



GEOLIFT - Spatial mapping framework for enriching RDF datasets with Geo-spatial information.

Abstract:

This manual presents the spatial mapping component dubbed GEOLIFT. The goal of GEOLIFT is to enrich RDF datasets with geo-spatial information. To achieve this goal, GeoLit relies on three atomic modules based on dereferencing, linking and NLP. GEOLIFT was implemented in Java, is open-source and can be accessed at <https://github.com/GeoKnow/GeoLift/>.

Contents

1	Introduction	4
2	Assumptions	4
3	Technical Approach	5
3.1	Architecture	5
3.2	Using Dereferencing	6
3.3	Using Linking	7
3.4	Using Named Entity Recognition	9
4	Developers' Manual	10
4.1	Dereferencing Module	11
4.1.1	Input	11
4.1.2	Output	12
4.1.3	Process	12
4.1.4	Code Sample	12
4.2	Linking Module	13
4.2.1	Input	13
4.2.2	Output	13
4.2.3	Process	14
4.2.4	Code Sample	14
4.3	NLP Module	15
4.3.1	Input	15
4.3.2	Output	15
4.3.3	Process	15
4.3.4	Code Sample	16
4.4	Conformation Module	16
4.5	Filter Module	16
5	Running GeoLift From Command-Line	16
5.1	How GEOLIFT execute the RDF configuration file?	18
5.2	RDF Configuration main nodes	18
5.2.1	Dataset Resource	18
5.2.2	Module Resource	19
5.2.3	Operator Resource	19
5.2.4	Parameter Resource	19
6	Conclusions	22

1 Introduction

Manifold RDF data contain implicit references to geographic data. For example, music datasets such as *Jamendo* include references to locations of record labels, places where artists were born or have been, etc. The aim of the spatial mapping component, dubbed GEOLIFT, is to retrieve this information and make it explicit. In the following, we begin by presenting the basic assumptions that influence the development of the first component of GEOLIFT. Then, we present the technical approach behind GEOLIFT. Finally, we present the detailed developers' manual of GEOLIFT.

2 Assumptions

Geographical information can be mentioned in three different ways within Linked Data:

1. *Through dereferencing*: Several datasets contain links to datasets with explicit geographical information such as *DBpedia* or *LinkedGeoData*. For example, in a music dataset, one might find information such as `http://example.org/Leipzig`
`owl:sameAs`
`http://dbpedia.org/resource/Leipzig`.

We call this type of reference *explicit*. We can now use the semantics of RDF to fetch geographical information from DBpedia and attach it to the resource in the other ontology as `http://example.org/Leipzig` and `http://dbpedia.org/resource/Leipzig` refer to the same real-world object.

2. *Through linking*: It is known that the Web of Data contains an insufficient number of links. The latest approximations suggest that the Linked Open Data Cloud alone consists of 31+ billion triples but only contains approximately 0.5 billion links (i.e., less than 2% of the triples are links between knowledge bases). The second intuition behind our approach is thus to use link discovery to map resources in an input knowledge base to resources in a knowledge that contains explicit geographical information. For example, given a resource `http://example.org/Athen`, GEOLIFT should aim to find a resource such as `http://dbpedia.org/resource/Athen` to map it with. Once having established the link between the two resources, GEOLIFT can then resolve to the approach defined above.

3. *Through Natural Language Processing*: In some cases, the geographic information is hidden in the objects of data type properties. For example, some datasets contain biographies, textual abstracts describing resources, comments from users, etc. The idea here is to use this information by extracting Named Entities and keywords using automated Information Extraction techniques. Semantic Web Frameworks such as FOX¹ have the main advantage of providing URIs for the keywords and entities that they detect. These URIs can finally be linked with the resources to which the datatype properties were attached. Finally, the geographical information can be dereferenced and attached to the resources whose datatype properties were analyzed.

The idea behind GEOLIFT is to provide a generic architecture that contains means to exploit these three characteristics of Linked Data. In the following, we present the technical approach underlying GEOLIFT.

3 Technical Approach

3.1 Architecture

GEOLIFT was designed to be a modular tool which can be easily extended and re-purposed. In its first version, it provides two main types of artifacts:

1. *Modules*: These artifacts are in charge of generating geographical data based on RDF data. To this aim, they implement the three intuitions presented above. The input for such a module is an RDF dataset (in Java, a *Jena Model*). The output is also an RDF dataset enriched with geographical information (in Java, an enriched *Jena Model*). Formally, a module can thus be regarded as a function $\mu : \mathcal{R} \rightarrow \mathcal{R}$, where \mathcal{R} is the set of all RDF datasets.
2. *Operators*: The idea behind operators is to enable users to define a workflow for processing their input dataset. Thus, in case a user knows the type of enrichment that is to be carried out (using linking and then links for example), he can define the sequence of modules that must be used to process his dataset. Note that the format of the input and output of modules is identical. Thus, the user is empowered to create workflows of arbitrary complexity by simply connecting modules. Formally, an operator can be regarded as a function $\varphi : \mathcal{R} \cup \mathcal{R}^2 \rightarrow \mathcal{R} \cup \mathcal{R}^2$.

¹<http://fox.aksw.org>

The corresponding architecture is shown in Figure 1. The input layer allows reading RDF in different serializations. The enrichment modules are in the second layer and allow adding geographical information to RDF datasets by different means. The operators (which will be implemented in the future version of GEOLIFT) will combine the enrichment modules and allow defining a workflow for processing information. The output layer serializes the results in different format. The enrichment procedure will be monitored by implementing a controller, which will be added in the future version of GEOLIFT.

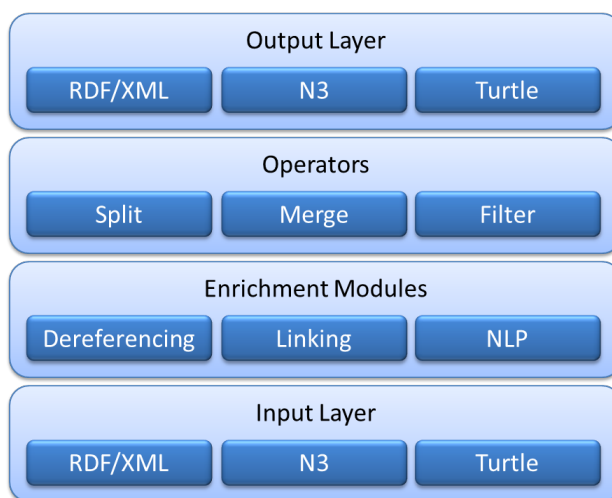


Figure 1: Architecture of GEOLIFT

In the following, we present the implementation of the three intuitions presented above in GEOLIFT.

3.2 Using Dereferencing

For datasets which contain `owl:sameAs` links, we deference all links from the dataset to other datasets by using a content negotiation on HTTP as shown in Figure 2. This returns a set of triples that needs to be filtered for relevant geographical information. Here, we use a predefined list of attributes that links to geographical information. 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 2: Content Negotiation as used by GEOLIFT (courtesy of W3C)

3.3 Using Linking

As pointed out before, links to geographical resources do not occur in several knowledge bases. Here, we rely on the metrics implemented in the LIMES framework² [5, 4, 6] to link the resources in the input dataset with geographical datasets. LIMES, the **Link** Discovery Framework for **Metric** Spaces, is a framework for discovering links between entities contained in Linked Data sources. LIMES is a hybrid framework [4] 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. The architecture of LIMES is shown in Figure 3

Linking using LIMES [4, 3] can be achieved in three ways:

1. *Manually*, by the means of a link specification [4], which is an XML-description of (1) the resource in the input and target datasets that are to be linked and (2) of the similarity measure that is to be employed to link these datasets.
2. *Semi-automatically* based on active learning [7, 8, 9]. Here, the idea is that if the user is not an expert and thus unable to create a link specification, he 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

²<http://\limes.sf.net>

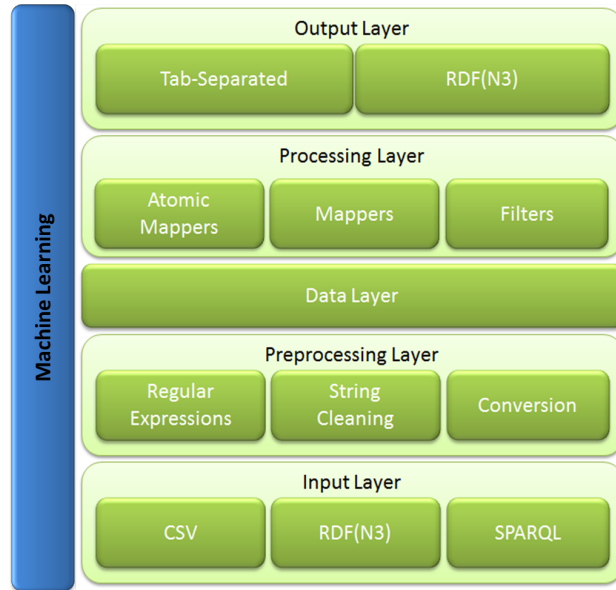


Figure 3: Architecture of LIMES

machine-learning approaches that optimize a pseudo-F-measure to create a one-to-one mapping.

The techniques implemented by LIMES can be accessed via the SAIM user interface³, of which a screenshot is shown in Figure 4.

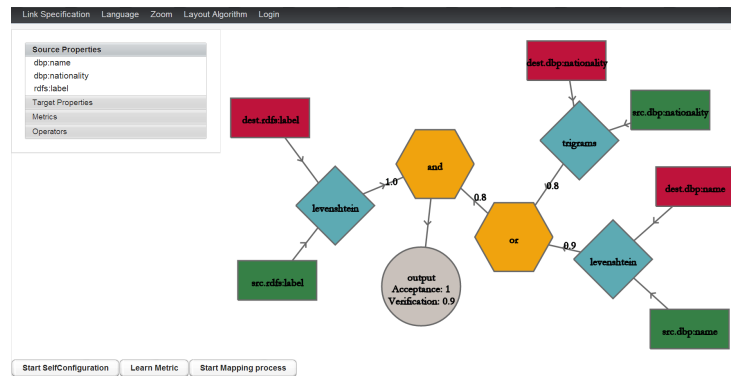


Figure 4: Screenshot of SAIM

³<http://saim.aksw.org>

3.4 Using Named Entity Recognition

The geographical information hidden in datatype properties is retrieved by using Named Entity Recognition. In the first version of GEOLIFT, we rely on the FOX framework. The FOX framework is a stateless and extensible framework that encompasses keyword extraction and named entity recognition. Its architecture consists of three layers as shown in Figure 5.

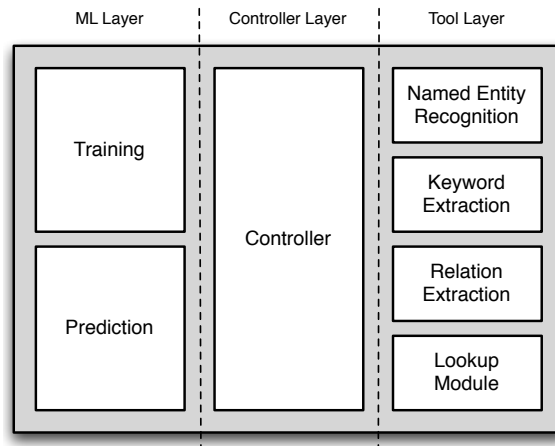


Figure 5: Architecture of the FOX framework.

FOX takes text or HTML as input. Here we use the objects of datatype properties, i.e., plain text. This data is sent to the *controller layer*, which implements the functionality necessary to clean the data, i.e., remove HTML and XML tags as well as further noise. Once the data has been cleaned, the controller layer begins with the orchestration of the tools in the *tool layer*. Each of the tools is assigned a thread from a thread pool, so as to maximize usage of multi-core CPUs. Every thread runs its tool and generates an event once it has completed its computation. In the event that a tool does not complete after a set time, the corresponding thread is terminated. So far, FOX integrates tools for KE, NER and RE. The KE is realized by tools such as KEA⁴ and the Yahoo Term Extraction service⁵. In addition, FOX integrates the Stanford Named Entity Recognizer⁶ [2], the Illinois Named Entity Tagger⁷ [10] and Alchemy⁸ for NER.

The results from the tool layer are forwarded to the *prediction module* of

⁴<http://www.nzdl.org/Kea/>

⁵<http://developer.yahoo.com/search/content/V1/termExtraction.html>

⁶<http://nlp.stanford.edu/software/CRF-NER.shtml>

⁷http://cogcomp.cs.illinois.edu/page/software_view/4

⁸<http://www.alchemyapi.com>

the *machine-learning layer*. The role of the prediction module is to generate FOX's output based on the output the tools in FOX's backend. For this purpose, it implements several ensemble learning techniques [1] with which it can combine the output of several tools. Currently, the prediction module carries out this combination by using a feed-forward neural network. The neural network inserted in FOX was trained by using 117 news articles. It reached 89.21% F-Score in an evaluation based on a ten-fold-cross-validation on NER, therewith outperforming even commercial systems such as Alchemy.

Once the neural network has combined the output of the tool and generated a better prediction of the named entities, the output of FOX is generated by using the vocabularies shown in Figure 6. These vocabularies extend the two broadly used vocabularies Annotea⁹ and Autotag¹⁰. In particular, we added the constructs explicated in the following:

- `scms:beginIndex` denotes the index in a literal value string at which a particular annotation or keyphrase begins;
- `scms:endIndex` stands for the index in a literal value string at which a particular annotation or keyphrase ends;
- `scms:means` marks the URI assigned to a named entity identified for an annotation;
- `scms:source` denotes the provenance of the annotation, i.e., the URI of the tool which computed the annotation or even the system ID of the person who curated or created the annotation and
- `scmsann` is the namespace for the annotation classes, i.e, location, person, organization and miscellaneous.

4 Developers' Manual

GEOLIFT contains three basic *Java* packages:

- `IO package` which deals with input/output operations using the *Reader* and *Writer* classes.
- `Operators package` will contains implementation of different operators like *merge*, *split*, *filter*, ...

NOTE: Operators package will be implemented in future version of GEOLIFT .

⁹<http://www.w3.org/2000/10/annotation-ns#>

¹⁰<http://commontag.org/ns#>

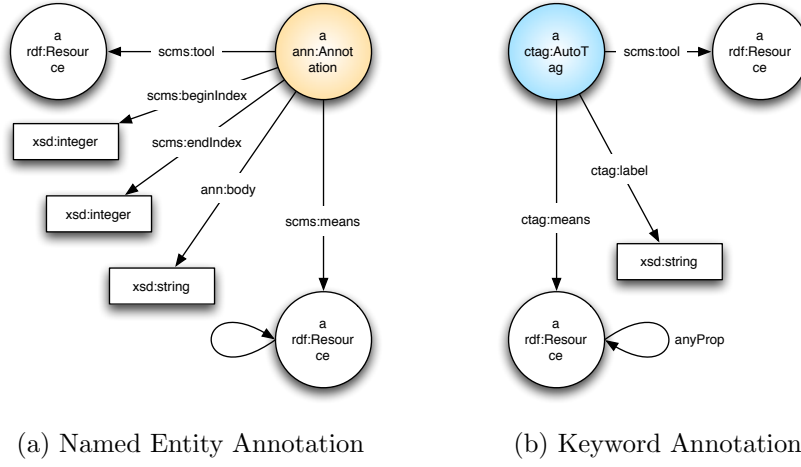


Figure 6: Vocabularies used by FOX for representing named entities (a) and keywords (b)

- **Modules package** contains the **GeoLiftModule** interface which is implemented by the basic classes: **URIDereferencing** class which handles dereferencing geographical information extending process. **Linking** class which handles linking geographical information extending processes. and **nlpEnrichGeoTriples** class which handles named entity extraction process.

All modules implement the **GeoLiftModule** interface's two methods **getParameters()** and **process()**, where the **getParameters()** method returns a list of its module input parameters, and the **process()** method takes as input a *Jena* model and a Map of different parameters in form of ("parameterName", "parameterValue"), and as output it generates a Jena model also, this organization ease the usage of the modules in different workflows.

Moreover, a *Java* jar file is generated for each module, which can be executed directly from command line along with each module parameters.

4.1 Dereferencing Module

The purpose of the dereferencing module is to extend the model's Geo-spatial information by set of information through specified predicates.

4.1.1 Input

- **Data model** contains the triples of the dataset to be enriched (This data model can be an output from previous stage or can be loaded

from file/URI directly before using the module).

- **Predicates list** list of interested predicates to be added as enrichment to the data model. Table 1 provides details about each of the `URIDereferencing` module’s parameters.

Table 1: Dereferencing parameters description

Parameter Name	Default	Description
<code>inputProperty<n></code>	<code>null</code>	List of interesting predicates to enrich the model, and their Objects’ values. e.g. <code><“inputProperty1”, “http://www.w3.org/2003/01/geo/wgs84_pos#lat”></code> . Mandatory parameter
<code>outputProperty<n></code>	<code>null</code>	The enriched output property.
<code>useBlankNodes</code>	<code>false</code>	Use blank node in output dataset.

4.1.2 Output

- **Data model** enriched data model with additional Geo-spatial information added by the given input predicates with its extracted object values.

4.1.3 Process

In this module, a *Java Jena* model and a list of interested predicates are given as inputs. This is done by iterating over the model’s resources (dubbed as original resources) and for each of the original resource an extraction of the predicates’ values (objects) that are in the form of URI is performed. These URIs (dubbed as dereferenced resources) are more filtered to be the resources used in *DBpedia*. The dereferenced resources are handled by a dereference operation in order to find the interested predicates list for them. Such predicates and their objects’ values are fetched and added to the the original resource to extend its information.

4.1.4 Code Sample

Listing 1 provides a sample code showing how to use the `URIDereference` module:

Listing 1: Code fragment to call the URIDereferencing class.

```
1 // Define URIDereferencing object
2 URIDereferencing u = new URIDereferencing();
3 // Define parameters Map
4 Map<String, String> parameters = new HashMap<String, String>();
5 // Set parameters
6 parameters.put("predicate1", predicate1Value);
7 parameters.put("predicate2", predicate2Value);
8 // read input Model
9 Model model=org.aksw.geolift.io.Reader.readModel(datasetSource);
10 // Enrich the Model
11 Model resultedModel = u.process(model, parameters);
12 // Use the enriched model
13 resultedModel.write(System.out,"TTL");
```

4.2 Linking Module

4.2.1 Input

- **Data model** contains the triples of the dataset to be enriched (This data model can be an output from previous stage or can be loaded from file/URI directly before using the module).
- **Parameters list** that will be used during the process. These parameters include:

Specification file path, the path to the `spec.xml` file contains the linking specifications

URI position, represents the original model's URI position as source or target in the linking specifications.

The parameters, other than the data model parameter, are collected in Map structure form, where each entry's value in the Map represents the parameter itself. Table 2 provides details about the Linking module's parameters .

4.2.2 Output

- **Data model** enriched with additional geographic information URIs represented in `owl:sameAs` predicates.

Table 2: Linking parameters description

Parameter Name	Default	Description
<code>specFilePath</code>	N/A	The path to specification file used for linking process, the original dataset to be enriched must be on the source dataset , e.g. <code>linkingModuleData/linking/spec.xml</code> .
<code>linksFilePath</code>	N/A	The path to links file resulted from the linking process. This file's path is the same as the one specified in LIME's specifications file as output file, e.g. <code>linkingModuleData/linking/links.nt</code> .
<code>linksPart</code>	N/A	Represents the original model's URI position as source/left or target/right in the linking specifications. Its value is either 'source' or 'target'.

4.2.3 Process

In this module an input model is given and list of parameters for used files during the process. The process starts by generating links between the dataset model and another dataset as second partner. This is done using LIMEs link discovery framework by specifying the linking specification file given as parameter. The links are generated in `accept.nt` file that is used after to combine such links with their original resources in the source dataset model as `owl:samAs` predicates objects. The result is a dataset model enriched with geographical information links. Another forward step is to feed the Dereference module with the resulted enriched model from Linking module as input. The previously generated links in the model in addition to other objects in the URIs form will be dereferenced adding more and detailed geographical information.

4.2.4 Code Sample

Listing 2 provides a sample code showing how to use the Linking module.

Listing 2: Code fragment to call the `Linking` class.

```
1 // Define Linking object
```

```

2 Linking l = new Linking();
3 // Define parameters Map
4 Map<String, String> parameters = new HashMap<String, String>();
5 // Set parameters
6 parameters.put("datasetSource",datasetSourceValue);
7 parameters.put("specFilePath",specFilePathValue);
8 parameters.put("linksFilePath",linksPathValue);
9 parameters.put("linksPart",linksPartValue);
10 // read input Model
11 Model model =
    org.aksw.geolift.io.Reader.readModel(parameters.get("datasetSource"));
12 // Enrich the Model
13 model = l.process(model, parameters);
14 // Use the enriched model
15 model.write(System.out,"TTL");

```

4.3 NLP Module

4.3.1 Input

- **Data model** contains the dataset to be enriched (This data model can be an output from previous module or can be loaded from file/URI directly before using the module).
- **Parameters list** that will be used during the process. The `getParameters()` method of the `NlpGeoEnricher` class returns a list of parameters, which can be set by the user to provide custom control of the *Named entity extraction* provided by the implemented *FOX framework*.

4.3.2 Output

- **Data model** enriched with additional Geo-spatial information URIs represented by the `addedGeoProperty` predicates, witch by default is `gn:relatedTo`¹¹ predicates.

4.3.3 Process

The `process()` method of the `NlpGeoEnricher` class takes as input a *Jena* model and a Map of different parameters, and as output it generates a *Jena* model also. Table 3 provides details about the `NlpGeoEnricher` module's parameters .

¹¹Prefix gn stands for <http://geoknow.org/ontology/>

4.3.4 Code Sample

A sample code showing how to call the `NlpGeoEnricher` class and giving it its parameters is introduced in Listing 3.

Listing 3: Code fragment to call the `NlpGeoEnricher` class.

```
1 // Define NlpGeoEnricher object
2 NlpGeoEnricher geoEnricher= new NlpGeoEnricher();
3 // Define parameters Map
4 Map<String, String> parameters = new HashMap<String, String>();
5 // Set parameters
6 parameters.put("useFoxLight", "OFF");
7 parameters.put("askEndPoint", "false");
8 // read input Model
9 Model inputModel =
    org.aksw.geolift.io.Reader.readModel(inputFileValue);
10 // Enrich the Model
11 Model enrichedModel = geoEnricher.process(inputModel, parameters);
12 // Use the enriched model
13 enrichedModel.write(System.out, "TTL");
```

4.4 Conformation Module

Change a specified source URI to a specified target URI

4.5 Filter Module

Runs a set of triples patterns' against an input model to filter some triples and export them to an output model.

5 Running GeoLift From Command-Line

The GEOLIFT can be directly executed from command-line using the provided `geolift.jar`¹² file. Simply, provide the RDF configuration file as the only one parameter for the GEOLIFT jar file.

Example: `geolift.jar src/main/resources/workflow/config.ttl`

Listing 4 introduces an example of RDF configuration file, to make it easier to be understood we represent the configuration file in form of graph in figure 7.

¹²the `geolift.jar` file is in the `jars` folder of the project

Table 3: NLP parameters description

Parameter Name	Default	Description
input	N/A	The input file/URI to be enriched
output	N/A	The output file to write the enriched model to it
literalProperty	Top Ranked	Literal property used by FOX for NER, if not set the top ranked literal property is pecked by <code>LiteralPropertyRanker</code> module, 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.
addedGeoProperty	gn:relatedTo ¹¹	property added to the input model with additional Geo-spatial knowledge through NLP.
useFoxLight	OFF	An implemented NER class name (<code>org.aksw.fox.nertools.NEROpenNLP</code> , <code>org.aksw.fox.nertools.NERIllinoisExtended</code> , <code>org.aksw.fox.nertools.NERBalie</code> , <code>org.aksw.fox.nertools.NERStanford</code>) or OFF, If this parameter is given with a wrong value, <code>org.aksw.fox.nertools.NERStanford</code> will be used.
askEndPoint	false	Ask the <i>DBpedia</i> endpoint for each location returned by FOX (setting it generates slower execution time but more accurate results)
foxType	TEXT	text or an url (e.g.: ‘G. W. Leibniz was born in Leipzig’, ‘ http://en.wikipedia.org/wiki/Leipzig_University ’)
foxTask	NER	FOX task :{NER} for Named Entity Recognition
foxInput	""	FOX actual input as text or an URL
foxOutput	TURTLE	FOX output format: { JSON-LD N-Triples RDF/{ JSON XML } }
foxUseNif	false	FOX generates NIF: { true false }
foxReturnHtml	false	FOX returns HTML: { true false }

5.1 How GeoLift execute the RDF configuration file?

First of all GEOLIFT RDF configuration reader determines the set of output datasets D , i.e. datasets which are included as output of some modules/operators but not as input to any other modules/operators. In our example it will be only d_8 .

For each dataset in D GEOLIFT tries to find it either trivially from file, URI or endpoint if provided by the dataset node in the RDF configuration file, or recursively by solving for modules/operators that generate it as output.

Going back to our example, GEOLIFT first tries to read d_8 trivially but it fails as there are no direct way to get it. Then, GEOLIFT recursively goes for solving the **Conformation** module as it generates d_8 as output. This recursive procedure continues until it reads the dataset d_1 from the endpoint, then going back in the recursion stack.

5.2 RDF Configuration main nodes

GEOLIFT RDF configuration contains four main resource types (dataset, module, operator and parameter nodes).

5.2.1 Dataset Resource

As its name implies, a Dataset resource represents a dataset. For an example see figure 7 (lines 4:6), in which it represents a dataset with URI `http://dbpedia.org/resource/Berlin` and endpoint `http://dbpedia.org/sparql`. GEOLIFT will generate the Concise Bounded Description (CBD)¹³ of the *Berlin* resource from *DBpedia* endpoint.

Beside defining a resource as a dataset (ex: figure 7 (lines 4)), a dataset resource may also contain the following predicates:

- `rdfs:label` specifying label for the dataset.
- `:hasUri` setting URI of the dataset resource, to read resource through content negotiation (ex: figure 7 (lines 5)).
- `:FromEndPoint` end point to read the CBD from. *Note: must be used in conjunction with `:hasUri` predicate, otherwise an error will be generated by GEOLIFT.* (ex: figure 7 (lines 6))
- `:inputFile` input file to load the dataset from.

¹³For more details about CBD see <http://www.w3.org/Submission/CBD/>

- `:outputFile` output file to save the dataset to (ex: figure 7(lines 14).
- `:outputFormat` output format to save the dataset in it (ex: figure 7(lines 14), Turtle format will be used as default format if this property is not set.

5.2.2 Module Resource

Beside defining a resource as a Module (ex: figure 7(lines 16), a Module resource may also contains the following predicates:

- `rdfs:label` pacifying label for the module (ex: figure 7(lines 17).
- `:hasInput` pacifying input datasets to the module (ex: figure 7(lines 18).
- `:hasOutput` pacifying output datasets to the module (ex: figure 7(lines 19).
- `:hasParameter` pacifying Parameters for the module (ex: figure 7(lines 20).

5.2.3 Operator Resource

Beside defining a resource as an operator (ex: figure 7(lines 24), a Module resource may also contains the following predicates:

- `rdfs:label` pacifying label for the operator (ex: figure 7(lines 25).
- `:hasInput` pacifying input datasets to the operator (ex: figure 7(lines 26).
- `:hasOutput` pacifying output datasets to the operator (ex: figure 7(lines 27).
- `:hasParameter` pacifying Parameters for the operator.

5.2.4 Parameter Resource

Beside defining a resource as a parameter (ex: figure 7(lines 21), a Module resource may also contains the following predicates:

- `rdfs:label` pacifying label for the parameter.
- `:hasKey` pacifying the parameter key (ex: figure 7(lines 22).

- `:HasValue` pacifying the parameter value (ex: figure 7(lines 23)).

The values of (key, values) pairs are modules/operators dependent. see modules/operators sections for more details.

Listing 4: Example of RDF configuration file.

```

1 @prefix : <http://geoknow.org/specsontology/> .
2 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3 @prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
4 :d1          a          :Dataset ;
5              :hasUri     <http://dbpedia.org/resource/Berlin> ;
6              :FromEndPoint <http://dbpedia.org/sparql> .
7 :d2          a          :Dataset .
8 :d3          a          :Dataset .
9 :d4          a          :Dataset .
10 :d5         a          :Dataset .
11 :d6         a          :Dataset .
12 :d7         a          :Dataset .
13 :d8         a          :Dataset ;
14             :outputFile  "GeoLiftBerlin.ttl" ;
15             :outputFormat "Turtle" .
16 :deref       a          :Module, :DereferencengModule ;
17             rdfs:label   "Dereferenceng module" ;
18             :hasInput    :d1 ;
19             :hasOutput    :d2 ;
20             :hasParameter :derefParam1 .
21 :derefParam1 a          :ModuleParameter, :DereferencengModuleParameter ;
22             :hasKey      "inputProperty1" ;
23             :hasValue    geo:lat .
24 :split       a          :Operator, :SplitOperator ;
25             rdfs:label   "Split operator" ;
26             :hasInput    :d2 ;
27             :hasOutput    :d3, :d4 .
28 :nlp         a          :Module, :NLPModule ;
29             rdfs:label   "NLP module" ;
30             :hasInput    :d3 ;
31             :hasOutput    :d5 ;
32             :hasParameter :nlpPram1, :nlpPram2 .
33 :nlpPram1    a          :ModuleParameter, :NLPModuleParameter ;
34             :hasKey      "useFoxLight" ;
35             :hasValue    "OFF" .
36 :nlpPram2    a          :ModuleParameter, :NLPModuleParameter ;
37             :hasKey      "askEndPoint" ;
38             :hasValue    false .
39 :filter      a          :Module, :FilterModule ;
40             rdfs:label   "Filter module" ;
41             :hasInput    :d4 ;
42             :hasOutput    :d6 ;

```

```

43             :hasParameter :FilterPram1 .
44 :filterPram1 a      :ModuleParameter, :NLPModuleParameter ;
45             :hasKey    "triplesPattern" ;
46             :hasValue  "?s <http://dbpedia.org/ontology/abstract> ?o".
47 :merge        a      :Operator, :MergeOperator ;
48             rdfs:label  "Merge operator" ;
49             :hasInput   :d6, :d5 ;
50             :hasOutput  :d7 .
51 :conform       a      :Module, :ConformationModule ;
52             rdfs:label  "Conformation module" ;
53             :hasInput   :d7 ;
54             :hasOutput  :d8 ;
55             :hasParameter :conformPram1, :conformPram2 .
56 :conformPram1 a      :ModuleParameter, :NLPModuleParameter ;
57             :hasKey     "sourceURI" ;
58             :hasValue   "http://dbpedia.org" .
59 :conformPram2 a      :ModuleParameter, :NLPModuleParameter ;
60             :hasKey     "targetURI" ;
61             :hasValue   "http://geolift.org" .

```

Details of RDF configuration file will be here soon

6 Conclusions

In this manual, we presented the GEOLIFT component for enriching RDF datasets with geo-spatial data. In future work, we aim to implement a graphical user interface on top of GEOLIFT to enable users to specify their workflows graphically. Moreover, we aim to implement workflow checking functionality.

References

- [1] Thomas G. Dietterich. Ensemble methods in machine learning. In *MCS*, pages 1–15, London, UK, 2000. Springer-Verlag.
- [2] J. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *ACL*, pages 363–370, 2005.
- [3] Axel-Cyrille Ngonga Ngomo. Link discovery with guaranteed reduction ratio in affine spaces with minkowski measures. In *Proceedings of ISWC*, 2012.

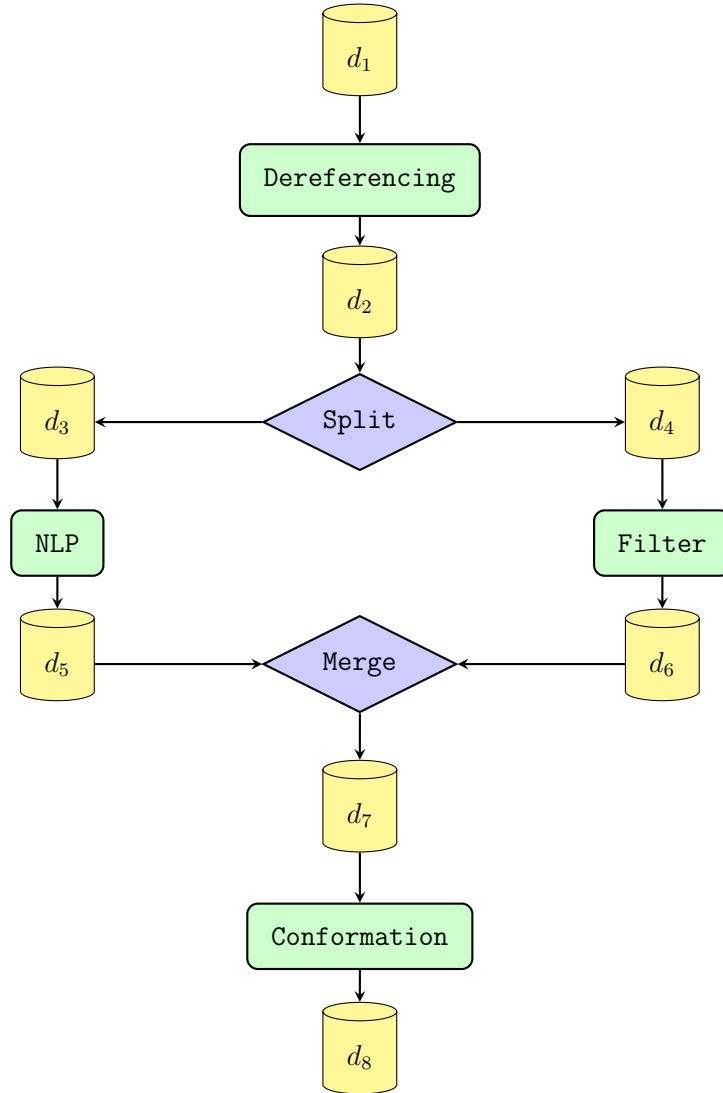


Figure 7: Graph representation of configuration file presented by listing 4

- [4] Axel-Cyrille Ngonga Ngomo. On link discovery using a hybrid approach. *Journal on Data Semantics*, 1:203 – 217, December 2012.
- [5] Axel-Cyrille Ngonga Ngomo and Sren Auer. A time-efficient approach for large-scale link discovery on the web of data. In *IJCAI*, page 2011, 2011.
- [6] Axel-Cyrille Ngonga Ngomo, Lars Kolb, Norman Heino, Michael Hartung, Sören Auer, and Erhard Rahm. When to reach for the cloud:

- Using parallel hardware for link discovery. In *Proceedings of ESCW*, 2013.
- [7] Axel-Cyrille Ngonga Ngomo, Jens Lehmann, Sören Auer, and Konrad Höffner. Raven – active learning of link specifications. In *Proceedings of OM@ISWC*, 2011.
- [8] Axel-Cyrille Ngonga Ngomo and Klaus Lyko. Eagle: Efficient active learning of link specifications using genetic programming. In *Proceedings of ESWC*, 2012.
- [9] Axel-Cyrille Ngonga Ngomo, Klaus Lyko, and Victor Christen. Coala – correlation-aware active learning of link specifications. In *Proceedings of ESWC*, 2013.
- [10] Lev Ratinov and Dan Roth. Design challenges and misconceptions in named entity recognition. In *CONLL*, pages 147–155, 2009.