REPORT ON DELIVERABLE D3.2

Initial release of the POI enrichment framework

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.

Dissemination Level	Public
Due Date of Deliverable	Month 18, 30/06/2018
Actual Submission Date	28/06/2018
Work Package	WP3 - Data Integration and Quality Assurance
Tasks	3.2 Enrichment
Type	Other
Approval Status	Submitted for approval
Version	1.0
Number of Pages	44
Filename	D3.2_Initial_release_of_the_POI_enrichment_framework.pdf

Abstract

This document presents DEER, our software framework for the enrichment of POIs. First, we briefly describe the current state of the art in dataset enrichment, as well as the initial versions of the DEER software, which comprised the starting point of our work. Then, we present DEER v1.0.0, the current version optimized for scalable and quality assured enrichment of POIs. Apart from implementation and deployment information, we thoroughly describe the POI-specific enrichment approaches we developed and tested against real-world, commercial POI datasets. Finally, we present our evaluation experiments assessing both the effectiveness and the scalability of the software.

History

version	date	reason	revised by
0.1	15/05/2018	Deliverable structure and initial content	Mohamed Sherif
0.2	22/05/2018	Added content on DEER components and architecture section	Mohamed Sherif
0.3	30/05/2018	First draft	Mohamed Sherif
0.4	01/06/2018	Multiple revisions and improvements	Mohamed Sherif
0.5	26/06/2018	Added evaluation section	Kevin Dreßler
0.6	12/06/2018	Multiple revisions and improvements	Mohamed Sherif, Kevin Dreßler
0.7	26/06/2018	Reviewed version	Giorgos Giannopoulos, Spiros Athanasiou
1.0	28/06/2018	Final version	Mohamed Sherif, Kevin Dreßler, Axel Ngonga

Author list

organization	name	contact information
INFAI	Mohamed Sherif	sherif@informatik.uni-leipzig.de
INFAI	Kevin Dreßler	dressler@informatik.uni-leipzig.de
INFAI	Axel Ngonga	ngonga@informatik.uni-leipzig.de
ATHENA RC	Spiros Athanasiou	spathan@imis.athena-innovation.gr
ATHENA RC	Giorgos Giannopoulos	giann@imis.athena-innovation.gr

Executive Summary

This document presents DEER, our framework for 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. 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. For example, if we have a source knowledge base that contains information about POI resources in form of hotels, it may be missing information about the near-by restaurants and their respective opening hours. The goal of our enrichment framework DEER would be to extract such missing information (in a form of RDF triples) from another POI knowledge base and insert it to our 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*. For instance, using the entity-recognition-based enrichment, our framework is able to extract embedded information within natural-language text description of the POI resources and made it explicitly machine interoperable/processable in RDF format. For our previous example, our entity-recognition-based enrichment can extract information about the origin of the offered meals (e.g., Mexican, Chinese) from the restaurants' menus (represented in a form of natural text).

Furthermore, DEER incorporates mechanisms for quality statistics/indicators extraction to facilitate and assure the quality of the POI enrichment process.

The presented version of DEER extends the initial framework (DEER v0.5.0), which was developed during the *GeoKnow* project for the enrichment of spatial resources (*original project dubbed GeoLift)*. The current version, DEER v1.0.0, developed in the context of SLIPO, has been extended to specialize on the effective and scalable enrichment of POI entities. To this end, in order to implement the core mechanisms for POI enrichment, we first performed an analysis of real world, commercial POI datasets. The analysis engaged the industrial partners of the project. This analysis and iterative experimentation allowed us to fine-tune the enrichment mechanisms on real world data.

The layout of document is the following.

In Section 1, we introduce the setting of the enrichment task. We first describe the objectives of enrichment in the frame of SLIPO project, and then we provide some background knowledge, briefly presenting existing works on enrichment of geospatial entities.

In Section 2, we review the roadmap for the development of DEER, and present the evolution of the software from its initial version at the beginning of SLIPO. Also, we summarize the major advancements of the software during the first period of the project, and the key features of DEER v2.0.0.

In Section 3, we present a complete user's guide for DEER v1.0.0, including building and installation instructions, configuration settings, and a short demonstration of the usage of the software.

Finally, in Section 4, we evaluate the new POI enrichment approaches implemented DEER.

Abbreviations and Acronyms

API	Application Programming Interface
CRS	Coordinate Reference System
KPI	Key Performance Indicator
ML	Machine Learning
RDF	Resource Description Framework
UI	User Interface
VM	Virtual Machine
LD	Link Discovery
LS	Link Specification
CLI	Command-Line Interface
NER	Named Entity Recognition
AEO	Atomic Enrichment Operators

Table of Contents

1. Introduction

In this section, we first discuss the basic concepts and goals of POI data enrichment. Next, we present a formal overview of the POI enrichment problem. Then, we present the current state of the art in POI enrichment approaches and frameworks. Finally, we briefly present the rule of our POI enrichment component DEER within the SLIPO lifecycle.

1.1. POI data Enrichment

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.

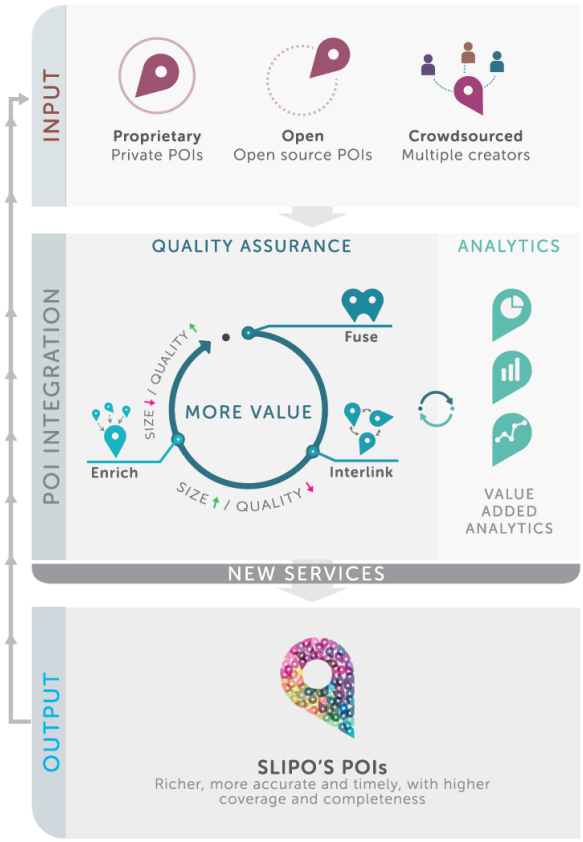


Figure 1: POI integration lifecycle

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.

Considering the big picture of the POI integration lifecycle (Figure 1), the enrichment process is tightly interconnected with validation and quality assurance. To this end, the enrichment process needs to incorporate several mechanisms to assess the quality of the proposed enrichment operations and their results.

1.2. 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.

1.3. 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 [13], Linked Education [6] and all forms of semantic mashups [9].

In recent work, several challenges and requirements data consumption and integration have been pointed out [14]. For example, the **R2R framework** [2] addresses those by allowing to publish mappings across knowledge bases that allow to map classes and 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. *T*

he Linked Data Integration Framework ***LDIF*** [21], whose goal is to support the integration of RDF data, builds upon R2R mappings and technologies such as *SILK* [10] and *LDSpider*[1]. 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.

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

Semantic Web Pipes[2] [20] follows the idea of **Yahoo Pipe**[3] to enable the integration of data in formats such as RDF and XML. By using Semantic Web Pipes, users can efficiently create semantic mashups by using a number of operators (such as getRDF, getXML, etc.) and connecting these manually within a simple interface.

KnoFuss [19] addresses data integration from the point of view of link discovery. It begins by detecting URIs that stand for the same real-world entity and either merging them to one or linking them via `owl:sameAs`. In addition, it allows to monitor the interaction between instance and dataset matching (which is similar to ontology matching [7]).

Fluid Operations' Information Workbench[4] allows to search through, manipulate and integrate for purposes such as business intelligence.

The work in [5] describes a framework for semantic enrichment, ranking and integration of web videos, and [1] presents a semantic enrichment framework for *Twitter* posts. Finally, [8] tackles the linked data enrichment problem for sensor data via an approach that sees enrichment as a process driven by situations of interest.

1.4. 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 D1.3 "Beta SLIPO Integrated 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)).

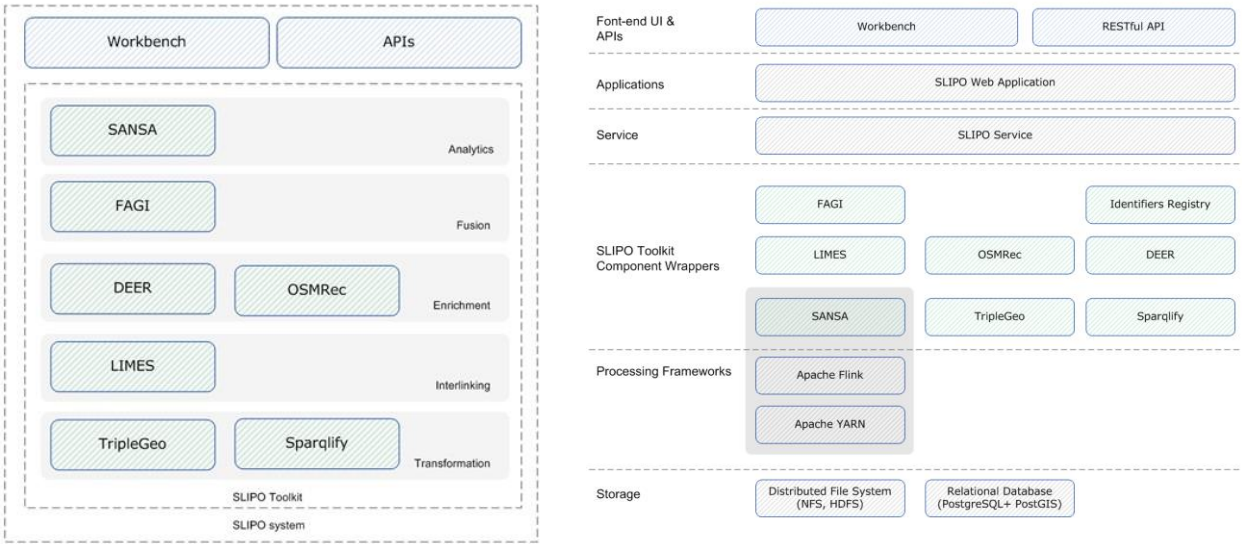
The goal of Task 3.2 is to deliver DEER, the POI enrichment framework of SLIPO. DEER incorporates many approaches for performing efficient enrichment among POI resources. In the context of SLIPO, DEER receives as input one or more RDF POI dataset(s) conforming to the SLIPO ontology. DEER's input POI 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 parameters. DEER's output consists in one or more files. The files contain the enriched versions of the input datasets. Figure 2 shows an overview of the SLIPO architecture.

---

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

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

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



(a) High level architecture (b) Logical architecture

Figure 2: SLIPO Architecture

2. The DEER Framework

In the following section, we start by giving a detailed introduction to the components of our framework. Then, we present our achievements within this reporting period towards the delivery of DEER v.1.0.0. Finally, we describe the planned extensions on DEER in the context of the SLIPO project towards DEER v2.0.0.

2.1. DEER Components

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 artifacts: **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 artifacts to meet his/her needs.

2.1.1. 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.

2.1.2. 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.

2.1.3. Enrichment Operators

By enrichment operators we mean these artifacts 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.

2.1.3.1. 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)[5] of Berlin

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

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

2.1.3.2. 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 (*see Deliverable D3.1 "Initial Release of the POI interlinking software"*) 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.

2.1.3.3. 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 3. 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.

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

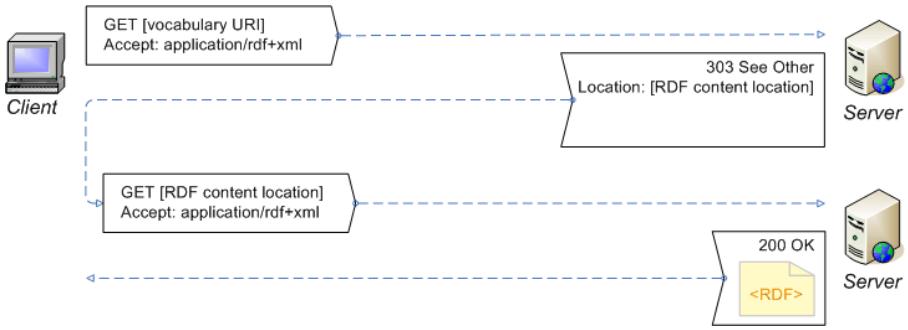


Figure 3: Content Negotiation as used by DEER (courtesy of W3C)

2.1.3.4. 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)

2.1.3.5. 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 artifact). Thus, DEER is able to execute all workflows of output datasets in parallel.

2.1.3.6. 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 artifact).

2.1.3.7. 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`.

2.1.3.8. 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`.

2.1.3.9. 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.

2.1.3.10. 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.

2.2. Achievements - DEER v1.0.0

DEER v1.0.0 is the first 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 first period of the SLIPO project.

2.2.1. 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 end-points.

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.

2.2.2. Non-linear enrichment pipeline

The previous DEER version was only able to provide linear enrichment pipelines. During the project, we implemented the *first non-linear enrichment pipelines execution engine* 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.

2.2.3. Automatic parallelization approach.

In the current version of DEER, we implemented the new *parallel execution engine* for the DEER execution DAG. Our parallelization approach applies *graph coloring* to the DEER execution DAG to find the parts of the DAG 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 recourses.

We also parallelize most of the DEER's enrichment operators internally in order to achieve the maximum utilization of the system where the 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.

2.2.4. POI-specific enrichment functions

In DEER v.1.0.0 we provide two novel POI-specific enrichment operators:

1. *The geo-fusion enrichment operator (see Section 2.1.3.9)* 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.
2. *The geo-distance enrichment operator (see Section 2.1.3.10)* 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.

2.3. Towards DEER v2.0.0

In this section, we describe the planned extensions on DEER in the context of the SLIPO project, towards evolving it into a platform for the efficient and effective enrichment of large datasets of linked POI entities. Towards this, we have revised and adapted our original development roadmap (see D1.2 Architecture, Section 2.4.2), introducing v1.0.0 as an interim version of the software, and increasing the version of the final anticipated software to 2.0.0 This inconsistency reflects the dynamic development of the software and the multitude advances we have managed to introduce already in the project's lifecycle. Overall, the proposed extensions focus on the following development pathways.

2.3.1. Learning of Non-linear Enrichment Pipelines

In DEER v2.0.0, we will extend up on the initial version of our enrichment pipeline learning algorithm provided in the current DEER version (v1.0.0). The extended non-linear enrichment pipe line learning will implement learning from existing data to ensure that high-quality datasets can be created by combining information from many heterogeneous sources.

The current machine learning implementation of DEER is only able to learn linear enrichment pipelines based on refinement operators. We will extend upon the current machine learning approach within DEER to deal with the new non-linear enrichment pipeline. Our machine learning approach will aim to assure the quality of the enriched datasets by monitoring a set of quality indicators such as F-measure.

2.3.2. Introducing POI-specific Enrichment Operator

We will extend the kernel of DEER to integrate POI-specific enrichment functions that will allow discovering attributes of POIs from sources of various structures ranging from textual, to structured.

In the current version of the framework, we provided the first version of two new POI-specific operators. the *geo-fusion enrichment operator* for fusing the geo-spatial representation of POI resources, and the *geo-distance operator* for enriching POI resources that are connected via a predefined predicate with the distance between their respective geo-coordinates. In the DEER v2.0.0, we will optimize the performance of the geo-distance operators to scale to millions of POI resources. Also, we will implement a self-configurator for the geo-distance operator. Moreover, in DEER v2.0.0, we will implement 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.

2.3.3. Parallel Execution Engine

We will extend our initial implementation of the parallel execution engine currently available in DEER v1.0.0 (*dubbed FARADAY-CAGE*). The new version of the FARADAY-CAGE will feature automatic validation of plugin parameters using SHACL[6] as well as an API to restructure the execution graph. This will be used in the next version of the learning algorithm and allows for rich tooling, such as caching of previous computation steps, in order to increase productivity and decrease waiting time for users of DEER.

2.4. Libraries and Frameworks

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

- Google Collections[7]: A set of core libraries that includes new collection types. The library has been replaced by Google Guava[8].
- FOX[9]: 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[10]: LIMES – Link Discovery Framework for Metric Spaces.
- Apache HttpComponents[11]: A toolset of low level Java components focused on HTTP and associated protocols.
- Apache Log4j: An open-source Java logging library.
- DL-Learner[12]: A tool for supervised Machine Learning in OWL and Description Logics.
- JSON-java[13]: 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.

---

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

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

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

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

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

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

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

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

- 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.
- FARADAY-CAGE[14] is the Framework for Acyclic Directed Graphs Yielding Parallel Computations of Great Efficiency.
- PF4J[15] 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.
-

2.5. License

DEER is an open source software and is available from GitHub[16] under the terms of the GNU AFFERO GENERAL PUBLIC LICENSE[17].

2.6. 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

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

15 https://github.com/pf4j/pf4j

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

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

3. Usage Manual for DEER

In this Section, we provide the usage manual for DEER v1.0.0. First, we give details on building the application from the Java source code. Next, we provide insights into the architecture of DEER and its parallel execution engine, called FARADAY-CAGE, before we provide instructions for the configuration of DEER. Finally, we present a short demonstration example on configuring and running DEER.

3.1. Building Installation

DEER v1.0.0 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[18] (or later) as well as Maven 3.3.3[19] (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.

3.1.1. 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`.

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

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

3.1.2. 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>
 <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>
```

3.2. DEER Architecture

DEER's former execution engine has been replaced by *FARADAY-CAGE*[20], 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 graphs edges.

This kind of graph is called an execution graph. It is dynamically built from an RDF specification by use of PF4J's plugin[21] 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 4.

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.

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

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

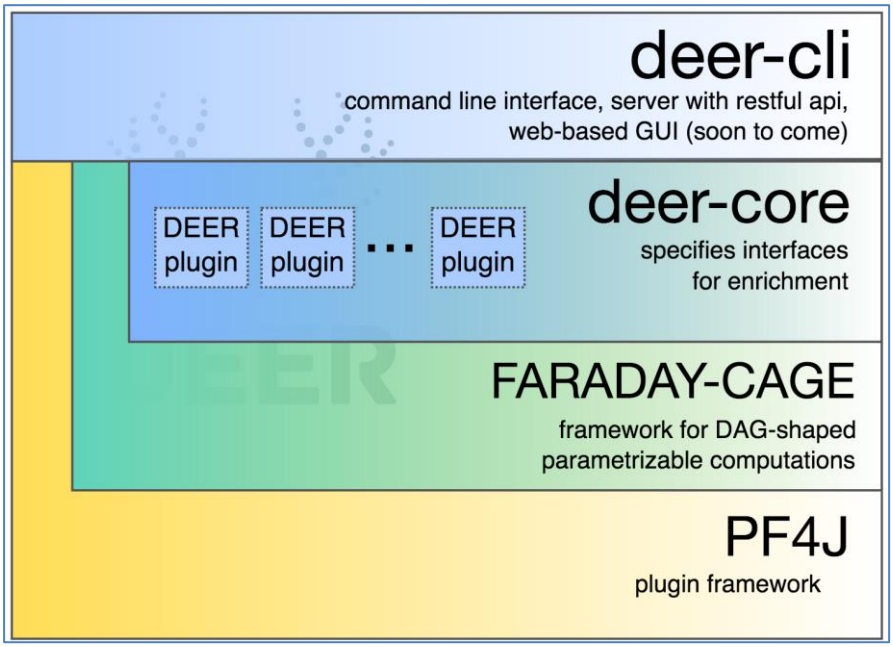


Figure 4: DEER architecture

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
  1. `minIn` is the minimum number of allowed inputs
  2. `maxIn` is the maximum number of allowed inputs
  3. `minIn` is the minimum number of allowed outputs
  4. `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. Configuring DEER

3.3.1. 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*.

There are two ways to connect execution nodes with edges:

1. explicitly by giving only `:hasOutput` declarations on the nodes having outgoing edges.
2. implicitly by matching `:hasOutput` and `:hasInput` declarations of endpoints on the nodes.

Execution nodes will be discovered automatically over the `:hasOutput` predicate.

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

- `:hasOutput` (required) declares the outgoing edges from this execution node. Allowed values are:
  1. another execution `node :e2` (if `:e1`s only edge goes to `:e2`)
  2. a list of other execution nodes (if :e1 has edges to each execution node in the list)
  3. a list of blank nodes (or resources) representing outgoing edges with the following properties:
     - `:toNode` the execution node this edge points to
     - `:toPort` the port of the execution node this edge points to
- `:hasInput` (deprecated) declares the incoming edges to this execution node. Allowed values are:
  1. another execution `node :e2` (if `:e1s` only edge comes from `:e2`)
  2. a list of other execution nodes (if `:e1` has edges leading to it from each execution node in the list)

- Note that, `:hasInput` needs to be declared only if there is more than one input and if the incoming edges have not been declared explicitly at their origin nodes. It is deprecated as explicitly declaring nodes will become the single standard in a future version.
- `:implementedIn` (for plugin nodes) declares the resource identifying the implementation of the plugin.

3.3.1.1. 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 :hasOutput :e2 .

:e2 :hasOutput ( :e3 :e4 ) .

:e3 :hasOutput ( [ :toNode :e4 ; :toPort 0 ]
                 [ :toNode :e5 ; :toPort 1 ]
                 [ :toNode :e5 ; :toPort 0 ] ) .

:e4 :hasInput ( :e3 :e2 ) ;
    :hasOutput ( [ :toNode :e5 ; :toPort 2 ] ) .

:e5 :implementedIn :somePluginClassIdentifier
```

The above example generates the following graph:

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

Note that, the edges declared between `:e3` and `:e5` require the explicit syntax, because with the implicit syntax double edges between nodes will be assigned to ports in order, i.e.,

```
:e3 :hasOutput ( :e5 :e5 ) .
:e5 :hasInput ( :e3 :e3 ) .
```

is equivalent to

```
:e3 :hasOutput  ( [ :toNode :e5 ; :toPort 0 ]
                  [ :toNode :e5 ; :toPort 1 ] ) .
```

Therefore, if edges need to be assigned to ports in a different order than the origin ports, explicit syntax needs to be used.

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.

3.3.2. DEER IO Plugins

3.3.2.1. 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.

3.3.2.2. 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.

3.3.3. 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)`.

3.3.3.1. 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 `:selectors`. The `:selectors` parameter accepts one list of selector types. 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://deer.aksw.org/vocabulary#> .
@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
              :implementedIn     :FilterEnrichmentOperator ;
              :hasInput          ( :node_in ) ;
              :hasOutput         ( :node_out ) ;
              :selectors         (
                    [ :predicate geo:lat ]
                    [ :predicate geo:long ]
                    [ :predicate rdfs:label ]
                    [ :predicate owl:sameAs ]
              ) .
```

3.3.3.2. 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)
    1. This mode will be selected if the linking enrichment operator node has exactly one input dataset.
    2. If it has exactly one output dataset, it will be the original input dataset enriched with the links.
    3. 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.
    4. 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)
    5. This mode will be selected if the linking enrichment operator node has exactly two input datasets.
    6. If it has exactly one output dataset, it will be the result of merging the two input datasets and the generated links

7. 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.
8. 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.
9. 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
              :implementedIn     :LinkingEnrichmentOperator ;
              :hasInput          ( :node_in ) ;
              :hasOutput         ( :node_out ) ;
              :specFile          "limes_specs.xml" ;
              :linksPart         "source" .
```

3.3.3.3. 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 `:operations`, which is a list of 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://deer.aksw.org/vocabulary#> .
 @prefix dbo: <http://dbpedia.org/property/> .
 @prefix dcterms: <http://purl.org/dc/terms/> .

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

3.3.3.4. 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.
-

3.3.3.5. Clone Enrichment Operator

The idea behind the clone operator is to enable parallel enrichment of multiple copies of the same dataset. The clone operator takes one dataset as input and produces n ≥ 2 output datasets, which are all identical to the input dataset.

In the following example, the clone operator is used to make 2 copies of the input dataset from `:node_1` into `:node_1_clone_1` and `:node_1_clone_2`

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

:node_clone
              :implementedIn     :CloneEnrichmentOperator
              :hasInput          ( :node_in ) ;
              :hasOutput         ( :node_in_clone_1  :node_in_clone_2) .
```

3.3.3.6. 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://deer.aksw.org/vocabulary#> .

:node_merge
              :implementedIn     :MergeEnrichmentOperator ;
              :hasInput          ( :node_in_1 :node_in_2) ;
              :hasOutput         ( :node_out) .
```

3.3.3.7. 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://deer.aksw.org/vocabulary#> .
@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
              :implementedIn     :GeoFusionEnrichmentOperator ;
              :hasInput          ( :node_in_1 :node_in_2 ) ;
              :hasOutput         ( :node_out ) ;
              :fusionAction      "takeAll" ;
```

```
              :mergeOtherStatements
                                  "true" .
```

3.3.3.8. 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 two basic parameters of `:sourceSubjectAuthority` and `:targetSubjectAuthority` 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://deer.aksw.org/vocabulary#> .

:node_a_conf
              :implementedIn     :AuthorityConformationEnrichmentOperator ;
              :hasInput          ( :node_in ) ;
              :hasOutput         ( :node_out ) ;
              :sourceSubjectAuthority
                                 "http://dbpedia.org" ;
              :targetSubjectAuthority
                                 "http://slipo.eu" .
```

3.3.3.9. 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 `:propertyMapping` for specifying the `:source` and `:target` predicates as a list.

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://deer.aksw.org/vocabulary#> .
@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
              :implementedIn     :PredicateConformationEnrichmentOperator ;
              :hasInput          ( :node_in ) ;
              :hasOutput         ( :node_out ) ;
              :propertyMapping    (
                    [ :source rdf:label ]
                    [ :target skos:prefLabel ]
              ) .
```

3.3.3.10. 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
              :implementedIn     :GeoDistanceEnrichmentOperator ;
              :hasInput          ( :node_in ) ;
              :hasOutput         ( :node_out ) ;
              :selectPredicate   dbp:west ;
              :distancePredicate :distanceToNextCityInTheWest .
```

3.3.4. Configuration File Example

The following example configuration demonstrates an example of a configuration file for DEER:

```
@prefix : <http://deer.aksw.org/vocabulary#> .
@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_reader1
              :implementedIn     :DefaultModelReader ;
              :fromUri           <http://de.dbpedia.org/resource/Paderborn> ;
              :useEndpoint       <http://de.dbpedia.org/sparql> ;
              :hasOutput         ( :node_conf ) .

:node_reader2
              :implementedIn     :DefaultModelReader ;
              :fromUri           <http://dbpedia.org/resource/Paderborn> ;
              :useEndpoint       <http://dbpedia.org/sparql> ;
              :hasOutput         ( :node_geofusion ) .

:node_conf
              :implementedIn     :AuthorityConformationEnrichmentOperator ;
              :hasInput          ( :node_reader1 ) ;
              :hasOutput         ( :node_geofusion ) ;
              :sourceSubjectAuthority
                                 "http://dbpedia.org" ;
              :targetSubjectAuthority
                                 "http://deer.org" .

:node_geofusion
              :implementedIn     :GeoFusionEnrichmentOperator ;
              :hasInput          ( :node_conf :node_reader2 ) ;
              :hasOutput         ( :node_filter ) ;
              :fusionAction      "takeAll" ;
              :mergeOtherStatements
```

```
                                "true" .

:node_filter
              :implementedIn     :FilterEnrichmentOperator ;
              :hasInput          ( :node_geofusion ) ;
              :hasOutput         ( :node_writer ) ;
              :selectors         (
                    [ :predicate geo:lat ]
                    [ :predicate geo:long ]
                    [ :predicate rdfs:label ]
                    [ :predicate owl:sameAs ]
              ) .

:node_writer
              :implementedIn     :DefaultModelWriter ;
              :outputFile        "output_demo.ttl" ;
              :outputFormat      "Turtle" ;
:hasInput ( :node_filter ) .
```

3.4. 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.

3.5. 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/`.

4. Experimental Evaluation

In this Section, we present the experiments that assess the efficiency and scalability of DEER. In our experiments, we evaluate both the running time of DEER, as well as the size of the datasets in the enrichment process which can act as a measure of the completeness of our enrichment configuration. First, we briefly present the POI datasets we used in our evaluation. Next, we present the enrichment configuration which we implemented exemplarily for the use case DINUC A.2, as specified in deliverable D5.1 "Pilot Specifications". The following experiment assesses the performance and completeness of the software for this enrichment configuration involving more than 5.5K and 13K POIs, respectively, and shows that DEER is able to directly target specific POIs and in enrich them with meaningful data from sources such as *DBpedia*.

4.1. Datasets and measures

4.1.1. Datasets

In our experiments, we use the *TomTom* and *OSM* datasets for *Bucharest, Romania*. The sizes of these datasets are depictured in Table 1.

Table 1: Evaluation Datasets

	POIs	Triples
TomTom Bucharest	5,899	186,739
OSM Bucharest	13,313	234,098

4.1.2. Measures

In the performed experiments, we measure run times (*in milliseconds*), for several parts of the enrichment process. Specifically, we report run times for each of the enrichment operators, model readers and model writers in the used DEER configuration graph.

Furthermore, we also measure the size of the intermediary datasets flowing through the graph of enrichment nodes. These numbers tell a story of how many POIs were selected and consecutively enriched in each step. The ratio between the sum of all combined output dataset sizes and the sum of all combined input dataset sizes effectively gives a measure how much new information was added to the POIs in the enrichment process.

4.2. Enrichment configuration

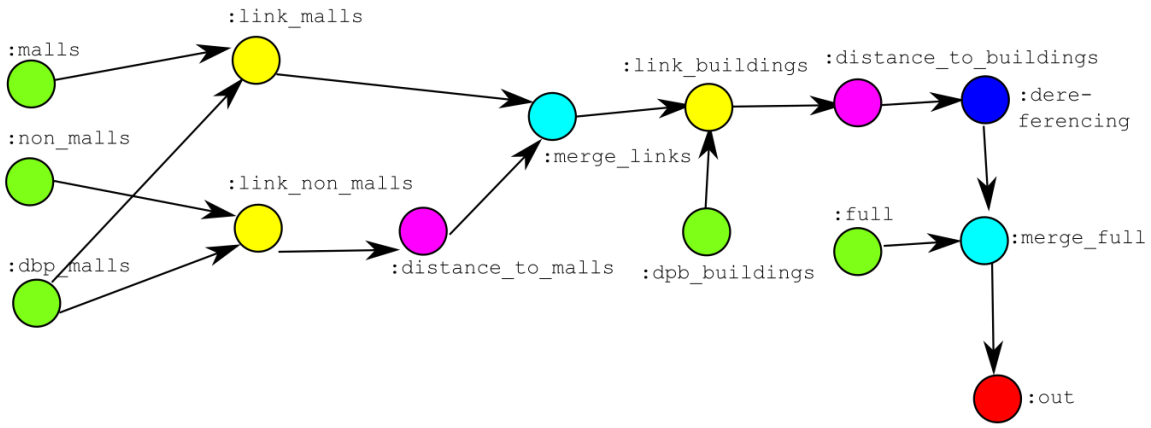


Figure 5: Configuration Graph for DINUC A.2

In the Annex, we provide the complete DEER configuration of our experiment, while the respective configuration graph is shown in Figure 5.

First, we had to split the input dataset into those POIs that represent malls or businesses within malls and those POIs that have no direct relationship with malls, because the following requirements for enrichment were based on this *"mall property"* of a POI. The splitting is taken care of by the filter enrichment operators `:malls` and `:non_malls`, that are both provided with the full input dataset and a SPARQL construct query that was specifically crafted for this task.

Parallel to the fetching of the input dataset and its splitting, we query *DBpedia* for data about malls and buildings in *Bucharest* using two model reader plugins (`:dbp_malls` and `:dbp_buildings`) that are configured with the endpoint URL of DBpedia as well as SPARQL construct queries that select just the data needed at this level from the DBpedia.

In a second step of enrichment, we generate links between the malls in our input dataset and the malls from DBpedia as well as between the "non-malls" in our input dataset and the malls from DBpedia. This is done using two linking enrichment operators configured with different LIMES link specifications: while mall-to-mall (same as) links are being discovered in the node `:link_malls` using only similarities on POI names, mall-to-non-mall (nearby) links are being discovered in the node `:link_non_malls` using a geospatial distance function on the geo coordinates of the POIs.

As a next enrichment step, we calculate the distance between the pairs of POIs linked with the nearby relationship from before using the geo-distance enrichment operator with the id `:distance_to_malls`.

Consecutively, we merge the results from the distance calculation with the results from the `:link_malls` node using the merge enrichment operator in `:merge_links`.

This is where the buildings POI data from DBpedia enters the scene: in another linking enrichment step (`:link_buildings`), we discover all nearby relationships between large buildings from DBpedia and all the POIs in our input dataset.

Then again, we utilize the geo-distance enrichment operator (`:distance_to_buildings`) to calculate distances for the nearby relationship of POIs with buildings discovered in the step before.

In the next and biggest enrichment step we use the dereferencing enrichment operator in order to add valuable information about nearby malls and buildings, as specified in DINUC A.2 to our POIs in `:dereferencing`.

The last nodes in our enrichment graph do not perform actual enrichment, but instead prepare and write the resulting output dataset to disk. By preparation, we mean that we first add all triples that were originally in the input dataset back to our intermediary dataset using the merge enrichment operator in `:merge_full`. These were left out before in order to speed up the intermediary enrichment process and reduce RAM usage. Finally, we use the default model writer to output our finalized, enriched dataset to disk using the Turtle serialization format.

4.3. Results

Our novel analytics feature of DEER allows to collect statistic information when executing a given configuration graph. The most important KPIs for DEER are its runtime and how much data are flowing through the graph. We present the analytics output of DEER after executing the configuration graph that has been discussed in the last section in Table 2.

Table 2: DEER Analytics from running the specification on both datasets. All runtimes are in milliseconds. When an enrichment operator has multiple inputs or outputs the respective sizes are separated by a semi-comma.

Plugin node identifier	Dataset name	Runtime (milliseconds)	Number of input triples	Number of output triples
`:dbp_buildings`	TomTom	1,808	-	{196}
	OSM	1,224	-	{196}
`:dbp_malls`	TomTom	1,808	-	{43}
	OSM	1,226	-	{43}
`:full`	TomTom	4,526	-	{186,739}
	OSM	5,420	-	{234,098}
`:non_malls`	TomTom	1,720	{186,739}	{25,305}
	OSM	2,613	{234,098}	{53,164}
`:link_non_malls`	TomTom	1,071	{25,305; 43}	{27,973}
	OSM	1,496	{53,164; 43}	{59,162}
`:distance to mal ls`	TomTom	112	{27,973}	{30,598}
	OSM	191	{59,162}	{65,117}
`:malls`	TomTom	1,107	{186,739}	{108}
	OSM	1,611	{234,098}	{88}

:link_malls	TomTom	195	{108; 43}	{178}
	OSM	214	{88; 43}	{153}
:merge_links	TomTom	88	{178; 30,598}	{30,682}
	OSM	192	{153; 65,117}	{65,227}
:link_buildings	TomTom	694	{30,682; 196}	{34,670}
	OSM	1,859	{65,227; 196}	{73,551}
:distance_to_buildings	TomTom	193	{34,670}	{38,371}
	OSM	152	{73,551}	{81,515}
:dereferencing	TomTom	5,367	{38,371}	{62,601}
	OSM	4,982	{81,515}	{133,452}
:merge_full	TomTom	475	{187,739; 62,601}	{237,542}
	OSM	680	{234,098; 133,452}	{340,924}
:out	TomTom	3,626	{237,542}	-
	OSM	5,058	{340,924}	-

As can be seen, the most time apart from I/O is spent on the dereferencing enrichment operator which was expected as it a) contributes the newest triples to the overall enrichment process, and b) has to issue a couple of hundred HTTP queries against the *DBpedia* server in order to obtain the information.

The total run times of the TomTom and OSM datasets are 19,018ms 24,020ms respectively, indicating a **linear scaling of DEER overall**, as the ratios between input sizes and overall run times are almost identical: 0.7976 on the input sizes and 0.7917 on the run times.

Overall, the end product of the enrichment process has a **mean share of enriched information of 37,5%** which highlights that DEER indeed does add a lot of information to the original datasets. However, it remains to be evaluated if the enrichment is distributed uniformly over all input POIs and if not, how large is the portion of POIs missing out on the enrichment process. We aim to implement more and better analytics targeted at this question in the future version of the DEER framework.

5. References

[1] F. Abel, Q. Gao, G.-J. Houben, and K. Tao. *Semantic enrichment of twitter posts for user profile construction on the social web.* In Proc. of ESWC, pages 375–389. Springer, 2011.

[2] C. Bizer and A. Schultz. The R2R Framework: *Publishing and Discovering Mappings on the Web.* In Proceedings of COLD, 2010.

[3] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. Inf. Process. Lett., 24(6):377–380, Apr. 1987.

[4] L. Buhmann and J. Lehmann. Pattern based knowledge base enrichment. In 12th International Semantic Web Conference, 21-25 October 2013, Sydney, Australia, 2013.

[5] S. Choudhury, J. G. Breslin, and A. Passant. Enrichment and ranking of the YouTube tag space and integration with the linked data cloud. Springer, 2009.

[6] S. Dietze, S.Sanchez-Alonso,H.Ebner,H.Q.Yu,D.Giordano,I.Marenzi,andB.P.Nunes. Interlinking educational resources and the web of data: A survey of challenges and approaches. Program: electronic library and information systems, 47(1):60–91, 2013.

[7] J. Euzenat and P. Shvaiko. Ontology matching. Springer-Verlag, Heidelberg (DE), 2007.

[8] S. Hasan, E.Curry, M. Banduk, and S. O'Riain. *Toward situation awareness for the semantic sensor web: Complex event processing with dynamic linked data enrichment.* Semantic Sensor Networks, page 60, 2011.

[9] H. H. Hoang, T. N.-P. Cung, D. K. Truong, D. Hwang, and J. J. Jung. *Semantic information integration with linked data mashups approaches.* International Journal of Distributed Sensor Networks, 2014, 2014.

[10] R. Isele and C. Bizer. *Learning Linkage Rules using Genetic Programming.* In Sixth International Ontology Matching Workshop, 2011.

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

[12] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. *DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia.* Semantic Web Journal, 2014.

[13] V. Lopez, C. Unger, P. Cimiano, and E. Motta. *Evaluating question answering over linked data.* Web Semantics: Science, Services and Agents on the World Wide Web, 21:3–13, 2013.

[14] I. Millard, H. Glaser, M. Salvadores, and N. Shadbolt. *Consuming multiple linked data sources: Challenges and experiences.* In COLD Workshop, 2010.

[15] A.-C. Ngonga Ngomo. *On link discovery using a hybrid approach.* Journal on Data Semantics, 1:203 – 217, December 2012.

[16] A.-C. Ngonga Ngomo, S. Auer, J. Lehmann, and A. Zaveri. *Introduction to linked data and its lifecycle on the web.* In Reasoning Web. Semantic Technologies for Intelligent Data Access, pages 1–90. Springer Berlin Heidelberg, 2014.

[17] A.-C. Ngonga Ngomo, N. Heino, K. Lyko, R. Speck, and M. Kaltenböck. *SCMS - semantifying content management systems.* In ISWC 2011, 2011.

[18] A.-C. Ngonga Ngomo and K. Lyko. *Unsupervised learning of link specifications: deterministic vs. non-deterministic.* In Proceedings of the Ontology Matching Workshop, 2013.

[19] A. Nikolov, V. Uren, E. Motta, and A. Roeck. *Overcoming schema heterogeneity between linked semantic repositories to improve coreference resolution*. In Proceedings of the 4th Asian Conference on The Semantic Web, pages 332–346, 2009.

[20] D. L. Phuoc, A. Polleres, M. Hauswirth, G. Tummarello and C. Morbidoni. *Rapid prototyping of semantic mash-ups through semantic web pipes*. In WWW, pages 581–590, 2009.

[21] A. Schultz, A. Matteini, R. Isele, C. Bizer, and C. Becker. *LDIF - linked data integration framework*. In COLD, 2011.

[22] R. Speck and A. N. Ngomo. *Ensemble learning for named entity recognition*. In Proc. of ISWC (International Semantic Web Conference) 2014, pages 519–534, 2014.

6. Annex

```
@prefix : <http://deer.aksw.org/vocabulary#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix geos: <http://www.opengis.net/ont/geosparql#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix dbp: <http://dbpedia.org/property/> .
@prefix slipo: <http://slipo.eu/def#> .

####
# Input Nodes
####

:fullInput
              :implementedIn     :DefaultModelReader ;
              :fromUri      "slipo-bucharest-input.ttl" ;
              :hasOutput         ( :malls :non_malls :mergeFull ) .

:malls
              :implementedIn     :FilterEnrichmentOperator ;
              :sparqlConstructQuery
                  '''PREFIX geos: <http://www.opengis.net/ont/geosparql#>
                     PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
                     PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
                     PREFIX slipo: <http://slipo.eu/def#>
                     CONSTRUCT {
                       ?s rdfs:label ?x .
                       ?s geo:lat ?lat .
                       ?s geo:long ?long .
                       ?s geo:geometry ?geometry .
                     } WHERE {
                        {
                           ?s geo:long ?long .
                           ?s geo:lat ?lat .
                           ?s geos:hasGeometry/geos:asWKT ?geometry .
                           ?s slipo:name/slipo:nameValue ?x .
                           FILTER regex(str(?x),'Mall')
                         } UNION {
                           ?s geo:long ?long .
                           ?s geo:lat ?lat .
                           ?s geos:hasGeometry/geos:asWKT ?geometry .
                           ?s slipo:name/slipo:nameValue ?x .
                           ?s slipo:category/slipo:value ?y.
                           FILTER regex(str(?y),'Shopping Center')
                         }
                   }''' ;
              :hasOutput         ( :linkMalls ) .

:non_malls
              :implementedIn     :FilterEnrichmentOperator ;
              :sparqlConstructQuery '''
              PREFIX geos: <http://www.opengis.net/ont/geosparql#>
              PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
```

```
               PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
               PREFIX slipo: <http://slipo.eu/def#>
               CONSTRUCT {
                  ?s rdfs:label ?x .
                  ?s geo:lat ?lat .
                  ?s geo:long ?long .
                  ?s geo:geometry ?geometry .
               } WHERE {
                  ?s geo:long ?long .
                  ?s geo:lat ?lat .
                  ?s geos:hasGeometry/geos:asWKT ?geometry .
                  ?s slipo:name/slipo:nameValue ?x .
                  FILTER regex(str(?x), '^((?!Mall).)*$') .
                  OPTIONAL { ?s slipo:category/slipo:value ?y .
                  FILTER regex(str(?y), '^((?!Shopping Center).)*$') .
               }
            } ''' ;
               :hasOutput         ( :linkNonMalls ) .

:dbp_malls
               :implementedIn     :DefaultModelReader ;
               :useEndpoint       <http://dbpedia.org/sparql> ;
               :useSparqlConstruct '''
               PREFIX geos: <http://www.opengis.net/ont/geosparql#>
               PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
               PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
               PREFIX slipo: <http://slipo.eu/def#>
               PREFIX dbo: <http://dbpedia.org/ontology/>
               PREFIX dbr: <http://dbpedia.org/resource/>
               PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
               construct {
                   ?mall rdfs:label ?label .
                   ?mall geo:lat ?lat .
                   ?mall geo:long ?long .
                   ?mall geo:geometry ?geometry .
               } where {
                   ?mall rdf:type dbo:ShoppingMall .
                   ?mall dbo:location dbr:Bucharest .
                   ?mall rdfs:label ?label .
                   ?mall geo:lat ?lat .
                   ?mall geo:long ?long .
                   ?mall geo:geometry ?geometry .
                } LIMIT 10000''' ;
               :hasOutput         ( :linkMalls :linkNonMalls ) .

:dbp_buildings
               :implementedIn     :DefaultModelReader ;
               :useEndpoint       <http://dbpedia.org/sparql> ;
               :useSparqlConstruct '''
               PREFIX geos: <http://www.opengis.net/ont/geosparql#>
               PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
               PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
               PREFIX slipo: <http://slipo.eu/def#>
               PREFIX dbo: <http://dbpedia.org/ontology/>
               PREFIX dbr: <http://dbpedia.org/resource/>
               PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
               construct {
                  ?building rdfs:label ?label .
                  ?building geo:lat ?lat .
                  ?building geo:long ?long .
```

```
                ?building geo:geometry ?geometry .
              } where {
                ?building rdf:type dbo:Building .
                ?building dbo:location dbr:Bucharest .
                ?building rdfs:label ?label .
                ?building geo:lat ?lat .
                ?building geo:long ?long .
                ?building geo:geometry ?geometry .
                FILTER NOT EXISTS { ?building rdf:type dbo:ShoppingMall . }
              } LIMIT 10000''' ;
             :hasOutput         ( :linkBuildings ) .

####
# Enrichment Nodes
####

:linkMalls
             :implementedIn     :LinkingEnrichmentOperator ;
             :hasInput          ( :malls :dbp_malls ) ;
             :hasOutput         ( :mergeLinks ) ;
             :selectMode        "best1toN" ;
             :linkingPredicate  owl:sameAs ;
             :linkSpecification "mongeelkan(y.http://www.w3.org/2000/01/rdf-
schema#label, x.http://www.w3.org/2000/01/rdf-schema#label)" ;
             :threshold 0.3 .
:linkNonMalls
             :implementedIn     :LinkingEnrichmentOperator ;
             :hasInput          ( :non_malls :dbp_malls ) ;
             :hasOutput         ( :distanceToMalls ) ;
             :selectMode        "best1toN" ;
             :linkingPredicate  slipo:nearestMall ;
             :linkSpecification
"geo_hausdorff(x.http://www.w3.org/2003/01/geo/wgs84_pos#geometry,y.http://www.w
3.org/2003/01/geo/wgs84_pos#geometry)" ;
             :threshold          0.333 .

:distanceToMalls
             :implementedIn     :GeoDistanceEnrichmentOperator ;
             :hasInput          ( :linkNonMalls ) ;
             :hasOutput         ( :mergeLinks ) ;
             :selectPredicate  slipo:nearestMall ;
             :distancePredicate slipo:distanceToNearestMall .

:mergeLinks
            :implementedIn       :MergeEnrichmentOperator ;
            :hasInput          ( :linkMalls :distanceToMalls ) ;
            :hasOutput         ( :linkBuildings ) .

:linkBuildings
             :implementedIn     :LinkingEnrichmentOperator ;
             :hasInput          ( :mergeLinks :dbp_buildings ) ;
             :hasOutput         ( :distanceToBuildings ) ;
             :selectMode        "best1toN" ;
             :linkingPredicate  slipo:nearestBuilding ;
             :linkSpecification
"geo_hausdorff(x.http://www.w3.org/2003/01/geo/wgs84_pos#geometry,y.http://www.w
3.org/2003/01/geo/wgs84_pos#geometry)" ;
             :threshold          0.333 .

:distanceToBuildings
```

```
              :implementedIn     :GeoDistanceEnrichmentOperator ;
              :hasInput          ( :linkBuildings ) ;
              :hasOutput         ( :dereferencing ) ;
              :selectPredicate  slipo:nearestBuilding ;
              :distancePredicate slipo:distanceToNearestBuilding .

:dereferencing
              :implementedIn     :DereferencingEnrichmentOperator ;
              :hasInput          ( :distanceToBuildings ) ;
              :hasOutput         ( :mergeFull ) ;
              :operations        (
                                  [ :lookUpProperty owl:sameAs ;
:dereferencingProperty dbo:openingDate ; :importProperty slipo:openingDate ]
                                  [ :lookUpProperty owl:sameAs ;
:dereferencingProperty dbp:numberOfStores ; :importProperty slipo:numberOfStores
]
                                  [ :lookUpProperty owl:sameAs ;
:dereferencingProperty dbp:parking ; :importProperty slipo:parkingSlots ]
                                  [ :lookUpProperty slipo:nearestMall ;
:dereferencingProperty dbo:openingDate ; :importProperty
slipo:nearestMallOpeneningDate ]
                                  [ :lookUpProperty slipo:nearestMall ;
:dereferencingProperty dbp:numberOfStores ; :importProperty
slipo:nearestMallNumberOfStores ]
                                  [ :lookUpProperty slipo:nearestMall ;
:dereferencingProperty dbp:parking ; :importProperty
slipo:nearestMallParkingSlots ]
                                  [ :lookUpProperty slipo:nearestMall ;
:dereferencingProperty rdfs:label ; :importProperty slipo:nearestMallName ]
                                  [ :lookUpProperty slipo:nearestBuilding ;
:dereferencingProperty dbo:openingDate ; :importProperty
slipo:nearestBuildingOpeningDate ]
                                  [ :lookUpProperty slipo:nearestBuilding ;
:dereferencingProperty dbp:status ; :importProperty slipo:nearestBuildingStatus
]
                                  [ :lookUpProperty slipo:nearestBuilding ;
:dereferencingProperty dbo:floorArea ; :importProperty
slipo:nearestBuildingFloorArea ]
                                  [ :lookUpProperty slipo:nearestBuilding ;
:dereferencingProperty dbo:floorCount ; :importProperty
slipo:nearestBuildingFloorCount ]
                                  [ :lookUpProperty slipo:nearestBuilding ;
:dereferencingProperty rdfs:label ; :importProperty slipo:nearestBuildingName ]
                                 ) .

:mergeFull
            :implementedIn       :MergeEnrichmentOperator ;
            :hasInput          ( :fullInput :dereferencing ) ;
            :hasOutput         ( :output_node ) .


####
# Output Nodes
####

:output_node
              :implementedIn     :DefaultModelWriter ;
              :outputFile        "slipo-bucharest-enriched.ttl" ;
              :outputFormat      "Turtle" .
```