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

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 FOX[1] 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.

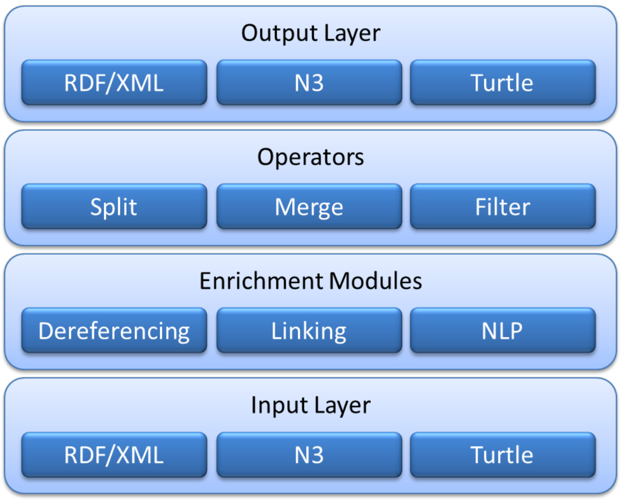


Figure 1: Architecture of GeoLift

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.

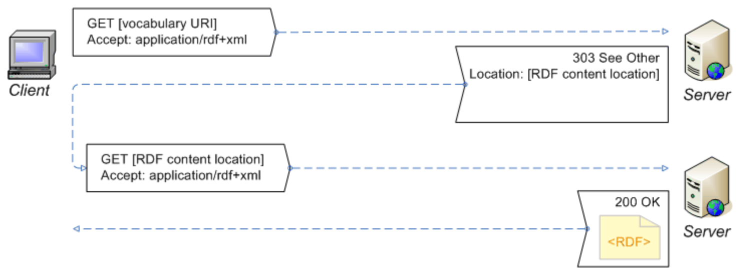

GET [vocabulary URI]
Accept: application/rdf+xml
Client
303 See Other
Location: [RDF content location]
Server
GET [RDF content location]
Accept: application/rdf+xml
200 OK
<RDF>
Server

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 framework[2] [5, 4, 6] to link the resources in the input dataset with geographical datasets. Limes, the **Li**nk Discovery Framework for **Me**tric **S**paces, is a framework for discovering links between entities contained in Linked Data sources. Limes is a hybrid framework [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 loosing 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 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`

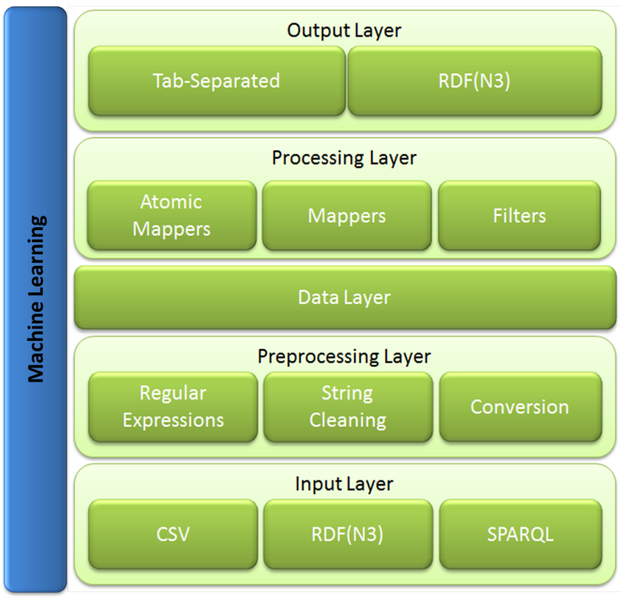


Figure 3: Architecture of LIMES

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 interface[3], of which a screenshot is shown in Figure 4.

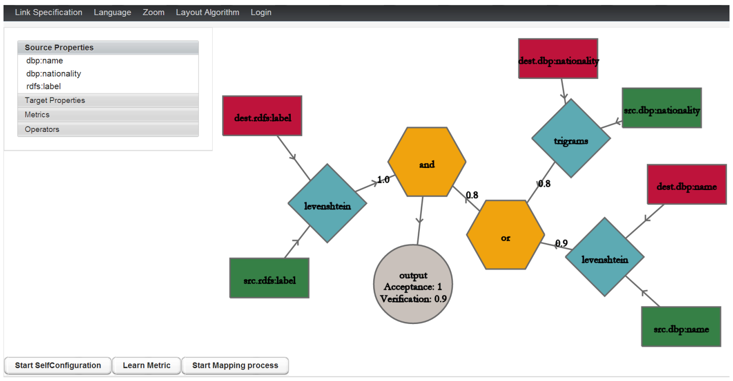


Figure 4: Screenshot of SAIM

[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.

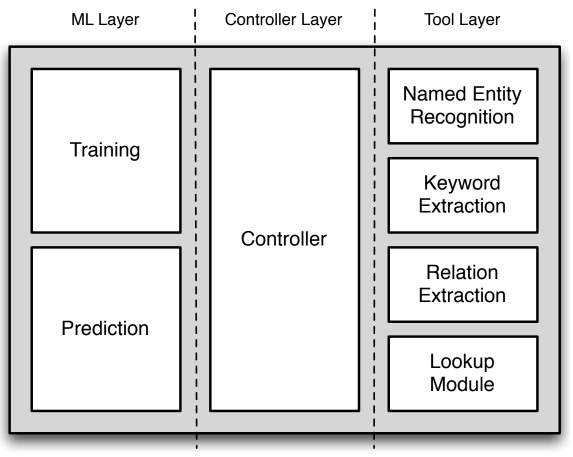


Figure 5: Architecture of the FOX framework.

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 KEA[4] and the Yahoo Term Extraction service[5]. In addition, FOX integrates the Stanford Named Entity Recognizer[6] [2], the Illinois Named Entity Tagger[7] [10] and Alchemy[8] 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`

[7]`http://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 Annotea[9] and Autotag [10]. In particular, we added the constructs explicated in the following:

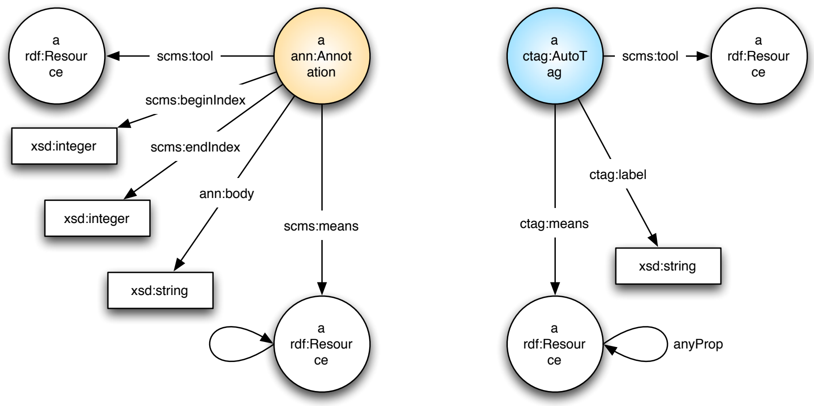
- `scms:beginIndex` denotes the index in a literal value string at which a particular annotation or keyphrase begins;
- `scms:endIndex` stands for the index in a literal value string at which a particular annotation or keyphrase ends;
- `scms:means` marks the URI assigned to a named entity identified 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#`



(a) Named Entity Annotation (b) Keyword Annotation

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 value**	**Description**
`predicates`	`null`	List of interesting predicates to enrich the model, and their Objects' values. The predicates are given in form of Map structure where the key is a user-defined name for the predicate and the map value is the predicate itself. e.g. ⟨"predicate1","`http://www.w3.org/2003/01/geo/wgs84_pos#lat`"⟩.

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 Sample code to run the module

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

Listing 1: Code fragment to call the `URIDereferencing` class.

```java
public static void main( String[] args ){
  String datasetSource="";
  List<String> predicatesLines=null;
  if(args.length > 0){
    for(int i=0;i<args.length;i+=2){
      if(args[i].equals("-d") || args[i].equals("--data"))
        datasetSource = args[i+1];
        if(args[i].equals("-p") || args[i].equals("--predicate"))
          predicatesLines= getConfigurations(args[i+1]);
        }
    }
    try{
      Model
          model=org.aksw.geolift.io.Reader.readModel(datasetSource);//First
          parameter: model is loaded with dataset from specified
          file/endpoint
      //Collect list of targeted predicates into Map
      Map<String, String> parameters= list2map(predicatesLines);
      //create Dereferencing object to start the process
      URIDereferencing u = new URIDereferencing();
      // run the dereferencing process it requires model contains
          the dataset and list of targeted predicates to enrich the
          model
      Model resultedModel = u.process(model, parameters);
      resultedModel.write(System.out,"TTL");
    } catch (Exception e) {
      e.printStackTrace();
      }
}
```

The dereferencing module can be also directly executed from command line using the provided `dereferencing.jar` file. The following example introduce calling it to enrich the *Berlin DBpedia* resource with additional predicates located at `predicatesList.txt` file.

```
java -jar dereferencing.jar -d http://dbpedia.org/page/Berlin -p
predicatesList.txt
```

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.

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 Sample code to run the module

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

Table 2: Linking parameters description

Parameter Name	Default value	Description
`Specification file`	`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. `linkingModuleData/linking/spec.xml`. The parameter's entry in th Map structure has key 'specFilePath'.
`Links file`	`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. `linkingModuleData/linking/links.nt`. The parameter's entry in th Map structure has key 'linksFilePath'.
`Original URI position`	`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'. The parameter's entry in th Map structure has key 'linksPart'.

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

```
public static void main(String[] args){
  Map<String, String> parameters=new HashMap<String, String>();
  String linksPath = "";
  if(args.length > 0){
    for(int i=0 ; i<args.length ; i+=2){
      if(args[i].equals("-d"))
      parameters.put("datasetSource",args[i+1]);
      if(args[i].equals("-s")){
        parameters.put("specFilePath",args[i+1]);
        linksPath =
            args[i+1].substring(0,args[i+1].lastIndexOf("/"))+"/accept.nt";
        parameters.put("linksFilePath",linksPath);
      }
      if(args[i].equals("-p"))
        parameters.put("linksPart",args[i+1]);
      }
    }
    try{
      Model model =
          org.aksw.geolift.io.Reader.readModel(parameters.get("datasetSource"));
      Linking l = new Linking();
      model = l.process(model, parameters);
      try{
        File file = new File(linksPath);
        file.delete();
      }catch(Exception e){
          e.printStackTrace();
        }
      model.write(System.out,"TTL");
    } catch (Exception e) {
        e.printStackTrace();
      }
  }
```

The linking module can be directly executed from command line using the provided `linking.jar` file. The following example introduces its execution to enrich loaded dataset from *dataset.ttl* with additional links to another dataset according to specifications in `spec.xml` file.

```
java -jar linking.jar -d dataset.ttl -s spec.xml -p target
```

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 `http://geoknow.org/ontology/relatedTo` 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 .

4.3.4 Sample code to run the module

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

The Nlp module can be directly executed from command line using the provided `nlp.jar` file. The following example introduces its execution to enrich *DBpedia Berlin* resource with additional Geo-spatial data through NLP using the FOX light version (-l true). The output will be written in the `enriched.ttl` file.

```
java -jar nlp.jar -i http://dbpedia.org/resource/Berlin -o output.ttl
-l true
```

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

```java
// Define geoEnricher object
NlpGeoEnricher geoEnricher= new NlpGeoEnricher();
// Define parameters Map
Map<String, String> parameters = new HashMap<String, String>();
// Set parameters
parameters.put("useFoxLight", "true");
parameters.put("askEndPoint", "false");
// read input Model
Model inputModel =
    org.aksw.geolift.io.Reader.readModel(inputFileURI);
// Enrich the Model
Model enrichedModel = geoEnricher.process(inputModel, parameters);
// Use the enriched output model
enrichedModel.write(System.out,"TTL");
```

5 Running GeoLift From Command Line

The GeoLift different modules can be directly executed from command line using the provided `geolift.jar` file. The following example introduces the command line execution of GeoLift to enrich `example.ttl` with additional

Geo-spatial data. the GeoLift command line read the input model from the configuration from a tape separated file `config.tsv`. The output will be written in the `output.ttl` file.

```
java -jar geolift.jar -i example.ttl -c config.tsv -o output.ttl
```

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.

[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.

Table 3: NLP parameters description

Parameter Name	**Default value**	**Description**
`input`	`N/A`	The input file/URI to be enriched
`output`	`N/A`	The output file to write the enriched model to it
`litralProperty`	`Top Ranked`	Literal property used by FOX for NER, if not set the top ranked literal property is pecked by `LiteralPropertyRanker` 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.
`useFoxLight`	`false`	Use the light version of FOX, setting it generates faster execution time but less accurate results)
`askEndPoint`	`false`	Ask the *DBpedia* endpoint for each location returned by FOX (setting it generates slower execution time but more accurate results)
`foxType`	`TEXT`	FOX input type : { `text` | `url` }
`foxTask`	`NER`	FOX task :{`NER`} for Named Entity Recognition
`foxInput`	`""`	FOX actual input as text or an URL
`foxOutput`	`TURTLE`	FOX output format: { `JSONLD` | `N3` | `N-TRIPLE` | `RDF/`{ `JSON` | `XML` | `XML-ABBREV`} | `TURTLE` }
`foxUseNif`	`false`	FOX generates NIF: { true | false }
`foxReturnHtml`	`false`	FOX returns HTML: { `true` | `false` }