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011, IJCAI/AAAI, 2011,pp. 2312–2317.

[LOGMAP] E. Jiménez-Ruiz, B. C. Grau, LogMap: Logic-based and scalable ontology matching, in: The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I, 2011, pp. 273–288.

[LTSM] A.-C. N. Ngomo, M. M. Hassan, The lazy traveling salesman – memory management for large-scale link discovery., in: H. Sack, E. Blomqvist, M. d'Aquin, C. Ghidini, S. P. Ponzetto, C. Lange (Eds.), ESWC, Vol. 9678 of Lecture Notes in Computer Science, Springer, 2016, pp. 423–438.

[MAVEN] Apache Maven. http://maven.apache.org/

[MK+12] Michelfeit, J. and Knap, T. Linked Data Fusion in ODCleanStore. In Proc. of the International Semantic Web Conference (Posters & Demos), 2012.

[MKTI] J. F. Allen, Maintaining knowledge about temporal intervals, Commun. ACM 26 (11) (1983) 832–843.

[MMB12] Mendes, P. N. and Muhleisen, H. and Bizer, C. Sieve: linked data quality assessment and fusion. In Proc. of the Joint EDBT/ICDT Workshops, pp. 116-123, 2012.

[MONGE] The field matching problem: Algorithms and applications. Second International Conference on Knowledge Discovery and Data Mining (KDD), 1996.

[OLDHA] A. N. Ngomo, On link discovery using a hybrid approach, J. Data Semantics 1 (4) (2012) 203–217. doi:10.1007/s13740-012-0012-y.
URL http://dx.doi.org/10.1007/s13740-012-0012-y

[OM+11] P. Shvaiko, J. Euzenat, T. Heath, C. Quix, M. Mao, I. F. Cruz (Eds.), Proceedings of the 6th International Workshop on Ontology Matching, Bonn, Germany, October 24, 2011, Vol. 814 of CEUR Workshop Proceedings, CEUR-WS.org, 2011

[ORCHID] A. N. Ngomo, ORCHID - reduction-ratio-optimal computation of geospatial distances for link discovery, in: Alani et al.

[PSC+16] Comparative Analysis of Approximate Blocking Techniques for Entity Resolution. Proceedings of the VLDB Endowment, 2017

[PSDM] M. A. Sherif, A.-C. N. Ngomo, A Systematic Survey of Point Set Distance Measures for Link Discovery, Semantic Web Journal.

[RADON] M. A. Sherif, K. Dreßler, P. Smeros, A. N. Ngomo, Radon - rapid discovery of topological relations, in: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA., 2017, pp. 175–181.

[RADON2] RADON2: A buffered-Intersection Matrix Computing Approach To Accelerate Link Discovery Over Geo-Spatial RDF Knowledge Bases (OAEI2018 Results) by Mohamed Ahmed Sherif and Axel-Cyrille Ngonga Ngomo Abdullah Fathi Ahmed in Proceedings of Ontology Matching Workshop 2018.

[RAVEN] A. N. Ngomo, J. Lehmann, S. Auer, K. Höffner, RAVEN - active learning of link specifications, in: Shvaiko et al. [53], pp. 25–36.

[REWED] T. Soru, A. N. Ngomo, Rapid execution of weighted edit distances, in:Shvaiko et al. [52], pp. 25–36.

[ROCKER] T. Soru, E. Marx, A. N. Ngomo, ROCKER: A refinement operator for key discovery, in: Proceedings of the 24th International Conference onWorld Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015, 2015, pp. 1025–1033.

[Saa88] Saalfeld, Alan. Conflation: Automated Map Compilation. Int. J. Geogr. Inf. Sci. v. 2:3, pp. 217–228, 1988.

[SCIK] scikit-learn - Machine Learning in Python. http://scikit-learn.org/stable/index.html

[SERIMI] S. Araújo, J. Hidders, D. Schwabe, A. P. de Vries, SERIMI – resource description similarity, RDF instance matching and interlinking, in: Proceedings of the 6th International Workshop on Ontology Matching, Bonn, Germany, October 24, 2011, 2011.

[SFM17] Learning to combine multiple string similarity metrics for effective toponym matching. International Journal of Digital Earth, 2017.

[SILK] R. Isele, A. Jentzsch, C. Bizer, Efficient multidimensional blocking for link discovery without losing recall, in: Proceedings of the 14thInternational Workshop on the Web and Databases 2011, WebDB 2011, Athens, Greece, June 12, 2011, 2011.

[SLINT+] K. Nguyen, R. Ichise, B. Le, SLINT: a schema-independent linked data interlinking system, in: Proceedings of the 7th International Workshop on Ontology Matching, Boston, MA, USA, November 11, 2012, 2012.

[SLIPOD11] SLIPO EU/H2020 project. D1.1 Use Cases and Requirements.

[SLIPOD12] SLIPO EU/H2020 project. D1.2 Architecture.

[SLIPOD13] SLIPO EU/H2020 project. D1.3 Beta SLIPO integrated system.

[SLIPOD22] SLIPO EU/H2020 project. D2.2 Mapping specification and POI transformation service.

[SLIPOD31] SLIPO EU/H2020 project. D3.1 Interlinking.

[SMI+12] Schultz, A. and Matteini, A. and Isele, R. and Mendes, P. and Bizer, C. and Becker, C. LDIFA Framework for Large-Scale Linked Data Integration. In WWW, developer track, 2012.

[UERMTG] L. Zhu, M. Ghasemi-Gol, P. A. Szekely, A. Galstyan, C. A. Knoblock, Unsupervised entity resolution on multi-type graphs, in: The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I, 2016, pp. 649–667.

[ULLS] A. N. Ngomo, K. Lyko, Unsupervised learning of link specifications: deterministic vs. non-deterministic, in: Shvaiko et al. [52], pp. 1–12.

[WF99] Walter, Volker, and Fritsch, Dieter. Matching Spatial Data Sets: a Statistical Approach. Int. J. Geogr. Inf. Sci. v. 5:1, pp. 445--473, 1999.

[WOMBAT] M. Sherif, A.-C. Ngonga Ngomo, J. Lehmann, WOMBAT - A Generalization Approach for Automatic Link Discovery, in: 14th Extended Semantic Web Conference, Portorož, Slovenia, 28th May - 1st June 2017, Springer,2017.

[WTRTC] A.-C. Ngonga Ngomo, L. Kolb, N. Heino, M. Hartung, S. Auer, E. Rahm, When to reach for the cloud: Using parallel hardware for link discovery.,in: Proceedings of ESCW, 2013.

[ZHISHI.LINKS] X. Niu, S. Rong, Y. Zhang, H. Wang, Zhishi. links results for OAEI 2011, in: Shvaiko et al. [OM11].

9. Annex

9.1. Datasets characteristics

9.1.1. Dataset A

Dataset A includes 312,385 TomTom POIs for Austria. The dataset is initially provided as a Shapefile in the WGS84 reference system. The schema of the dataset, i.e. the properties of the contained POIs are provided in the table below.

Property Name	Description
OBJECTID	POI ID
NAME	POI name
A_NAME	Alternative POI name
STR	Street name
HNR	Street number
ZIP	ZIP code
A00	Country code
A08_NAME	city/place
LABEL_DE	Description of POI (ie class)
LABEL_EN	Description of POI (ie class)
PHONE	Telephone
BRAND	Brand name for chains
ACCURACY	Geographic accuracy
SUBCAT	Category by TomTom
SUBNAME_DE	Classification by TomTom
SUBNAME_EN	Classification by TomTom

9.1.2. Dataset B

Dataset B contains all company POIs available in the Herold Business Data for Austria database, summing to 350,053 records. The dataset is initially provided as a Shapefile in the WGS84 reference system. The schema of the dataset, i.e. the properties of the contained POIs are provided in the table below.

Property Name	Description
SID	Company ID
FIRMA	Company name
BR	Branch of business

PLZ	ZIP code
ORT	City/place
STRASSE	Street name
HNR	House number
BL	Country
TELEFON	Phone
TELEFAX	Fax
HTTP	Homepage
EMAIL	Email
OENACE	Categorization
OENACE_ERW	Categorization
OENACE_BEZ	Categorization
BR_ID	HEROLD's branch of business ID
HIC	Branch group