FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

# ASSIGNMENT OF MASTER'S THESIS

| | |
|---|---|
| **Title:** | Enrichment of the DBpedia NIF dataset |
| **Student:** | BA. Pragalbha Lakshmanan M.A. |
| **Supervisor:** | Ing. Milan Dojčinovski, Ph.D. |
| **Study Programme:** | Informatics |
| **Study Branch:** | Web and Software Engineering |
| **Department:** | Department of Software Engineering |
| **Validity:** | Until the end of summer semester 2019/20 |

## Instructions

DBpedia is a crowd-sourced community effort that aims at the extraction of information from Wikipedia and providing this information in a machine-readable format. One of the core datasets behind DBpedia is the DBpedia NIF dataset which provides the content of all Wikipedia articles in 128 languages. When using the dataset for training different NLP tasks, there is a need to pre-process the dataset, e.g. tokenization, sentence splitting, POS tagging, enrichment with additional links, etc. The ultimate goal of the thesis is to enrich the dataset with additional information, where the main challenge is the size of the dataset.
Guidelines:
- Get familiar with the DBpedia NIF dataset.
- Analyze the existing text pre-processing methods.
- Select and adapt several pre-processing methods (for several languages) for the DBpedia NIF dataset.
- Apply the implemented methods on several DBpedia NIF languages.
- Validate and evaluate the results.

## References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 7, 2019

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

# Enrichment of the DBpedia NIF dataset

***Bc. Pragalbha Lakshmanan***

Supervisor: Ing. Milan Dojchinovski, Ph.D.

28th May 2020

# Acknowledgements

First and foremost, I would like to thank my parents for motivating me and supporting me during my studies in Czech Technical University. Next, I would like to thank god for his unfailing love and affection throughout my life. It is with incredible appreciation that I acknowledge the help of my supervisor Mr.Milan Dojchinovski. Without his assistance, guidance and support this thesis would not be possible.

Besides my supervisor, I would like to thank the Czech Technical University in Prague for giving me an opportunity to study in this prestigious institution.

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 28th May 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology


**Citation of this thesis**

Lakshmanan, Pragalbha. *Enrichment of the DBpedia NIF dataset.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

# Abstrakt

DBpedia je komunitním projektem který má za cíl poskytnout obsah článku z Wikipedie ve strojově čitelném formátu. DBpedia poskytuje získanou informaci jako NIF dataset obsahující všechny články z Wikipedie v 128 jazycích. Cílem diplomové práce je obohatit datový soubor o výsledky rozdělení na věty, rozdělování tokenu, nacházení částí řeči a tokenu a obohacení odkazů pro Wikipedia články v Anglickém, Francouzském, Německém, Španělskem a Japonským jazycích. Následně na výsledku pouštíme různé NLP úlohy, konkrétně rozdělení vět, Tokenizaci a označování částí řeči. Později přispějeme do DBpedia komunity přidáním dalších odkazů na články z Wikipedie. Nakonec vyhodnotíme a zkontrolujeme statistickou výsledků. Obohacení datasetu výsledkami těchto úloh bude nápomocné pro provedení složitějších a víc mocných NLP úloh.

**Klíčová slova** DBpedia, NIF dataset, předběžné zpracováni, NLP úlohy, dalších odkazu

# Abstract

DBpedia is a crowd-sourced community effort which aims at extracting information from Wikipedia articles and providing this information in a machine-readable format. DBpedia provides the extracted information as NIF datasets with the content of all Wikipedia articles in 128 languages. The aim of the thesis is to enrich this dataset with additional information by providing the results of splitting sentences, segregating tokens, finding parts of speech for tokens and enhancing links for the content of Wikipedia articles in English, French, German, Spanish and Japanese languages. The implementation consists of performing NLP tasks namely sentence splitting, tokenization, part of speech tagging on the pre-processed NIF datasets. Eventually contributing to the DBpedia community by adding additional links to the Wikipedia articles. Finally, evaluating the runtime of various NLP tasks and checking the accuracy of the results statistically. Enriching NIF dataset with the result-set of NLP tasks generated from the tool, is useful for performing more complicated NLP task(s).

**Keywords** DBpedia, NIF dataset, text pre-processing, NLP, enhancing links

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

Wikipedia defines DBpedia as "DBpedia (from 'DB' for 'database') is a project aiming to extract structured content from the information created in the Wikipedia project" [3]. DBpedia community project [4] extracts knowledge from Wikipedia and makes it widely available using Semantic Web standards [5]. DBpedia community provides this extracted knowledge in a machine-readable format. It allows users to semantically query relationships and properties of Wikipedia resources. DBpedia generates several sets of datasets to represent the extracted knowledge and DBpedia NIF dataset is one among them. DBpedia NIF dataset [6] provides content of all Wiki articles in 128 languages. The knowledge is obtained from different Wikipedia language editions, thus covering more than 100 languages, and mapped to the DBpedia community ontology. The resulting datasets are linked to more than 1000 other datasets in the Linked Open Data (LOD) cloud [7]. DBpedia project was started in 2006 and has meanwhile attracted large interest in research.

DBpedia community [8] uses a flexible and extensible framework to extract structured information from Wikipedia primarily by representing the knowledge present at Wikipedia Infoboxes. DBpedia represents the Wiki pages in NLP Interchange Format (NIF) in order to broaden and deepen the quantity of structured data. DBpedia NIF provides all information directly extractable from the HTML source code divided into three datasets namely NIF context, NIF page structure and NIF text links.

DBpedia NIF dataset is stored in RDF triples. Resource description framework is realized with the concept of triples (Subject - Predicate - Object). NIF datasets are available in different formats on the official DBpedia site [9].
Key features of NIF datasets [10] are:
1) Content available in over 128 Wikipedia languages.

2) Over 9 billion RDF triples, which is almost 40% of DBpedia.
3) Selected partitions published as linked data [11].
4) Exploited within the TextExt- DBpedia Open Extraction challenge.
5) Available for large-scale training NLP methods.

Various Natural Language Processing (NLP) tasks are trained on the NIF dataset. Applying NLP over the dataset provides meaningful information which is used to enrich the NIF dataset with additional information. NLP is a branch of artificial intelligence that deals with the interaction between computers and humans using the natural language. The ultimate objective of NLP is to read, decipher, understand, and make sense of the human languages in a manner that is valuable. Most NLP techniques rely on machine learning to derive meaning from human languages. NLP entails applying algorithms to identify and extract the natural language rules such that the unstructured language data is converted into a form that computers can understand. With natural language, data can be assessed, analysed and communicated with precision. The NLP tasks implemented in the thesis include:
1) Sentence splitting - To split a given paragraph of text into sentences, by identifying the sentence boundaries.
2) Tokenization - Chopping up a large sample of text into words called tokens, perhaps at the same time throwing away certain characters such as punctuations.
3) POS tagging - Must read text and assign parts of speech to tokens or words that is detected.
4) Links enhancement- Every Wikipedia article has a set of links or hyperlinks that redirects the page to another Wikipedia article. This NLP task increases the amount of such links.

The following pre-processing tasks had to be done for the implementation of above mentioned NLP tasks:
1) NIF datasets are not confined to an individual Wikipedia article. NIF context dataset contains the full text of all Wiki pages from a particular language in a single file. Separation of this huge dataset into individual file for each article is required as this allows faster and easier processing of the dataset. This also allows NLP tasks to be performed on a subset of NIF context dataset.

2) In order to perform the enhancing links NLP task, it is necessary to have the list of possible surface forms (links on the Wiki articles). NIF text links dataset provides details regarding the surface forms. Recording the details of surface forms on a CSV file helps enhancing links on a Wiki article by searching the tokens one by one on the generated CSV file. The CSV file should consist the tokens of surface form, it’s link to the Wiki

page (DBpedia resource) and it's part of speech. The combination of these pre-preprocessing tasks along with the NLP tasks are my main focus on the thesis.

Since DBpedia is an open source organisation, I admire the effort put by the team and am motivated to contribute my part to this organisation. DBpedia project was started roughly 15 years ago and has meanwhile attracted huge amount of interest among various research teams and engineers. My contribution to DBpedia community is the result-sets of the 4 NLP tasks namely sentence splitting, tokenization, part of speech and enhancing links performed on NIF datasets for 5 different languages namely English, French, German, Spanish and Japanese. These result-sets serves as an input for performing a wide variety of other complicated tasks such as sentiment analysis. I have performed each NLP task with multiple libraries which serve me as an opportunity to compare the accuracy of these libraries. The produced result-sets gives an opportunity to analyse and experiment it by finding exquisite insights such as average number of sentences in an article, most frequently used part of speech, average number of tokens in each article, average number of links etc.

## 1.1 Goals

The goals of the thesis are:
1) Develop a tool for performing NLP tasks namely sentence splitting, tokenization, part of speech tagging and links enhancement on NIF dataset(s).

2) Performing NLP tasks for multiple languages. The tool should work for English, French, German, Spanish and Japanese languages. A documentation on integrating a new language to the developed tool should be specified.

3) Integrate multiple NLP libraries to perform each of the NLP tasks. Tool currently supports libraries namely Natural Language Tool Kit package, TextBlob , Spacy, StanfordPOSTagger, Pattern, Nagisa and Konoha. A documentation on integrating a new library to the developed tool should be specified.

4) The tool should be configurable. User should be able to generate the result-set of the NLP tasks with the opportunity to choose the task (sentence splitting, tokenization, part of speech, links enhancement) to be performed, number of Wiki articles on which the NLP task should be performed, library with which the tasks must be performed and the language

(among English, Spanish, German, French and Japanese). This allows generation of the output files in a much simpler fashion based on user's choice.

5) Evaluate the performance of NLP tasks by comparing the runtime among different libraries. Evaluate the accuracy of the generated results for each task through different libraries. Generated result-set should be analysed to gain insights and should be depicted statistically.

Chapter 2

# Background and Related works

It is necessary to get familiar with certain concepts before moving on to my contribution to the DBpedia community. Such concepts include Resource Description Framework (RDF), linked data, DBpedia projects, contents of NIF dataset(s) and natural language processing in general. I have performed a case study on the related works which are summarised in the Section 2.2.

## 2.1 Background

This section provides understanding of RDF, linked data, DBPedia NIF dataset and NLP with short explanations and examples wherever necessary. A clear understanding of these topics are considered to be a prerequisite for achieving my goals.

### 2.1.1 Resource Description Framework

Resource Description Framework (RDF) [12] is a family of specifications developed by the organization World Wide Web Consortium (W3C), originally designed as a metadata model. It is used as a general method for modelling information in different syntaxes. RDF is a standardized format that lets you express descriptive information about web resources. The source described by RDF could be any resource that can be uniquely identified by a Uniform Resource Identifier (URI). This identifier uniquely determines the particular resource involved in a web content composed of different types of documents.

#### 2.1.1.1 Triples

A semantic triple is an atomic data entity in the Resource Description Framework (RDF) data model. As its name indicate, a triple is a set of three

entities that codifies a statement about semantic data in the form of subject, predicate and object expressions. Given this semantic data, it allows to be queried without any ambiguity. An example such as "Elephant has the colour black" symbolically implies "elephant" as the subject, "colour" as predicate and "black" as object. The object oriented design has a similar notation as entity, attribute and value model ie. the entity is "elephant", attribute is the "colour" and value is "black".

What makes RDF triples special is that every part of the triple could have a URI associated with it, so a statement "John's age is 23" might be represented in RDF as:

```
<http://database.org/people#John> <http://database.org/attribute/age> "23"
```

Statements encoded in triples could be spread across different websites as shown:

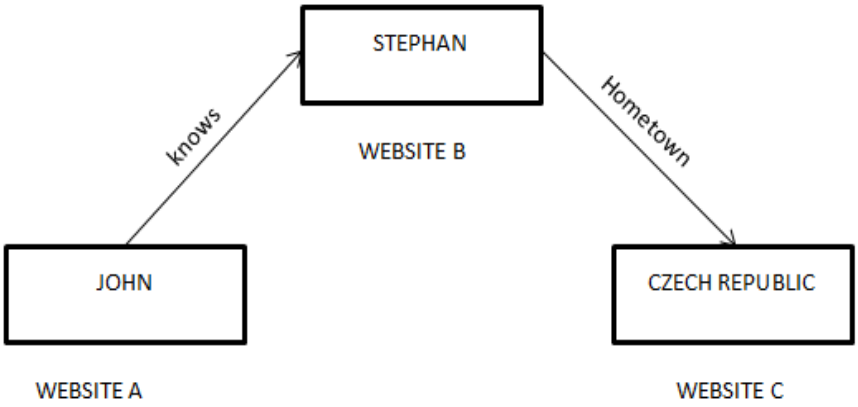


Fig. 2.1. Representation of RDF

Website A presents the entity John and the fact that he knows Stephan. Website B provides all the information about Stephan and on the Website C we can find information about Stephan's hometown. Each page contains the structured data to describe an entity.

#### 2.1.1.2 Various Serialization Formats

**Turtle** : It is a syntax and file format for expressing data in the Resource Description Framework (RDF) data model. DBpedia NIF datasets are available in the turtle Format. RDF in terse RDF triple language (turtle) format is much easier as you can define prefixes at the beginning of the file, shortening each triple. Another feature of turtle is that multiple triples with the same subject are grouped into blocks for instance:

```
1 @prefix dbr: <http://dbpedia.org/resource/>
2 @prefix nif: <http://persistence.uni-leipzig.org/nlp2rdf/ontologies/
      nif-core#>
3 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
4 dbr:Animalia_(book) a nif.Context
5 nif.beginIndex "0"
6 nif.endIndex "2294"
7 nif.predLang <http://lexvo.org/id/iso639-3/eng>
```

Listing 2.1: Sample triples from NIF context dataset

There are 3 prefixes declared on each of the first 3 lines of the listing. The set of triples starts from Line 4 with all the triples having the same subject as "dbr:Animalia_(book)". One unique shortening is the predicate <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> represented with an alphabet 'a'. The subject is not repeated for the rest of the 3 triples as it is common for all of them. On Line 5, the predicate is 'nif.beginIndex' and the object is "0". Similarly the next two lines have predicate and object pairs. So, totally there are 4 triples in this listing.

**N-triples** : Storing and reading RDF as N-Triples is simple since every line on the file has a single triple (<subject> <predicate> <object>) that together forms a directed knowledge graph. N-Triples is a format for storing and transmitting data. It is a line-based, plain text serialisation format for RDF graphs. For example:

```
1 <http://one.example/subject1> <http://one.example/predicate1> <http
      ://one.example/object1> .
2 _:subject1 <http://an.example/predicate1> "object1" .
3 _:subject2 <http://an.example/predicate2> "object2" .
```

Listing 2.2: Examples of N-triples serialization format

**RDF/XML** : It is an XML based syntax that is the first standard format for serializing RDF. Like turtle, prefixes can be defined at the top of RDF/XML files to avoid unnecessary repetition of URIs. RDF/XML is still not as humanly readable as turtle.

**Notation 3** : It is a shorthand non-XML serialization of resource description framework models, designed with human-readability in mind. N3 is much more compact and readable than XML RDF notation. A non-standard serialization that is very similar to turtle, but has some additional features such as the ability to define inference rules.

### 2.1.2 Linked Data

Linked data [11] is using the web to connect related data that wasn't previously linked, or using the web to reduce the barriers to linking data currently linked using other methods. Wikipedia defines linked data as "a term used to describe a recommended best practice for exposing, sharing, and connecting pieces of data, information, and knowledge on the Semantic Web using URIs and RDF" [13].

According to Tim Berners-Lee, "The Semantic Web isn't just about putting data on the web. It is about making links, so that a person or machine can explore the web of data. With linked data, when you have some of it, you can find other, related, data." [14]. In addition Tim Berners-Lee coined the four main principles of Linked data:
1) Use URIs as names for things.
2) Use HTTP URIs so that people can look up those names
3) When someone looks up a URI, provide useful information.
4) Include links to other URIs. So that they can discover more things.

The idea behind these principles is to use standards for representation and access to data on web. The principles propagate to set hyperlinks between data from different sources. These hyperlinks connect all linked data into a single global data graph similar to the hyperlinks on the classic web which connects all HTML documents into a single global information space. The rationale of these principles uses URIs to identify individuals, classes, and properties.
Names perform two important roles:
1) It refers to the relevant thing
2) It give us a location on the web where we could look for information about the thing itself. Computing linked data describes a method for publishing and linking data coming from heterogeneous data sources that can be interlinked and shared.

#### 2.1.2.1 Linked Open Data

Open data is the data that can be freely used and distributed by anyone. Linked Open Data (LOD) is a powerful blend of Linked Data and Open Data. It is both linked and uses open sources. One notable example of an LOD set is DBpedia. Linked Data breaks down the information that exist between various formats and brings down the fences between various sources. It facilitates the extension of the data models and allows easy updates. As a result, data integration and browsing through complex data become easier and much more efficient.

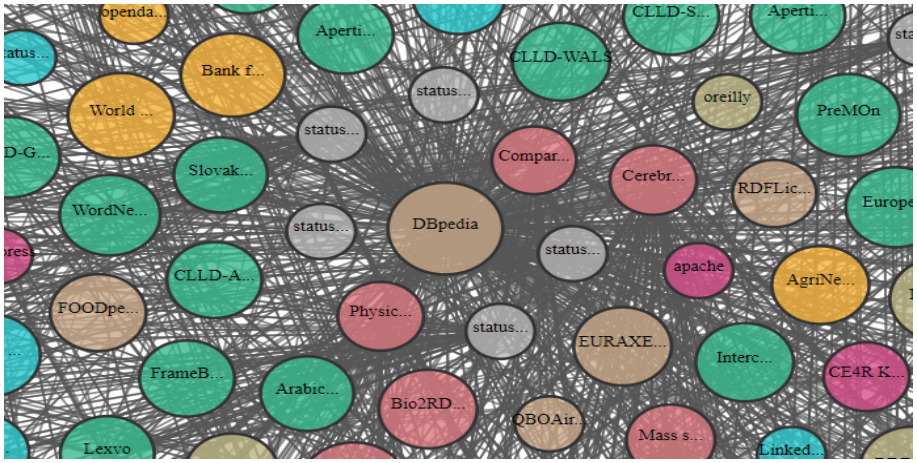


Fig. 2.2. Linked Open Data Cloud

The Linked Open Data Cloud [7] depicts publicly available linked datasets. LOD Cloud has been divided into nine subclouds each one representing a separate knowledge domain: geography, government, linguistics, life science, media, publications, social networking, user-generated and cross-domain (anything else like DBpedia, Wikidata). Each colour of the bubble represents different subcloud. It currently contains over 1,200 datasets with 16,000 links. DBpedia contains 4.6 million concepts which are described by over 1 billion triples [2].

### 2.1.3 DBpedia

DBpedia is a project aiming to extract structured content from the information created in the Wikipedia project. This structured information is made available on the World Wide Web (WWW). DBpedia allows users to semantically query relationships and properties of Wikipedia resources, including links to other related datasets. DBpedia is a crowd sourced community effort to extract structured content from the information created in various Wikimedia (a mission to bring free educational content) projects. Wikimedia includes Wikipedia, Wiktionary, Wikiquote, Wikibook, Wikisource etc. Through various projects, chapters, and the support structure of the non-profit Wikimedia foundation, Wikimedia strives to bring about a world in which every single human being can freely share all knowledge. This structured information resembles an open knowledge graph which is available for everyone on the web. A knowledge graph is a special kind of database which stores knowledge in a machine readable form and provides a means for information to be collected, organised, shared, searched and utilised.

A knowledge base is used to store complex structured and unstructured information used by a computer system. Some important points about DBpedia knowledge base are:
1) It covers many domains
2) It represents real community agreement
3) It automatically evolves as Wikipedia changes
4) It is truly multilingual
5) Support for information integration.

DBpedia's knowledge base for the English version consists of 4.58 million things. DBpedia knowledge base for French, German, Spanish and Japanese versions consists of 1.78 million, 1.9 million, 1.3 million and 0.9 million things respectively. Out of the 4.58 million things in English language, about 4.22 million are for the consistent ontology which includes 1.4 million persons, 0.75 million places (including 0.48 million populated places), 0.41 million creative works (including 123,000 music albums, 87,000 films and 19,000 video games), 0.24 million organizations (including 58,000 companies and 49,000 educational institutions), 0.25 million species and 6,000 diseases [2].

DBpedia provides localized versions of DBpedia in 128 languages. Looking at all the versions, DBpedia provides 38.3 million things, out of which 23.8 million are localized descriptions of things. The full DBpedia dataset features 38 million labels and abstracts in 125 different languages. It can be noticed that only 11.7% of the total Wikipedia articles belongs to English. About 10.7% belongs to Cebuano, a language spoken in Phillipines (most of the articles just has one or two sentences), 7.5% belongs to Swedish and 4.5% in German and 4.2% in French. DBpedia consists of 4.58 million things in English compared to the overall 38.3 million things which is about 13%. The ratio is very close to that of 11.7% in Wikipedia.

DBpedia's knowledge could help us get interesting information for very complex queries unlike Wikipedia. For eg. it is possible to retrieve answers for these queries through DBpedia: Display all the Cities in the world with a population more than 10 million people or display all Fresco painters/artists from the 18th century. Altogether, the use cases of the DBpedia knowledge base are widespread and range from enterprise knowledge management over web search to revolutionizing Wikipedia search.

DBpedia data is served as linked data, which is revolutionizing the way applications interact using the web. One can navigate this web of facts with standard web browsers, automated crawlers or pose complex queries with SQL like query languages (eg. SPARQL). DBpedia dataset provided in RDF triples is hosted and published by using "OpenLink Virtuso". The access to the DBpedia's RDF dataset is provided through SPARQL endpoint,

alongside HTTP support for any web client's standard GET requests for HTML or RDF representations of DBpedia resources.

#### 2.1.3.1 DBpedia Ontology

DBpedia Ontology has been manually created from the infoboxes present inside the Wikipedia. An infobox is a fixed-format table usually added to the top right-hand corner of articles to consistently present a summary of some unifying aspect that the articles share and sometimes to improve navigation to other interrelated articles. Many infoboxes also emit structured metadata which is sourced by DBpedia and other third party re-users. The generalized infobox feature grew out of the original taxoboxes (taxonomy infoboxes) that editors developed to visually express the scientific classification of organisms. DBpedia ontology can also be queried via the DBpedia SPARQL endpoint and can be explored via the DBpedia linked data interface. The table below lists the number of instances for several classes within the ontology for English language.

| Class | Instances |
|---|---|
| Resource (overall) | 4,233,000 |
| Place | 735,000 |
| Person | 1,450,000 |
| Work | 411,000 |
| Species | 251,000 |
| Organisation | 241,000 |

Table 2.1: DBpedia ontology for English language [2]

DBpedia ontology is shallow and available across multiple domains. DBpedia ontology currently uses 685 classes which are described by about 2800 different properties.

### 2.1.4 DBpedia NIF Dataset

DBpedia's primary focus is on retrieving the factual knowledge from the Wikipedia infoboxes. With the representation of wiki pages in the NLP Interchange Format (NIF), DBpedia provides all information directly extractable from the HTML source code divided into three datasets:
1) NIF context dataset: The full text of a page as context.
2) NIF page structure dataset: the structure of the page in sections and paragraphs (titles, subsections etc).
3) NIF text links dataset: Links in the Wikipedia articles

These datasets will serve as the groundwork for further NLP fact extraction tasks to enrich the gathered knowledge of DBpedia. From release

2016-10, they have started providing the full text of Wiki pages in NIF format.

The NLP Interchange Format (NIF) aims to achieve interoperability between NLP tools, language resources and annotations. To extend the versatility of DBpedia, furthering many NLP related tasks, they decided to extract the full text of any Wikipedia page (NIF context), annotated with NIF tags. For this first iteration, they restricted the extent of the annotations to the structural text elements directly inferable by the HTML (NIF page structure). In addition, all contained text links are recorded in a dedicated dataset (NIF text links).

#### 2.1.4.1 NIF 2.0 Core Ontology

NIF 2.0 Core Ontology [15] provides classes and properties to describe the relation between substrings, text, documents by assigning URIs to strings. The main class in this ontology is 'nif:String', which is the class of all words over the alphabet of unicode characters.
A pictorial representation of various relationships:

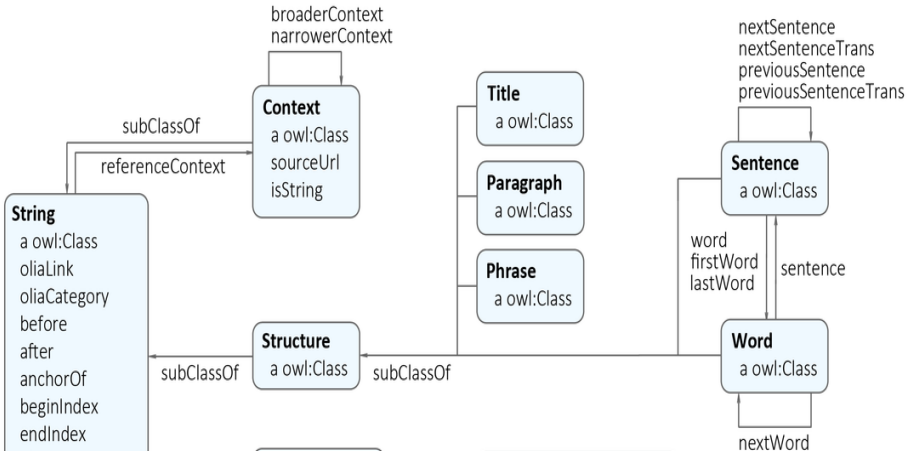


Fig. 2.3. NIF Core Ontology [1]

The subclass of 'nif:String' is the 'nif:Context' OWL class. This class is assigned to the whole string of the text (ie. all characters). The purpose of an individual of this class is special because string of an individual is used to calculate the indices for all substrings. Therefore, all substrings should have a relation 'nif:referenceContext' pointing to an instance of 'nif:Context'. The datatype property 'nif:isString' consists of full text of Wiki page. 'nif:word' could be combined with either 'nif:title', 'nif:phrase' or 'nif:paragraph' which are defined in the 'nif:Structure'. Thus, 'nif:word'

is a subclass of 'nif:Structure' which in turn is a subclass of string. All the subclasses can use the properties of parent class.

#### 2.1.4.2 NIF Context

The full text present in each Wikipedia page is available in NIF context dataset. I have used the NIF context dataset versions in English, French, German, Spanish and Japanese languages for my thesis work. Each Wiki page has 6 triples on NIF context dataset. The triples in NIF context dataset for the English Wiki article "Anthropology" are :

```
1 dbr:Anthropology?dbpv=2016-04&nif=context a nif:
    Context.
2 dbr:Anthropology?dbpv=2016-04&nif=context nif:isString "
    Anthropology is the study of humanity. Its main subdivisions are
    ." <The full text of Anthropology Wiki page> .
3 dbr:Anthropology?dbpv=2016-04&nif=context nif:beginIndex "0" .
4 dbr:Anthropology?dbpv=2016-04&nif=context nif:endIndex "634" .
5 dbr:Anthropology?dbpv=2016-04&nif=context nif:sourceUrl <http://
    en.wikipedia.org/wiki/Anthropology> .
6 dbr:Anthropology?dbpv=2016-04&nif=context nif:predLang <http://
    lexvo.org/id/iso639-3/eng> .
```

Listing 2.3: DBpedia NIF context for Anthropology Wiki page

Each line on this dataset consists of one triple. Every line has a subject, predicate and object separated by spaces among them. The triples mentioned on the listing are taken from 'nif_context_en.ttl' file. The name of the Wiki page appears on the subject of each triple occurring between the namespace "dbr:" and the question mark "?". Each triple gives the following information in chronological order:
1) These triples are a part of NIF context dataset.
2) The full text present in Wikipedia page.
3) The begin index of the page which is always 0.
4) The end index of the full text of Wiki page.
5) URL link to access the Wikipedia article via browser.
6) Indicates the language of Wiki page.
The 'nif_context_en.ttl' file consists these 6 triples for all the Wiki pages in English language. Although DBpedia provides 3 different extensions of NIF context dataset - TTL,TQL and HDT, I used the TTL format due to availability of favourable libraries in Python for parsing and processing in my tool.

#### 2.1.4.3 NIF Page Structure

The structure of the Wikipedia page as NIF:Structure instances, such as section, paragraph and title. A sportsman's Wikipedia page for instance

contains various sections such as Early age, personal life, achievements etc. Each such section could contain multiple paragraphs within them. It is necessary to keep track of these sections, titles and paragraphs in DBPedia for maintaining the amount of structured data. The following is an example taken from NIF page structure dataset :

```
1 dbr:Anthropology?dbpv=2016–04&nif=context nif:hasSection
      dbr:Anthropologydbpv=2016–04&nif=section_0_634 .
2 dbr:Anthropology?dbpv=2016–04&nif=section_0_634 nif:beginIndex
      "0" .
3 dbr:Anthropology?dbpv=2016–04&nif=section_0_634 nif:endIndex
      "634" .
4 dbr:Anthropology?dbpv=2016–04&nif=section_0_634 nif:
      referenceContext dbr:Anthropologydbpv=2016–04&nif=context .
5 dbr:Anthropology?dbpv=2016–04&nif=section_0_634 nif:hasParagraph
      dbr:Anthropologydbpv=2016–04&nif=paragraph_0_330 .
6 dbr:Anthropology?dbpv=2016–04&nif=section_0_634 nif:hasParagraph
      dbr:Anthropologydbpv=2016–04&nif=paragraph_331_634 .
7 dbr:Anthropology?dbpv=2016–04&nif=section_0_634 nif:firstParagrap
      dbr:Anthropologydbpv=2016–04&nif=paragraph_0_330 .
8 dbr:Anthropology?dbpv=2016–04&nif=section_0_634 nif:lastParagraph
      dbr:Anthropologydbpv=2016–04&nif=paragraph_331_63 .
```

Listing 2.4: DBpedia NIF page structure for Anthropology Wiki page

These triples correspond to Anthropology Wiki page informing about the section and paragraph details. The following could be inferred from the triples in chronological order:
1) Wikipedia page 'Anthropology' has section(s)
2) Section's begin index on the Wiki page
3) Section's end index on the Wiki page
4) Content's of this section could be seen in NIF context file ie. the full text present within this section
5) There exists a paragraph in this section from index 0 to 330
6) There exists a paragraph in this section from index 331 to 634
7) The paragraph with index 0 to 330 is the first paragraph
8) The paragraph with index 331 to 634 is the last paragraph

Similarly there is a description of each paragraph within the section as can be viewed below:

```
1 dbr:Anthropology?dbpv=2016–04&nif=paragraph_0_330 a nif:
      Paragraph .
2 dbr:Anthropology?dbpv=2016–04&nif=paragraph_0_330 nif:beginIndex
      "0" .
3 dbr:Anthropology?dbpv=2016–04&nif=paragraph_0_330 nif:endIndex
      "330" .
4 dbr:Anthropology?dbpv=2016–04&nif=paragraph_0_330 nif:
      referenceContext dbr:Anthropology?dbpv=2016–04&nif=context .
```

```
5 dbr:Anthropology?dbpv=2016–04&nif=paragraph_0_330      nif:superString
      dbr:Anthropology?dbpv=2016–04&nif=section_0_634  .
```

Listing 2.5: DBpedia NIF page structure wrt paragraph

This describes the paragraph properties and links the paragraph to its section with the "superstring" attribute. We could infer the following from each of the triples:
1) This is a paragraph taken from article Anthropology
2) The begin index of the paragraph is 0.
3) The end index of the paragraph is 330.
4) The reference to this paragraph is available in the NIF context dataset.
5) It is a part of the section from the index 0 to 634 in the Anthropology wikipedia page. It can be noticed from the predicate being "superString".

#### 2.1.4.4 NIF Text Links

This dataset has the properties of 'words' containing link(s) to other DBpedia resources (Wiki pages have a corresponding DBpedia resource). In other words, this dataset consists of links present in the Wiki pages. I have used the NIF text links dataset versions in English, French, German, Spanish and Japanese languages for developing my tool. A line taken from Wikipedia page of Anthropology in English language:
"Anthropology is the scientific study of humanity, human behavior and societies in the past and present."

Here "humanity" has a link on it, which when clicked redirects the browser to Wiki page of "human". Such words that has a link is called as Surface Form. In this dataset, properties of such surface forms are recorded. The details of "humanity" is stored as follows in NIF text links dataset:

```
1 dbr:Anthropology?dbpv=2016–04&nif=word_41_49     a     nif:Word .
2 dbr:Anthropology?dbpv=2016–04&nif=word_41_49     nif:referenceContex
    dbr:Anthropologydbpv=2016–04&nif=context .
3 dbr:Anthropology?dbpv=2016–04&nif=word_41_49     nif:beginIndex
    "41" .
4 dbr:Anthropology?dbpv=2016–04&nif=word_41_49     nif:endIndex
    "49" .
5 dbr:Anthropology?dbpv=2016–04&nif=word_41_49     nif:superString
    dbr:Anthropology?dbpv=2016–04&nif=paragraph_0_634 .
6 dbr:Anthropology?dbpv=2016–04&nif=word_41_49     <http://www.w3.org
    /2005/11/its/rdf#taIdentRef>     dbr:Human .
7 dbr:Anthropology?dbpv=2016–04&nif=word_41_49     nif:anchorOf     "
    humanity" .
```

Listing 2.6: DBpedia NIF text links for Anthropology Wiki page

Each of the triples inform the following in chronological order:
1) The surface form has only one word. If the surface form has more than one word, then the object is ’nif.Phrase’. Since ’humanity’ has one token, the object is ’nif.Word’.
2) The content is available in NIF context dataset referencing to DBpedia resource of Anthropology.
3) Begin index of the surface form in the Wiki page
4) End index of the surface form in the full text of the page
5) It is a part of the paragraph with index 0 to 634 on the Anthropology DBpedia resource.
6) Theoretically when ’humanity’ is clicked, the browser would redirect us to the Wiki page of human. Here "dbr:" is a namespace. This triple informs the DBpedia resource that this surface form is linked to.
7) The tokens of the surface form. Actual word that has the link from index 29 to 37 is "humanity".

### 2.1.5 Natural Language Processing

Natural Language Processing (NLP) in simple terms is the intersection of computer science, linguistics and machine learning. It is in concern with the communication between humans and computers in natural language. NLP is all about enabling computers to understand and generate human language. Applications of NLP techniques are voice assistants like Alexa and Siri but also machine translation and text-filtering. NLP is very much benefited by the recent advancement of machine learning especially from the deep learning. The field is divided into the three following parts:
1) Speech recognition is the translation of spoken words into text.
2) Natural language understanding is the computers ability to understand what we say.
3) Natural language generation is the generation of natural language by a computer.

Syntactic analysis (Syntax) and Semantic analysis (Semantic) are the two main techniques for understanding of natural language. Language is a set of valid sentences, but what makes a sentence valid? Actually, you can break validity down into two things: syntax and semantics. The term "syntax" refers to the grammatical structure of the text whereas the term "semantics" refers to the meaning that is conveyed by it. However, a sentence that is syntactically correct, does not have to be semantically correct. Just take a look at the following example. The sentence ’pigs flow adversely’ is grammatically valid (subject verb adverb) but does not make any sense.

Text Segmentation [16] in NLP is the process of transforming text into meaningful units which can be words, sentences, different topics, the underlying intent and much more. Mostly, the text is segmented into its

component words, which can be a difficult task, depending on the language. This is again due to the complexity of human language. For example, it works relatively well in English to separate words by spaces, except for words like 'ice box' that belong together but are separated by a space. The problem is that people sometimes also write it as 'ice-box'.

Python supports wide varieties of NLP libraries. Python is one of the widely used languages and it is implemented in almost all fields and domains. Some of the widely used NLP libraries are:
1) Natural Language Toolkit (NLTK) [17].
2) spaCy [18].
3) Pattern [19].
4) TextBlob [20].
5) CoreNLP [21].
Refer Section 3.2 for more information regarding the libraries used to build my tool.

#### 2.1.5.1 NLP Tasks

This section provides general understanding of various NLP tasks. Description of each task in basic words along with some of the challenges faced are listed out.

**Sentence splitting** - Sentence splitting means to split a given paragraph of text into sentences, by identifying the sentence boundaries [22]. In many cases, a full stop is all that is required to identify the end of a sentence, but the task is not all that simple. Let's look at some of the challenges faced:
1) Abbreviations: Dr. H.N.Abraham is the author of Animalia. In this case, the first and second dot(.) occurring after 'Dr' (Doctor) and 'H' (initial in the person's name) respectively. This should not be confused with the full stop.
2) Sentences enclosed in quotes: "What good are they? They're led about just for show!" should be identified as one sentence.
3) Questions and exclamations: What is it? - This is a question. This should be identified as a sentence. "I am tired!" - something which has been exclaimed. This should also be identified as a sentence.
4) Short forms for instance 'etc.' or 'ie.' doesn't refer to a sentence boundary. Although this contains a dot, it doesn't imply the end of a sentence.

The challenges differs for French, German, Spanish and Japanese languages for eg. Japanese language does not have spaces in written language. Every language is treated differently following their specific rules and standards while performing this NLP task.

**Tokenization**- Given a string or a paragraph or a sequence of characters and a defined document unit, tokenization [23] is the operation of cutting it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuations. Word tokenization is the process of splitting a large sample of text into words. Some of the challenges faced in this task:
1) Words that belong together but are separated by a special character such as "ice-box".
2) Numbers, special characters, hyphenation, and capitalization. In the expressions "don't," "I'd," "John's" do we have one, two or three tokens?
3) Removal of stop words. Once the tokens are separated lets say "This is it.", here the tokens should be "this" "is" and "it", however not "this" "is" "it." . So detecting stop words and separating it could be tricky.
4) German language has compound words wherein multiple words are written without spaces in between each of them.

**Part of speech** - Part of speech is a category to which a word is assigned in accordance with its syntactic functions. Some words have multiple meanings and they could be used in different contexts. The part of speech varies depending upon the context in which the word appears. On looking at the following sentence:
"They refuse to permit us to obtain the refuse permit"
The word refuse is being used twice in this sentence and has two different meanings here. Refuse is a verb meaning "deny," while refuse is a noun meaning "trash" (that is, they are not homophones). Thus, we need to know which word is being used in order to pronounce the text correctly (For this reason, text-to-speech systems usually perform POS tagging).

Numbers are usually adjectives because the information they give is how many of the "noun" present. They can be cardinal (like one, two, three), or ordinal (like first, second, third). Punctuations don't have part of speech. Part of speech in Japanese is quite different as Japanese is a SOV (Subject-Object-Verb) language whereas English is typically SVO (Subject-Verb-Object). In Japanese, the verb always appears at the end of clauses and sentences. Japanese parts of speech are usually marked with words called "particles" that follow the word they modify. A sentence or paragraph should be provided as an input in order to find the POS rather than using a token as the usage of token could differ based on the context.

**Links enhancement** - Enhancing the number of surface forms in Wiki pages ie. increasing the links in Wiki pages. While every Wikipedia page already contains few links, the goal of this task is to increase the amount of such links. This should ideally help the readers to enhance the knowledge of a certain page in depth. Some important challenges to be considered for this task are:

1) A surface form could appear multiple times within an article. Duplication of providing links to same surface forms on a particular Wiki article must be avoided. If a surface form occurs multiple times within a Wiki article, in this case only the first occurrence of the surface form must be given a link.
2) Should not provide too many links on a particular Wiki page. Could be achieved by ignoring obvious words or basic words.
3) Should be able to provide link to surface forms that has more than one word.
Please refer the Section 3.7 for a detailed explanation on link enhancement task and the challenges faced on it.

## 2.2 Related Work

This section provides understanding of related work to my thesis. Related work could be anything that applies NLP over Wikipedia articles or annotates Wikipedia articles or just enriches Wikipedia in general. Some of the notable related works are "Annotating documents with relevant Wikipedia concepts" by Wikifier, "A Model for Enriching Multilingual Wikipedias Using Infobox and Wikidata Property Alignment" by Thang Hoang Ta and "Extracting and annotating Wikipedia Subdomains" by Gisle and Flickinger.

### 2.2.1 Annotating documents with relevant Wikipedia concepts

This work is similar to the "Links enhancement" NLP task. In this work, they provide links based on the pagerank of Wiki pages. The tool developed for this work is called Wikifier [24]. All words, phrases from an input document (that needs to be annotated) referring to certain concepts from the Wikipedia are listed out.

All the possible concepts to Wiki page(s) are listed out for the input document using the internal links from the Wikipedia to identify such phrases:
1) If some Wikipedia page contains a link with the anchor text ’a’ and target page ’t’
2) Whenever ’a’ occurs in the input document, they consider that as a (possible) mention of the concept ’t’, and ’t’ is a candidate annotation for this input document.

The problem is that the links with the same anchor text ’a’ can point to different targets ’t’. They tried to overcome this by using a pagerank-based global disambiguation approach.
The approach to solve this issue :
• Construct a mention-concept graph

• Bipartite graph: left vertices = mentions, right vertices = concepts
• Transition probabilities: P(a -> t) = [number of links with anchor text a and target t] / [number of links with anchor text a]

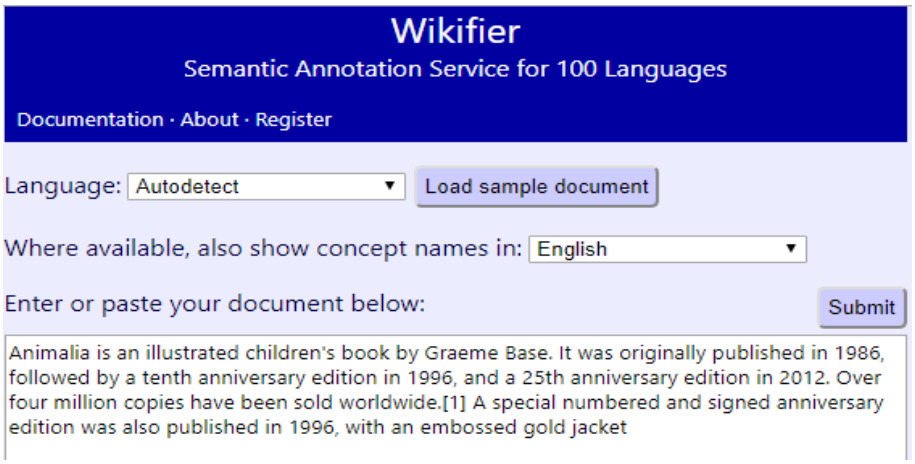


Fig. 2.4. Wikifier User Interface

The goal of Wikifier is to relate words on input document to appropriate Wikipedia concepts whereas goal of my thesis is to enhance links on Wiki pages by providing links to all surface forms. This tool relates Wikipedia concepts for any textual file whereas I have considered only full text of Wiki pages to be enhanced with links. The algorithm used is different from each other as well. Their algorithm chooses Wiki page with better pagerank for surface forms with multiple links. On the other hand, my work finds the POS of surface form. Surface forms with multiple links have different parts of speech. Depending upon the result of POS, links are assigned accordingly. My work is expected to give more accurate result compared to Wikifier. The surface form with the better pagerank always wins on Wikifier, so there are no chances for Wiki pages with lower pagerank to be given a link.

### 2.2.2 Model for Enriching Multilingual Wikipedias

A model for enriching multilingual Wikipedias using infobox and Wikidata property alignment [25]. It provides some processes to enrich Wikipedia content which will retrieve semantic relations based on alignment between infobox properties and Wikidata properties in various languages. The tool developed for this work compares the infobox contents to Wikipedia data and enriches the Wiki pages by filling the additional information present on infobox but are missing on Wiki pages. This work enriches Wikipedia in general. The goal of this work is similar to mine. The processes carried out in this work are:

- Align infobox parameters with Wikidata properties. A semi-automated tool is created to support searching and aligning the semantic equivalence between Wikidata properties (or items which has no "relevant property") and infobox properties.
- Detect missing inter Wiki links and connecting them to articles in different languages as well as synthesize all semantic relations. Comparisons of semantic relations and assess their correlations to detect missing interwiki links.
- Enrich article content and Wikidata statements after implementing the comparisons of gathered semantic relations. They enriched more data for articles which have new inter Wiki links.

Semantic analysis has been performed in this work. Semantic analysis in simple terms is the process of understanding the natural language ie. the way humans communicate based on meaning and context. The NLP tasks are performed on infobox and Wikidata in this project. However, my tool applies NLP tasks only on NIF datasets (ie. text of Wikipages). The infoboxes used in this work emits structured metadata which is considered to be a source for DBpedia datasets.

### 2.2.3 Extracting and Annotating Wikipedia Sub-Domains

This paper suggests a simple procedure for the extraction of Wikipedia sub-domains, proposes a plain-text (human and machine readable) corpus exchange format reflecting on the interactions of Wikipedia markup and linguistic analysis [26]. In their work, they have used a variety of NLP tasks - sentence segmentation, grammatical analysis, and discriminant based treebanking and the interactions of document mark-up, linguistic analysis. They have proposed a simple technique of compiling and annotating domain specific corpora of scholarly literature, initially drawing predominantly on Wikipedia. This is a related work to mine as NLP is applied over Wikipedia. For a given sub-domain, this work extracts and annotates the Wiki pages with results of various NLP tasks that fall under the given sub-domain.

These are the related works to my thesis. The work goals between my tool and the related works are compared. I have performed some of the NLP tasks which are used in these related works. My main goal is to enrich the DBpedia NIF dataset which is completely different from any of these. The results of my NLP tasks are stored in the format that complies with DBpedia norms.

Chapter 3

# Data Processing

This chapter provides description of the tool developed for enriching NIF dataset. It also provides explanation of pseudocodes and algorithms for performing various NLP tasks on multiple languages of NIF datasets by using different libraries. This chapter starts with a workflow explaining the step by step process carried out to in my thesis. This chapter also describes about the Python libraries used to build my tool.

Data processing [27] is generally a collection and manipulation of data to produce meaningful information. Carrying out operations on data especially by a computer to retrieve, transform, or classify information. With respect to my thesis, NIF context is the dataset on which various NLP operations are performed (sentence splitting, tokenization, part of speech tagging and enhancement of links). NIF text links dataset has been used for the pre-processing task. However, NLP operations were not performed on NIF text links dataset.

NIF context and NIF text links datasets have been downloaded from the official DBpedia download page [9]. DBpedia provides NIF datasets separately for each language ie. NIF context, NIF text links and NIF page structure datasets are provided separately for each language. I downloaded NIF context & NIF text links datasets for English, French, German, Spanish and Japanese languages. Size of the extracted file vary upon language. Although the datasets are available for other languages, I have taken these 5 languages as the primary focus of my thesis. In case if a user wants to integrate a new language with my tool, a documentation on how this could be achieved is mentioned in Usage chapter. The information provided on NIF structure dataset is not required for my work and was not considered.

## 3.1 Workflow

A flowchart to depict the steps carried out in the Thesis for building the tool :

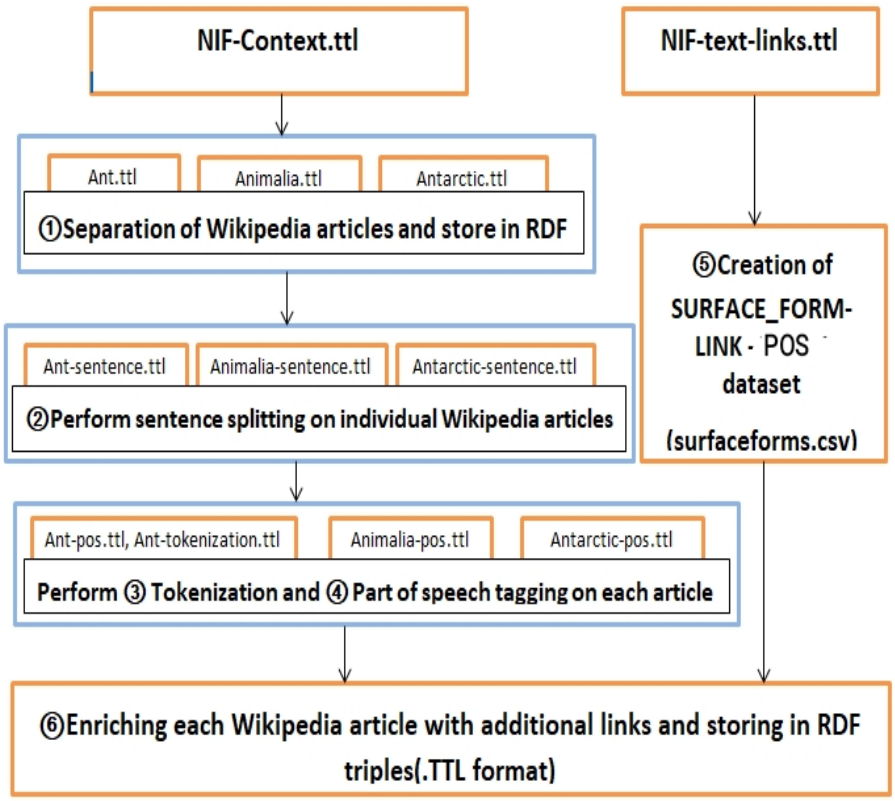


Fig. 3.1. Flowchart

Explanation of each step mentioned on flowchart :

- **STEP 1** Separation of Wikipedia articles: As NIF context dataset consists the triples of all the Wiki pages in one huge file for every language, there is a need to separate this huge file into triples pertaining to each Wiki article. The triples corresponding to each Wiki article must be stored separately. Since there is a separate NIF context dataset for each language, this step has to be done for all languages. This step helps to process data faster while performing various NLP operations on it. This also allows NLP tasks to be performed on a subset of the NIF datasets. A new directory will be created for each language to store the separated files from NIF context dataset. The naming convention of the directory is "Input<language_name>/". Each of the generated output files from this task contains its corresponding Wiki

page's triples obtained from NIF context dataset. These small files are saved with the name of their Wiki page. On looking at the Spanish NIF context dataset as an example "nif_context_es.ttl" has a size of 4.5 GB after extraction. This dataset comprises triples of all the 1.3 million Wiki pages in Spanish language. After step 1, a new directory called 'Inputes/' will be created containing the generated 1.3 million small files inside them. Each small file will have the name of its Wiki page, containing all its 6 triples obtained from nif_context_es.ttl". I have discussed the pseudocode of this task in section 3.3.

- **STEP 2** Sentence Splitting: Perform sentence splitting on files generated as an output to the previous step. Every small file generated in the previous step contains 6 triples and only one of these triples contain the full text of a Wiki page. This is the only triple which is required for performing the sentence splitting task and thus needs to be uniquely identified among the other five. The object of this uniquely identified triple will contain the full text of Wiki page. Split the obtained full text into individual sentences and store each of these sentences as a separate triple on the output file. The begin and end index of each sentence is recorded as the object of the predicates "nif:beginIndex" and "nif:endIndex" respectively. The contents of each sentence are stored in the object with predicate "nif:anchorOf". This task has to be performed on all 5 languages. The ideal outcome of this task should separate the full text of Wiki pages into its individual sentences. The separated sentences are stored in individual triples with appropriate namespaces, specifications [28] and annotations that complies with the DBpedia standards, adding great benefit to DBpedia's usefulness. The output file must be of turtle format. A directory to store the results of sentence splitting is created as "Sentences/". This NLP task has been performed with 5 different libraries namely NLTK, spaCy, TextBlob, Pattern and Konoha in Python.

- **STEP 3** Tokenization: This step should detect all the tokens present in the full text of Wiki pages. The full text of the Wiki page is obtained from one of the six triples contained in the output files of Step 1. The begin and end index of every token is recorded as the objects to the predicates "nif:beginIndex" and "nif:endIndex" respectively. The content of each token is provided as the object having predicate "nif:anchorOf". This task has to be performed on all 5 languages. I have discussed the algorithm to perform this task on Section 3.5. The ideal outcome of this task should detect all tokens on the full text of Wiki pages. The detected tokens should be stored in triples with

appropriate namespaces, specifications and annotations [28] that complies with the DBpedia standards. The output of tokenization task is stored on the directory "Tokens/". The results are stored with the name of the file same as their Wiki article name. The output file must be in turtle format containing details of each token. For instance, lets say there are 6000 words on a Wikipedia article, then the generated result should contain 6000 tokens. The generated output files for this task are stored with the name of the Wiki article, under "Tokens/" directory. Tokenization is performed with 4 different libraries namely NLTK, spaCy, TextBlob and nagisa. The results of each library are slightly different from each other.

- **STEP 4** POS Tagging: This step is to find the part of speech for each token. In this case, we cannot use the output of previous task to find the parts of speech. Every token could have multiple parts of speech and in order to detect the correct part of speech for the given word, it is always recommended to use the sentences or paragraphs as an input for this task. For instance, the word 'next' could be used as an adjective, adverb, preposition or even as a noun depending upon the context it is used. This NLP task has been performed with 4 different libraries namely Stanford POSTagger, spaCy, TextBlob and NLTK. The begin index, end index and content of each token are recorded along with the token's part of speech in the output file. Section 3.6 provides description of the implementation. The output file of this task should contain all triples that are in the output files of tokenization task. In addition, there should be three additional triples indicating the short form of token's POS, full form of token's POS and the output obtained from POS tagger. For instance let's consider a token called "Prague", the output of POS tagger is "NNP" (result obtained from NLP library), short form POS is "Noun" and the full form POS for this token is "Proper Noun". A directory called "POS/" is created to store the results of part of speech tagging task. The parts of speech for the tokens belonging to an individual article are stored with the name of the Wiki page that this token belongs to, on "POS/" directory in a format complying with DBpedia's norms.

- **STEP 5** Surface Form Analysis: All the existing surface forms are obtained from NIF text links dataset and thus this dataset is downloaded for 5 different languages from the official DBpedia Download page [9]. The goal of this step is to create CSV file containing 3 important attributes about the "Surface Forms". First attribute should consist the tokens present in a surface form (the word(s) that has link).

The second attribute should have the URL link to the DBpedia resource that will load when the corresponding surface form is clicked. The third attribute should consist the part of speech for the tokens present in the surface form. A surface form could contain more than one token. Parts of speech for the token(s) in surface form must be contained as the third attribute of the CSV file. The generated CSV file could contain duplicates and it is better to remove the duplicates for performing faster search operation on this file. This step is considered to be a preprocessing task for the Links enhancement NLP task. My tool consists a separate script for removing the duplicates on the generated CSV file from NIF text links file. Naming convention of the generated CSV file after removal of duplicates is "LinkDataset<language>.csv". For French language the CSV file is saved as "LinkDatasetfr.csv".

- **STEP 6** Links Enhancement: This is the final step wherein the Wiki pages are enhanced with additional links. Additional links are provided to the full text of Wiki pages and thus the output obtained from step 1 is used as an input to this task. The list of all surface forms already existing are available in the CSV file generated from the previous step. Read the full text of Wiki pages token by token in order to detect the tokens that are surface forms. If the content and part of speech of token(s) read on full text of Wiki pages matches the content and part of speech of a surface form entry on the CSV file, then a link is provided to these read tokens(s) for their respective DBpedia resource. Also, the longest possible match is taken into consideration. For instance let's take the words "colouring book" , here "colouring" and its POS has a match on csv file. Whereas "colouring book" and its POS has a match on the CSV file too. In this case, the link is provided to the longest surface form being "colouring book". This NLP task is performed with 3 different libraries namely spaCy, TextBlob and NLTK. A new directory called "Links/" is created which stores the result-set of the articles with enhanced links. The begin index, end index, content of the word and the link to the DBpedia resource are recorded as triples in the result-set with appropriate specifications and annotations [29] that complies with the ITS 2.0 standards.

These are the 6 main steps carried out in my thesis. The upcoming sections would describe these steps in detail with the help of pseduocodes and some examples. I used Python programming language and Shell scripting for developing my tool. Python is used for implementation of the NLP tasks and Shell scripting is used for interacting with the user in order to obtain the input to reproduce the result-set(s) for making my tool

configurable. The Shell script would in-turn call the appropriate Python script(s) for generating the output. Python programming language supports Object Oriented Programming (OOP) which is used for defining a class with attributes and methods that can be called later. This will be easier to organise and structurize the scripts used to build my tool. Creation of "main.py" - a central script in my tool used for calling the appropriate Python script(s) based on user's input and helps to follow a structured order for managing various python scripts.

## 3.2 Python libraries

This section covers all the Python libraries used to build my tool with short explanations. Also snippets taken from my python scripts for performing NLP tasks are provided as examples wherever necessary.

### 3.2.1 RDFLib

RDFLib [30] is a Python library for working with RDF, a simple yet powerful language for representing information. Some important features of RDFLib are:
1) RDFLib contains parsers and serializers for RDF/XML, N3, Turtle formats.
2) The library presents a graphical interface which can be backed by any one of the store implementations. It contains both in-memory and persistent graph backends.
3) The core RDFLib includes store implementations for in-memory storage, persistent storage.
4) A wrapper for remote SPARQL endpoints. Supports SPARQL queries and update statements.

RDFLib reads the triples from NIF datasets and stores it as a graph. The 'parse()' function takes name of the file and format as parameters. Parsing with RDFLib segregates the subject, predicate and object of every triple. This helps in uniquely identifying one out of the 6 triples containing the full text of Wiki pages, on the output files generated from Step 1. On parsing input file, the subject, predicate and object of each triple are accessible through a loop. RDFLib is used for storing the result-set of NLP task in sets of triples on the turtle format.

### 3.2.2 NLTK

NLTK [17] is a leading platform for building Python programs to work with human language data. It is a widely used NLP library in Python. Some of the cool features provided by NLTK :

1) It provides easy to use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial strength NLP libraries.

2) Processing the given dataset with various operations. NLP operations performed using NLTK:

- Separating the content of a file with individual sentences

```
1 sentence_result = nltk.sent_tokenize(o,language=''+assign+'')
```

  Here, the variable 'o' corresponds to the full text of Wiki pages on which sentences are split. Language is detected based on the user's input while executing the tool. It is supplied as a parameter to the sentence tokenize function. The variable 'sentence_result' is a list having each sentence stored on a different index.

- Separation of individual words

```
1 token_result = nltk.word_tokenize(txt,language=''+assign+'')
```

  The variable 'txt' is a sentence or a paragraph on which the tokenization is to be performed. Language is detected from the user's input to my tool and provided as a parameter to the word tokenize function. The variable "assign" contains the language name. Word tokenization is performed on the text present on variable 'txt' and the individual generated tokens are stored on different indices of the list "token_result".

- Finding the part of speech

```
1 tagged = nltk.pos_tag(tokens,lang=''+assign+'')
```

  The variable 'tagged' is a list storing tokens and their parts of speech as pairs on separate indices of the list.

- Separate words by spaces using WhitespaceTokenizer.

3) It can perform sentence splitting, tokenization and part of speech for multiple languages but the accuracy is good only for the English language. This library was built primarily for English.

For German, Spanish and French I have used StanfordPOSTagger on top of NLTK to perform the NLP tasks. I have used NLTK for performing all the 4 NLP tasks in English and NLTK with StanfordPOSTagger for French, German and Spanish languages.

### 3.2.3 Spacy

spaCy [18] is a library for advanced NLP in Python and Cython which comes with a variety of interesting features. Its becoming increasingly popular for processing and analysing data in NLP. Key features :

- spaCy comes with pre-trained statistical models and word vectors currently supporting tokenization for 49 different languages
- It features state of the art speed, convolutional neural network models for tagging, parsing, named entity recognition and easy deep learning integration.
- spaCy encodes all strings to hash values to reduce memory usage and improve efficiency

I have used spaCy for performing all 4 NLP tasks in English, German, French and Spanish languages. spaCy doesn't support Japanese language. On identifying the language based on user's input, appropriate spaCy language model is loaded as follows:

```
1 nlp = spacy.load(''+lang+'_core_news_sm')
2 sentences = nlp(o.encode().decode('utf-8'))
3 for i in sentences.sents:
4         content = nlp(i.text.encode().decode('utf-8'))
5         for j in content:
6                 print(j.text)
7                 print(j.tag_)
```

Listing 3.1: NLP tasks using Spacy

Each of the lines in the above mentioned snippet does the following on chronological order:
1) The variable 'lang' stores the language name obtained from user and is used to load the appropriate spaCy language model. All the Spacy language model names follows the same naming pattern - "<language name>_core_news_sm". The variable 'nlp' has the loaded Spacy language model.
2) The variable 'o' on the second line refers to the full text of a Wiki page. The variable 'sentences' stores the list of separated sentences from the full text of Wiki page, in a hashed format. Displaying variable 'sentences' would return a hashed hexadecimal value. Using the sents property ie. 'sentences.sents' it is possible to store the contents of each sentence stored on a separate index of the list.
3) Iterating over every sentence present on Wiki page in human readable format.
4) The 'content' stores the list of tokens on these sentences in hashed format.

5) Loop over all tokens on the list
6) Displays all the tokens in human readable format.
7) Displays all the parts of speech for the corresponding tokens listed on step 6.

### 3.2.4 StanfordPOSTagger

Stanford CoreNLP [21] provides a set of human language technology tools. Stanford CoreNLP's goal is to make it very easy to apply a bunch of linguistic analysis tools to a piece of text. Stanford CoreNLP integrates many of Stanford's NLP tools, including the POS tagger, the named entity recognizer (NER), the parser, the coreference resolution system, sentiment analysis, bootstrapped pattern learning, and the open information extraction tools. The library variously uses rule-based, probabilistic machine learning, and deep learning components. This software is a Java implementation of the log-linear POS taggers. It supports 9 languages but the downside is that the time taken to detect the parts of speech are longer than NLTK or Spacy or Textblob. This library provides 'fast-tagger' for German language whose execution time is considerably quick.
A small snippet on usage of this library:

```
st=StanfordPOSTagger(''+assign+'-ud.tagger')
if lang=="de":
    st=StanfordPOSTagger('german-fast.tagger')
    tagged=st.tag(txt.split())
```

Listing 3.2: Loading StanfordPOSTagger model

The 'StanfordPOSTagger()' function loads the appropriate POS tagger model depending upon language chosen by user which is supplied to this script by variable 'assign'. The loaded tagger is stored on variable 'st'. The fast-tagger version is loaded for German language and ud-tagger is loaded for other languages. The 'tag()' function detects the parts of speech of the tokens and stores it on the 'tagged' variable.

### 3.2.5 TextBlob

TextBlob [20] is a Python library for processing textual data. It provides a simple API for diving into common NLP tasks such as POS tagging, noun phrase extraction, sentiment analysis, classification, translation, WordNet integration, parsing, word inflection, adds new models or languages through extensions, and more. It supports multiple languages as well. I have used this library to perform all the 4 NLP tasks in my thesis.
A small snippet using TextBlob:

```
blob_object = TextBlob(text)
print(blob_object.tags)
```

The first line creates a textblob object. The variable 'text' inside the blob object can be a sentence, paragraph or even token. Tags property of created blob object stores the part of speech for the tokens. The second line prints the (tokens,POS) pairs of 'text'.

### 3.2.6 Pattern

Pattern [19] is a web mining module in Python. It has tools for data mining (Google, Twitter, and Wikipedia API, a web crawler, an HTML DOM parser), natural language processing (POS taggers, n-gram search, sentiment analysis, WordNet), machine learning (vector space model, clustering, SVM), network analysis by graph centrality and visualization. Pattern supports Python 2.7 and Python 3.6. I have used this library only for the sentence splitting NLP task. This library supports 5 languages.

### 3.2.7 Konoha and Nagisa

Konoha is a tiny sentence/word tokenizer for Japanese text written in Python. Nagisa [31] is a Python module for Japanese word segmentation/POS-tagging. Nagisa is designed to be a simple and easy-to-use library.
This library has the following features:
1) It is based on recurrent neural networks.
2) The word segmentation model uses character and word level features.
3) The POS tagging model uses tag dictionary information.
Usage of Nagisa for NLP tasks is as follows:

```
1 tokens = nagisa.tagging(text)
2 print(tokens.words)
3 print(tokens.postags)
```

Listing 3.3: Tokenization and POS using Nagisa

The variable 'text' is the input text on which NLP operations are to be performed. The 'text' is tokenized and stored on the variable 'tokens' as a list. The "words" property on line 2 displays the tokens of 'text' and line 3 displays the parts of speech for the tokens using postags property.

## 3.3 Separation of Wikipedia articles

This section provides description to the first step in the workflow ie. separation of Wiki articles. The input file to this step is NIF context dataset. After downloading and extracting the NIF context datasets in English (nif_context_en.ttl), French (nif_context_fr.ttl), German (nif_context_de.ttl), Spanish (nif_context_es.ttl) & Japanese (nif_context_ja.ttl) languages, the idea is to separate each dataset into smaller files. Each of the small files must have triples corresponding to a single Wiki page. The total number of

small files to be generated is same as the number of Wiki articles available in a language.

As mentioned earlier, NIF context datasets consists the triples of all the Wiki pages in one huge file for every language. The objective of this step is to separate this huge file into triples pertaining to each Wiki article. The triples corresponding to each Wiki article must be stored separately. This process has to be done for all 5 languages ie. on all the 5 versions of NIF context datasets. This helps to process the dataset quicker and perform various operations faster and easier.

This step allows my tool to perform NLP tasks only for a subset of the dataset based on user's choice. This step is considered to be a preprocessing step for performing the NLP tasks. The pseudocode of this task is as follows :

```
Get the path to extracted NIF context file
Store the language of NIF context file
i is assigned as 29 for English and 32 for other languages
previous Wiki article name is assigned as 'None'
Open NIF context dataset from the given path
    Loop with line from 1 to end of file
        Store the index of '?' on every line
        Store the name of Wiki article that this line belongs to
        if previous and current Wiki article names are same
            Write this line into same file as previous line
        else
            try
              Create a new file with the current Wiki article name
              Write this line into newly created file
            except
              pass
        previous wiki article name is assigned to current one
```

Listing 3.4: Pseudocode for separation of NIF context dataset

Explanation about the Python and Shell scripts in the tool, for separating Wikipedia articles:
1) The user has to execute the tool by running Shell script "separate_scripts.sh" with a path argument. The path argument should point to the location of extracted version of the NIF context file (of any language). The Shell script validates the path to NIF context file. If the validation is successful, the Shell script would then call the Python script whose pseudocode is mentioned above in the listing.

2) The Python library "sys" is required for receiving the location of NIF context file from the Shell script. For passing the value from Shell script to the Python script "sys" supports a property called "agrv" through which values entered by user are read on Python scripts via Shell interface.

3) The Python script used for separating the Wiki articles in my tool detects the language of NIF context dataset provided. It does so by checking the last 6 characters of the location to NIF context file entered in the path argument. The naming convention for NIF context dataset is "nif_context_<language-name>.ttl". The language is obtained from the sixth last index to fourth last index from the name of NIF context file. It is necessary to find and store the language of NIF context file in order to store the output files of this task in appropriate language directory.

4) A variable 'i' is declared as 29 if the language is English. Otherwise it is declared as 32. This variable assists in retrieving the article name of every Wiki page. Every line in NIF context file corresponds to one triple showed as follows, taken from English NIF context file:

```
<http://dbpedia.org/resource/Andrea_Andreani?dbpv=2016-10&nif=context>  <http://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core#beginIndex>  "0" .
```

The name of Wiki article here is "Andrea Andreani". Its index in the subject of the triple is from 29 till the occurrence of '?'. The index '29' is not expected to change in the future releases of DBpedia NIF datasets as they follow a convention of defining subject of the triples in such a manner. This is the reason why the index 29 is hard coded. On looking at a sample triple from French version of NIF context dataset :

```
<http://fr.dbpedia.org/resource/Cardonville?dbpv=2016-10&nif=context>  <http://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core#beginIndex>  "0" .
```

The name of the Wiki article is "Cardonville". Its index in the subject of the triple is from 32 till the occurrence of '?'. There is a change in the begin index because of the appearance of "fr." before the "dbpedia.org" on every subject of the triples.

5) The tool then opens the NIF context file mentioned on the given location, in "UTF-8" encoding style. This encoding style allows to read special characters if they appear inside the dataset. A loop to read all lines present in the file is created. This loop finishes on reaching the end of the file until all the lines are read.

6) Since every line consists a triple in this dataset, I will start addressing the lines of NIF context dataset as triples of NIF context dataset. For each triple on the dataset, find the name of the Wiki article that this triple belongs to. Each Wiki article consists of 6 triples and all these 6 triples appear one after the other in the NIF context dataset. The name of the

Wiki article could be obtained from index 29 to the first appearance of question mark(?) in case of English language. Otherwise it is obtained from index 32 to '?'.

7) Check if the previous triple's Wiki article name is same as the current triple's Wiki article name. If they are same, the current triple is added to the same file as that of previous triple. If they are different, a new file is created with the name of current Wiki article under the 'Input<language-name>/' directory and this triple is added to the newly created file. For the Japanese language, the newly created file gets saved on "Inputja/" directory where 'ja' indicating Japanese language.

8) There exists a 'try' block on line 12 within the else condition because some operating systems doesn't support certain file names that includes characters like '/', '*', ':', '?', '|' as these characters are reserved. Also there are some reserved keywords such as CON, PRN, AUX, NUL, COM1, LPT1 on Windows. However, these names and characters are very unlikely to appear as the Wiki article name. This try block prevents stopping the execution of the tool in case any of these reserved keywords appear as a name of Wiki article. If these keywords or special characters appear as the name of the Wiki article then the output file isn't generated and basically the triple is not stored. Again, there are hardly be 1000 files that has these characters or keywords. This is a very negligible number compared to the number of files that are being dealt with.

9) The previous Wiki article name is assigned as the current Wiki article name before an iteration completes. After this, it reads the next line in the NIF context dataset, if the name of the article is same it continues to add this triple on the same output file as that of the previous triple. If the name of the article is different, in this case a new output file is created with the name of the current Wiki article. And this loop goes on until the end of the file is reached.

So the triples belonging to same Wiki article name are stored on one small file saved in the name of Wiki article that the line (triple) belongs to. This is a very convenient way of storing so that in the future these files could be accessed easily. There is another argument which the shell script on my tool will accept apart from the path to the location of NIF context dataset. This optional argument is **Search** wherein a user can do the separation of Wiki articles only for a subset of NIF context file. Some users might be interested in only a subset of Wiki articles to be separated. It takes roughly about 30mins for entire nif_context_en.ttl to be separated into its individual articles. So, in this case if a user wants only the Wiki articles starting with alphabet 'A' separated from NIF context file - it is

achievable using the argument (-s) for filtering out articles that start with alphabet 'A' by executing the shell script as follows:
An example of execution with search argument:
"./separate_scripts.sh -p F:/Master_thesis/nif_context_en.ttl -s A" (Separates all Wiki articles that starts with "A" from NIF Context file in English Language and stores the results in "Files/Inputen/" directory).

It is also possible to specify name of an article that has to be separated from NIF context dataset. The shell script on my tool calls a different python script in case the search argument is entered by the user. The implementation of the search feature is similar to that of the one mentioned above with one additional step in the pseudocode as mentioned:

```
1 Open NIF Context file from the given path
2     Loop with line from 1 to end of file
3         if 'dbpedia.org/resource/<VALUE-TO-SEARCH>' exists on this
              triple(line)
4             same steps as the previous pseudocode
5         else
6             skip this triple(line)
```

Listing 3.5: Pseudocode for separating a subset of Wiki articles

This prevents separating those triples that doesn't satisfy the search condition. The "if" condition on line 3 ensures this and separates only the triples of Wiki articles those satisfying the search condition. An output file generated as a result of this task:

```
1 @prefix nif: <http://persistence.uni-leipzig.org/nlp2rdf/ontologies/
      nif-core#>
2 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 <http://dbpedia.org/resource/Animalia_(book)?dbpv=2016-10&nif=
      context> nif.beginIndex "0"
4 rdf.type nif.Context
5 nif.endIndex "2294"
6 nif.predLang <http://lexvo.org/id/iso639-3/eng>
7 nif.sourceUrl <http://en.wikipedia.org/wiki/Animalia_(book)?oldid
      =741600610>
8 nif.isString "< full text of Animalia_(book) article>"
```

Listing 3.6: Output file of this task - Animalia_(book).ttl

Explanation of the output with each point same as the line number on the listing:
1) A namespace "nif" is a prefix with the URL to NIF core ontology. Once defined as a prefix, it uses the namespace throughout the file instead of repeating the long URL everytime. The word "nif" is used as a substitute to the long URL. This conserves lot of space.
2) Another namespace "rdf" defined as a prefix pointing to "W3C".

3) This is the first triple. The URL within angular brackets is the subject and it is the same for all the triples. That's why it isn't repeated in the upcoming triples. All the other triples just have predicate-object pair (lines 4-8). This triple implies the begin index of the content (full text) of wiki page "Animalia_(book)".
4) These set of 6 triples are a part of NIF context file
5) The end index of Wiki article
6) The content of this Wiki article is in English
7) The URL for accessing this wikipedia article on a browser
8) The full text of this article always appearing with predicate in "nif:isString". Each of the output files have 6 triples inside them.

## 3.4 Sentence Splitting

This section provides understanding of the first NLP task ie. sentence splitting for each or a subset of the small files generated from the 'Separation of Wiki articles' step. The input to this task are the output files generated from previous step. Out of the 6 triples in every file, only one of the triple contains the full text of Wiki pages. This is the only triple which is required for performing the sentence splitting task and thus needs to be uniquely identified. Sentence splitting is performed on the object of this uniquely identified triple. The general challenges regarding the sentence splitting can be found on Section 2.1.5.1. The ideal outcome of this task should separate the full text into its individual sentences and store the begin index, end index and content of every sentence in turtle format. The pseudocode for this task is as follows:

```
1  Get the language and instance size as input
2  Initialize count as 0
3  Loop with files one by one in "Input<lang_name>/" directory
4    If count is less than the instance size
5      Create graph through RDFLIB
6      Parse the graph with file as a parameter
7      Serialize the graph with the parsed file
8      Store the name of the file
9      Loop over every sub, pred and obj on serialized graph
10       If type of obj is Literal and pred is nif:isString
11         Split sentences on the object via libraries
12         Loop over every sentence on list
13           Find Begin and End index of every sentence
14           Write BI,EI and sentence onto output file
15     Increment count
16   else
17     Break
```

Listing 3.7: Pseudocode for sentence splitting

Explanation about the pseudocode of the Python script in my tool for performing sentence splitting:
1) The user has to run my tool by executing the shell script specifying the arguments regarding language for which sentence splitting must be performed, number of files for which sentence splitting must be performed, task to be performed ie. sentence splitting in this case and the library that should be used to perform sentence splitting. The shell script passes these argument details to "main.py". This central python script manages all the other python scripts in the tool by calling the appropriate python script based on the library and task chosen. The above mentioned pseudocode is common for all the libraries for performing sentence splitting task. Only the commands to perform sentence splitting varies depending upon the library used. I have performed this task with 5 different libraries namely NLTK, Spacy, TextBlob, Pattern and Konoha. So, my tool consists of 5 different python scripts just for the implementation of this NLP task. The differences in result between each library is listed on "Experimentation" chapter.

2) The library "sys" is required for this Python script in my tool, to receive the user's input(s) from "main.py". The library "rdflib" is required for parsing the TTL file generated through the previous step. Since RDFLib also supports Namespace class, it helps in creating prefixes such as 'nif' for "http://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core". The library "os" is needed for reading the input files from "Input<language-name>/" directory. Apart from these 3 common libraries, each implementation of sentence splitting requires its own library to be imported such as NLTK, Spacy, Pattern, TextBlob, Konoha.

3) Once the language to be processed and the number of files for which the sentence splitting has to be performed is obtained, the variable 'count' is initialised as 0 to keep track of instance size. The processing must not continue for files more than the number of files (instance size) mentioned by the user.

4) All the files in the "Input<language-name>/" directory must be iterated one by one. Since the language and instance size are obtained as an input to this script based on user's choice, files from appropriate input directory has to be read one by one. The first step inside the loop is to check if the count is less than the instance size. If the instance size is lesser than the count, then break the loop and stop the execution.

5) If the condition satisfies ie. count is lesser than the instance size, then create a graph through RDFLib for parsing the file from "Input<language-name>/" directory provided as a parameter to the graph. Once parsed and

serialized it is convenient to separate all the triples as subject, predicate and object in the input files.
The parsing and serialization are implemented in Python as follows:

```
graph2=rdflib.Graph()
graph2.parse('Input'+lang+'/'+filename,format='nt')
s=graph2.serialize(format="nt")
```

Listing 3.8: Parsing and serializing graph

The 'graph2' parses and serializes the files present in "Input<language-name>/" directory one by one. The variable 'filename' contains the full name of the files with extension ".ttl". The name without extension is stored in order to name the output file that will be stored on "Sentences/" directory. Therefore, the name of the Wiki article is obtained by extracting content before dot on 'filename' variable.

6) Loop over every subject, predicate and object on the serialised graph. This loop runs 6 times for every file since there are 6 triples present inside each small file generated from the "Separation of Wiki articles" task. As mentioned earlier, we are interested in only the triple containing the full text of the Wiki article. In order to uniquely detect this triple, a condition checks if the type of object is Literal and the predicate is "nif:isString". The full text of the Wiki page is stored as an object to predicate "nif:isString". This object contains numerals as well along with the full text of Wiki pages. I am not interested in this numerals. So, in order to filter out this numerals, the object is checked if it has a "Literal" tag.

7) If the conditions are satisfied (ie. object is "Literal" and predicate is "nif:isString"), perform sentence splitting through each library on the object of this triple. Refer Section 3.2 to see how sentences are split through different libraries. Also, the source code is added as a part of the CD attached along with my Thesis. The result of the split sentences are stored on a variable in a list format. Iterate over every sentence present in the full text of Wiki pages in order to find its begin and end index in the full text of their corresponding Wiki page. Begin and end index could be found by searching for the sentences inside the object (whose predicate is nif:isString and object being Literal) of the triple. The search will be successful definitely because we are basically searching for a sentence from the paragraph which is obtained from the paragraph. On successful search python returns its begin index. End index is calculated by adding the length of the sentence to the begin index.

8) All the content needed to write into the output file (begin index, end index and content of every sentence) are available. Therefore, all these details are written in triples to the output TTL file. There are 5 triples

required for displaying the output of sentence split as per the DBpedia norms.
Storing result-set of sentence splitting is as follows:
Syntax: <graph>.add([subject,predicate,object])

```
1 g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
      =2016-10&nif=sentence_offset_"+str(BI)+"_"+str(EI)),RDF.type,nif
      .Sentence])
2 g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
      =2016-10&nif=sentence_offset_"+str(BI)+"_"+str(EI)),nif.
      beginIndex,rdflib.term.Literal(str(BI))])
3 g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
      =2016-10&nif=sentence_offset_"+str(BI)+"_"+str(EI)),nif.endIndex
      ,rdflib.term.Literal(str(EI))])
4 g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
      =2016-10&nif=sentence_offset_"+str(BI)+"_"+str(EI)),nif.anchorOf
      ,rdflib.term.Literal(str(i))])
5 g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
      =2016-10&nif=sentence_offset_"+str(BI)+"_"+str(EI)),nif.
      referenceContext,rdflib.term.URIRef("http://dbpedia.org/resource
      /"+name+"?dbpv=2016-10&nif=context")])
6 g.bind("nif",nif)
7 g.serialize(format="turtle"))
8 g.serialize(destination="Sentences/"+filename.ttl,format="turtle")
```

Listing 3.9: Triples to be added to the output file

The graph 'g' is used to store the triples on output files whereas graph 'graph2' was used to read, parse and serailize the input file. "nif" is a namespace created above. And similarly RDF.type is "http://www.w3.org/1999/02/22-rdf-syntax-ns#type". The 5 triples (Line 1 to 5) appear for every sentence in a particular output file, implying the following:
1) The following set of triples will inform details about a sentence.
2) Stores the begin index of the sentence.
3) Stores the end index of the sentence.
4) Stores the content of sentence - every iteration of "i" consists one sentence
5) This data is derived from NIF context file. And finally write the output in a serialized form on a TTL file.

Binding the graph is to give the name "nif" for the namespace "nif". Otherwise, python automatically assigns the name for the namespace created as "ns1" standing for namespace 1.
A sample of the output file looks like as shown, the following is taken from "Sentences/Animalia_(book).ttl" file :

```
<http://dbpedia.org/resource/Animalia_(book)?dbpv=2016-10&nif=
    sentence_offset_1179_1273> a nif:Sentence ;
    nif:anchorOf "The Great American Puzzle Factory created a 300-
        piece jigsaw puzzle based on the book's cover." ;
    nif:beginIndex "1179" ;
    nif:endIndex "1273" ;
    nif:referenceContext <http://dbpedia.org/resource/Animalia_(book
        )?dbpv=2016-10&nif=context> .
```

Listing 3.10: Sentence splitting result

It is interesting to note that triples are not stored in the order we saved. It gets jumbled up and sometimes appear in a different order. Since we added the second triple as begin index, however it is stored as third triple. So, if we execute the source code again, the result on output file might appear in a different order and the order in which it appears doesn't matter at all. The subject for all these triples is common, so it appears only on the first triple. Other triples just have predicate-object pairs.

### 3.4.1 Challenges

As far as the challenges are concerned, all the NLP libraries are good enough to detect the salutation, initial of people etc. Let us check one of the sentences from the output file generated on "Sentences/Animalia_(book).ttl" -
"H. N. Abrams published The Animalia Wall Frieze, a fold-out over 26 feet in length, in which the author created new riddles for each letter."
NLTK, spaCy, TextBlob, Pattern libraries can detect question marks as the end of the sentence and other punctuations as well.

However, the following is supposed to be a single sentence -
"Each creature (A is for alligator, B is for butterfly, etc.) is unique."
Some tools detect this as two different sentence since there is a full stop after etcetra within the brackets which is incorrect. However, these type of issues are hard to handle. Could be checked if special character appears immediately after "etc." without any space. However, adding a condition like this would only increase the complexity of the program as this needs to be checked in every iteration.

There is another problem where it doesn't detect sentences wrongly but it is arguable and debatable as shown below:
"External links
1) Graeme Base's official website
2) Animalia The Television Series official website "
The following is detected as a single sentence by NLTK. However I am

not sure if it is right or wrong. It is a side heading with points having incomplete sentences. One way to deal with this is segregating each point as a sentence and the heading external link as a separate sentence. But it is again debatable if it correct or not. Spacy automatically detects this as separate sentences while NLTK detects this as a single sentence.

Spacy supports sentence splitting on English, Spanish, French and German. Loading appropriate language model and using it for sentence splitting is straightforward in spaCy. TextBlob and NLTK supports sentence splitting for these languages too. It is implemented by specifying the language within the argument inside sent_tokenize function. Pattern library does sentence splitting for English, German and French. The sentence splitting in Japanese are well dealt by Konoha tool.

## 3.5 Tokenization

This section describes the second NLP task ie. tokenization for each or a subset of the small files generated from the separation of Wikipedia articles step. Tokenization has to be done on the full text of the Wiki page. The triple with object as Literal and predicate as "nif:isString" consists the full text of Wiki pages and is used to detect the tokens. Tokens are individual linguistic units which are obtained once we tokenize text. This task is performed with 4 different libraries namely NLTK, Spacy, TextBlob and Nagisa. The ideal outcome of this task should detect all tokens on the full text in Wiki pages. The detected tokens should be stored along with its begin index and end indexin the turtle format as per DBpedia norms. The output files of tokenization task are stored on the directory "Tokens/". The triples of all tokens in a Wiki page are stored on one output file.

The function to tokenize a string or a sentence is as follows:

```
Function spans(text):
    Split tokens of input text through different libraries
    Initialize offset as 0
    Loop over every token on the list:
        Store offset with the begin index of token
        Yield token, offset, offset+length(token)
        Increment offset by length of token
```

Listing 3.11: Function to perform tokenization

The function "spans" takes every sentence of the Wiki page as an input. Tokenization is performed through various libraries on the sentences received as an input parameter to the function. The pseudocode is same for implementation of tokenization through different libraries, only the specific commands to split tokens vary depending upon library used. The

source code of the Python scripts for implementing tokenization are added as a part of the CD and refer section 3.2 for their snippets. The result of tokenization applied over the given sentence is stored as a list with each index containing a token saved in the same order as they appear on the sentence.

The "offset" variable is used to keep track of the begin index of tokens. The index of any given sentence starts from 0, so offset is assigned as 0 initially. Loop over every token present in the list, that is obtained as a result of performing tokenization on an input sentence. On each iteration of the loop, a find operation is used to search the token being iterated on the input sentence. Every token being iterated will be found on the sentence and their begin index with respect to the sentence will be retrieved. The find operation in Python is declared as follows:

```
1 Syntax: <full-text>.find(<word-to-search-on-full-text>,<begin-index-
    to-start-search>)
2 text.find(token, offset)
```

The variable "token" is to be searched on the given "text" from the index corresponding to "offset". The search of a token in a sentence begins from the index that "offset" is currently referring to. This helps to retrieve the correct begin index of "token" on a sentence even if the "token" appears multiple times on the same sentence. The result of the index found is stored as "offset" in order to track the token currently being iterated. Since all the tokens in a sentence are going to be iterated in the order as they appear in a sentence, it is ok to assign the result of the find operation to "offset" again.

"Yield" is a keyword in Python that is used similar to return, except the function will return a generator. A generator function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. This situation demands yield over return since it sends sequence of values to its caller. The function 'spans' returns the token, its begin index and its end index to its caller. It returns these parameters for all the tokens in the sentence separately in each iteration of the loop. End index is calculated by adding the length of the token to the begin index of the token. The offset is then incremented by the length of the token. This allows retrieval of correct index of every token even if a particular token appears multiple times within a given sentence.

The loop finishes after all the tokens in a given sentence are iterated. From line 6, it is evident that it returns token, offset (begin index) and offset+ length of the token (end index).
The function is integrated in the main program as illustrated in the following listing:

```
1  Get the language and instance size as input
2  Initialize count as 0
3  Loop over files one by one in "Input<lang_name>/" directory
4    If count is less than the instance size
5      Create graph through RDFLIB
6      Parse the graph with file as a parameter
7      Serialize the graph with the parsed file
8      Store the name of the file
9      Loop over every sub, pred and obj on serialized graph
10       If type of obj is Literal and pred is nif:isString
11         Split sentences on the obj through different libraries
12         Loop over every sentence on list of sentences
13           Store the begin index of sentence
14           Loop over every token in spans(sentence) function
15             Assert condition checking if token is correct
16                 Calculate Begin index of every token
17                 Calculate End index of every token
18                 if token is not a punctuation
19                   Write BI, EI and tokens onto output file
20     Increment count
21   else
22     Break
```

Listing 3.12: Pseudocode for tokenization

Explanation about the pseudocode of the Python script in my tool for performing Tokenization:
1) The user has to run my tool by executing the Shell script with arguments namely NLP task, library, instance size and language. Instance size corresponds to the number of files from "Input<language-name>/" directory, on which tokenization must be performed. The Shell script passes the value of these arguments to "main.py". This central Python script manages all the other Python scripts calling appropriate Python script based on the library and task chosen.

2) The library "string" is used for finding if there are tokens detected only with punctuations. Apart from the common libraries namely "os" and "sys", each of the four different implementations requires its own library such as NLTK, Spacy, TextBlob, Nagisa for carrying out the tokenization task.

3) The implementation follows the same procedure as sentence splitting for the first 12 lines of the pseudocode. The begin index of the sentence has to be found out before calling the "spans" function. In order to calculate the correct index of tokens wrt to a Wiki page, we need to add the begin index returned by spans function and the begin index of the sentence. Every sentence is given as an input to the spans function. The iteration of every

token in a sentence is coded as follows in Python:

```
1 for token in spans(sentences[i]):
2         assert token[0]==sentences[i][token[1]:token[2]]
```

The "for" loop iterates over every token it receives from the spans function. If a sentence has 10 words, then this loop runs 10 times and the "spans" function returns 10 sets of tokens, begin index and end index. For instance (book,9,12) is one of the sets returned by spans function where the token is "book" (token[0]) with begin index "9" (token[1]) and end index "12" (token[2]).

4) The "assert" keyword helps detect problems earlier in a program rather than finding it later as a side-effect of another operation. "Assert" checks if the token returned by spans function exist on the returned begin index to the returned end index of a sentence. It is a better practice to insert the assert statement inside try-except block. If the assert condition fails, it just passes to next token without any interruption. The index of the tokens in a Wiki page needs to be recalculated as the begin and end index obtained from spans function is confined to the particular sentence. In order to calculate the index of token with respect to the entire Wiki article, adding the begin index of sentence to the begin index returned by spans function is necessary, to get the correct value. Same is applicable for end index as well.

5) "string.punctuation" includes string of ASCII characters which are considered punctuation characters in the C locales. In line 18 of the listing, the "if" condition checks whether a punctuation exist as an entire token returned by spans function. If so, it is prevented from writing onto the final output file as it isn't a valid token. Some libraries don't consider punctuation as tokens while some do. This condition is mainly to address the libraries that consider punctuation as tokens. If the condition is satisfied, write the begin index,end index and content of the tokens as shown:

```
1 g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
      =2016-10&nif=word_offset_"+str(BI)+"_"+str(EI)), RDF.type, nif.
      Word])
2 g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
      =2016-10&nif=word_offset_"+str(BI)+"_"+str(EI)), nif.beginIndex,
      rdflib.term.Literal(str(BI))])
3 g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
      =2016-10&nif=word_offset_"+str(BI)+"_"+str(EI)), nif.endIndex,
      rdflib.term.Literal(str(EI))])
4 g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
      =2016-10&nif=word_offset_"+str(BI)+"_"+str(EI)), nif.anchorOf,
      rdflib.term.Literal(token[0])])
```

```
5 g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
      =2016-10&nif=word_offset_"+str(BI)+"_"+str(EI)), nif.
      referenceContext, rdflib.term.URIRef("http://dbpedia.org/
      resource/"+name+"?dbpv=2016-10&nif=context")])
```

Listing 3.13: Triples added to output file for tokenization

The triples added to the output file should have the following information regarding tokens in order to store it in a structured manner according to Dbpedia norms:
1) The token in this case is a word. If there are multiple tokens having more than one word, it is considered to be a phrase.
2) Store the begin index of the token.
3) Store the end index of the token.
4) Store the content of the token ie. word present in the token. Should have the predicate as "nif:anchorOf".
5) The reference to this extracted data is NIF context dataset.

A sample taken of the output file taken from "Tokens/Animalia_(book)":

```
1 <http://dbpedia.org/resource/Animalia_(book)?dbpv=2016-10&nif=
      word_offset_2281_2284> a nif:Word ;
2       nif:anchorOf "Big" ;
3       nif:beginIndex "2281" ;
4       nif:endIndex "2284" ;
5       nif:referenceContext <http://dbpedia.org/resource/Animalia_(book
            )dbpv=2016-10&nif=context> .
```

Listing 3.14: Tokenization result

### 3.5.1 Challenges

I have identified several challenges related to tokenization:

1) **Separation of stop words** such as full stops, comma, open brackets and few other punctuations from the sentences. If a text "Base." is given as input to the spans function, then it detects "Base" as a token and ignores ".". Most of the NLP libraries return this result. Therefore, this function basically separates punctuation if it appears at the end or beginning of a token. Here, however "." isn't a token. In case if any library detects "." as a token, while writing onto output file, tokens are tested if they contain only punctuation. If so, this triple is not written onto the output file. It is necessary to keep note of the begin and end index of punctuation as well in "spans" function for incrementing the "offset" value.

2) The biggest challenge is when the **punctuations appear in the middle of the token**. It is really hard to deal with. Examples such as "twenty-six", "iphone/ipod" are considered as a single token according

to the nltk word tokenizer. By using Spacy we could eliminate the "/" in between the words and consider them as two different tokens. Also "children's" would be considered as single token in Spacy. But NLTK detects children as one token and "'s" as another.

3) **Reading files from "Input<lang_name>/" directory** - All files within the "Input<lang_name>/" directory has to be read in order to perform tokenization on them. The following piece of code is used in Python for reading all the files from a specific directory:

```
import os
for filename in os.listdir('Files/Input'+<language>+'/'):
    name=filename.split(".")[0]
    <code for tokenization>
    <store the results in Tokenization folder>
```

Listing 3.15: Reading all files on a directory

Importing "os" helps to get access to the contents of each and every individual file present inside the home directory of the tool. The variable "filename" has the name of every file inside the Input directory along with their extensions, so in order to strip out the extension I have used a split function and retrieved the part before the full stop. The entire tokenization source code appears within this loop.

4) **Performing Tokenization on 5 different languages** - Spacy supports tokenization on English, Spanish, French and German. Loading appropriate Spacy language model is sufficient for using it to tokenize contents of Wiki articles in these languages. TextBlob and NLTK supports tokenization for these languages as well. It is possible to be implemented by specifying the language within the parameter, inside tokenize function:

```
nltk.word_tokenize(str(txt),language=''+assign+'')
```

However, for accuracy purposes NLTK is used along with StanfordPOSTagger for languages other than English to detect tokens. Tokenization in Japanese is achieved by using Nagisa library.

## 3.6 Part of Speech Tagging

This section explains the third NLP task ie. part of speech tagging for each or a subset of the files generated from the separation of Wikipedia articles step. The goal is to find the part of speech for every token present in the full text of a Wiki page. This step serves to be important for performing the links enhancement task. Let's say a word "work" appears multiple times in different contexts. For eg. "Martin was tired after a day's work" and

"Martin was working on field theory", in both these cases the word "work" points to different Wikipedia articles. In the first case, "work" is a noun and in the second case "work" is a verb. Depending on the part of the speech, the new link assigned will be differed. This task is performed with 5 different libraries namely NLTK, StanfordPOSTagger, Nagisa, Spacy and TextBlob. Each library has a corresponding python script created on my tool. The result of each tool is expected to be quite different from each other.

POS tagger [32] is a piece of software that reads text and assigns parts of speech to all tokens or words it detects, such as noun, verb, adjective, pronoun, adverb etc. The output from the POS tagger of a token appears in abbreviations such as "NNP" or "PRP". Here, NNP signifies Proper Noun and PRP indicates Personal Pronoun. So, mapping the output of POS taggers to their corresponding expansions is required as it is hard to interpret that PRP is a Personal Pronoun. So, it is necessary to map all possible results of POS taggers to their respective human readable parts of speech. An efficient and quicker way to implement this mapping is by using a DataFrame. Pandas DataFrame is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labelled axes (rows and columns). In our case, the DataFrame will contain 3 attributes :

- The first column contains all possible abbreviations obtained as output from the library's POS taggers such as NNP, PRP etc. The entire list is gathered from the official NLTK documentation [33].
- The second column(SF) consists short form expansion of abbreviation on first column such as Noun, Pronoun.
- The third column(FF) contains the full form expansion of abbreviation such as Proper Noun, Personal Pronoun.

The implementation of the DataFrame in Python is done as follows:

```
data = [
['NN','Noun','Noun'],
['VB',  'Verb', 'Verb'],
['DT','DET','Determiner'],
['JJS', 'ADJ',  'Adjective']
]
df = pd.DataFrame(data, columns = ['POS','SF','FF'])
```

Listing 3.16: DataFrame for mapping POS

The variable 'data' stores the list of all possible POS tagger's results, their short form expansions and their full form expansions in a list. The values are hard-coded and there are 32 sets overall. The DataFrame 'df' is created

by loading 'data' as the contents of the DataFrame with the column names defined using 'columns' parameter. The DataFrame is used to obtain every token's short and full form expansion of parts of speech by searching the result of POS tagger on it. Sample data saved on the DataFrame :

| POS | SF | FF |
|---|---|---|
| NNP | Noun | ProperNoun |
| JJ | ADJ | Adjective |
| VBZ | Verb | VerbSingularPresent |
| PRP | Pronoun | PersonalPronoun |
| NNS | Noun | PluralNoun |
| VBD | Verb | VerbPastTense |

Table 3.1: Sample data on DataFrame

The ideal solution of this NLP task must provide the part of speech for every token. The output file should contain content, begin index, end index and the part of speech for every token stored in turtle format. Function to detect the part of speech :

```
Function Postags(text):
    Find (Token,POS) pair for the text through NLP library
    Store all the (Token,POS) pairs on a list for given text
    Initialize offset as 0
    Loop over every (Token,POS) pair on the list:
        Store offset as the begin index of token
        Return token, begin index, end index, POS
        Increment offset by length of token
```

Listing 3.17: Function to detect POS

The "Postags" function returns the content of the token, its begin index, its end index and its part of speech. The result of POS taggers using different NLP libraries is a list of (Token,POS) pairs. The first step in the implementation is to find and store such pairs. This pseudocode is very similar to the "spans" function for tokenization. Only difference is that instead of just finding the tokens of the text, "Postags" finds its corresponding parts of speech on the same step. It is not recommended to find the tokens first and then find the token's part of speech later because POS has to be found from a sentence's perspective and not from a token's perspective. The POS of tokens differs based on the context they appear. The input to this function is either a sentence or a paragraph. The offset is used to keep track of the token and provide the correct indexes of the tokens even if a token appears multiple times inside the given text.

Every token is iterated one by one inside the "Postags" function so that the results are returned one by one to the main program. The "yield" keyword has been used for passing the results of the function to the main program. This situation demands yield too as it sends sequence of values to its caller. The offset is incremented at the end to let know that the next token is ready to be processed. The functioning of Postags function for a sample text "Over three million copies have been sold." is as follows. Various libraries are used to find parts of speech for every token in the given text. The result of finding (Token,POS) pair for the sample text is stored as follows : [('Over', 'IN'), ('three', 'CD'), ('million', 'CD'), ('copies', 'NNS'), ('have', 'VBP'), ('been', 'VBN'), ('sold', 'VBN')].
Let's say that name of the variable that has (Token,POS) pairs stored is called 'tagged'.
Then tagged[0] is ('Over', 'IN') and tagged[1] is ('three', 'CD') and so on.
And tagged[0][0] should have 'Over' whereas tagged[0][1] should contain 'IN'.
The function is integrated to the main program as illustrated in the following listing:

```
1  Loop over every sub, pred and obj on serialized graph
2    If type of obj is Literal and pred is nif:isString
3      Split sentences on the full text via libraries
4      Loop over every sentence on list of sentences
5        Store the begin index of sentence
6        Loop over every token in postags(sentence) function
7          Assert condition checking if token exists
8          Calculate Begin index of every token
9          Calculate End index of every token
10         Store the POS obtained from Postags function
11         Get the full form of POS from DataFrame
12         if token is not a punctuation
13           Write the triples onto output file
```

Listing 3.18: Pseudocode for POS

Explanation about the pseudocode of the Python script in my tool for performing part of speech tagging:
1) The user has to run the developed tool by executing the shell script with NLP task argument chosen as POS. The shell script passes this value to central python script which would in turn call the script whose pseudocode is mentioned above in the listing. Libraries "pandas" and "numpy" are used for implementing DataFrame.

2) The implementation of POS follows the same procedure as tokenization for the first 9 lines of the pseudocode in the listing. Every sentence is given as an input to the 'Postags' function. If a sentence has 10 words, then this loop runs 10 times and the 'Postags' function returns 10 sets of

tokens, begin index, end index and POS. For instance (Prague,9,14,NNP) is one of the sets returned by the function to the main program. In this set, token is "Prague" (token[0]) with begin index "9" (token[1]), end index "14" (token[2]) and part of speech is "NNP" (token[3]). NNP indicates Proper Noun.

3) The next step is to search for the token's POS obtained from the 'Postags' function on the DataFrame to get its expansion. The value of 'token[3]' is searched against the attribute called 'POS' on the DataFrame. The search will always be successful because the DataFrame consists of all possible results from POS tagger. The entire row from the DataFrame is extracted on which the search turns out to be successful. The short form and full form are obtained as follows from the DataFrame:

```
value=df['SF'][df['POS']==token[3]]
for val in value:
        short_form="http://purl.org/olia/olia.owl#"+val
fullvalue=df['FF'][df['POS']==token[3]]
for jval in  fullvalue:
        full_form="http://purl.org/olia/olia.owl#"+jval
```

Listing 3.19: Integration of DataFrame to the main program

The variable 'value' picks up the short form expansion from the row which has the same POS as that of the one returned from the 'Postags' function for every token. The values of attribute 'POS' on the Dataframe are searched one by one to see if there is a match for the result from the POS Tagger. The row that has matching value is extracted, it's corresponding short form expansion is received from the attribute 'SF' and the full form expansion is received from the attribute 'FF'. Their values are recorded on variables 'value' and 'fullvalue' respectively in a list. The list is iterated inside a loop to store the result on variable 'short_form' and 'full_form' along with URL link for storing on the output file which suits the DBpedia norms. Even though the list has only one value, it has to be iterated inside a loop to get the value.

4) As a part of the URL link 'olia' is used to represent the POS. The Ontologies of Linguistic Annotations (OLiA) provide an OWL taxonomy of data categories as a reference for linguistic annotation (OLiA Reference Model), plus OWL models for a large number of annotation schemes (OLiA Annotation Models) and their relationship to reference data categories (OLiA Linking Models). The OLiA ontologies were originally developed in the context of an infrastructure for the sustainable maintenance of linguistic resources, and they have been applied for the formalization of annotation schemes, concept-based querying over heterogeneously annotated corpora, the development of interoperable NLP pipelines, and as a central hub for

annotation terminology in the Linguistic Linked Open Data (LLOD) cloud.

5) The 'string.punctuation' prevents the tokens from writing into the final output file that just has punctuation,. Some libraries don't consider these punctuation as tokens while some do. The output file should contain all the triples as that of the tokenization task and the following triples wrt. part of speech in addition :

```
g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
    =2016-10&nif=word_offset_"+str(BI)+"_"+str(EI)),nif.oliaLink,
    rdflib.term.URIRef("http://purl.org/olia/penn.owl#"+token[3])])
g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
    =2016-10&nif=word_offset_"+str(BI)+"_"+str(EI)),nif.oliaCategory
    ,term.URIRef(short_form)])
g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
    =2016-10&nif=word_offset_"+str(BI)+"_"+str(EI)),nif.oliaCategory
    ,term.URIRef(full_form)])
```

Listing 3.20: Triples added to output file for POS

The triples added to the output file should contain the following information to maintain a structured manner, according to Dbpedia norms:
1) The following set of triples correspond to a token which is a word.
2) Store the begin index of the token.
3) Store the end index of the token.
4) The content of the token. Should have the predicate as "nif:anchorOf".
5) The reference of where this data is extracted from ie. NIF context dataset.
6) Store the result obtained from the POS Tagger for the token.
7) The short form expansion of the POS for the token.
8) The full form expansion of the POS for the token.
All the output files are stored on "POS/" directory. A sample taken from the output file of "POS/Agricultural_science" is as follows:

```
<http://dbpedia.org/resource/Agricultural_science?dbpv=2016-10&nif=
    word_offset_10007_10016> a nif:Word ;
    nif:anchorOf "Pathology" ;
    nif:beginIndex "10007" ;
    nif:endIndex "10016" ;
    nif:oliaCategory <http://purl.org/olia/olia.owl#Noun>,
        <http://purl.org/olia/olia.owl#ProperNoun> ;
    nif:oliaLink <http://purl.org/olia/penn.owl#NNP> ;
    nif:referenceContext <http://dbpedia.org/resource/
        Agricultural_science?dbpv=2016-10&nif=context> .
```

Listing 3.21: Part of speech tagging result

All the triples have the same subject. The triples added to the output file informs the following:

1) Lines (1-4) are same as the output from the Tokenization task result-set.
2) Line 5 informing the short form expansion of the part of speech for the token as "Noun" (at the end of URL) under the predicate 'nif:OliaCategory' for the token 'Pathology'. The URL '<http://purl.org/olia/olia.owl#>' is added
3) Line 6 informing the full form expansion of POS for the token as "ProperNoun" under the same predicate as 'nif:oliaCategory'. It has the same predicate as the short form expansion and that is why the predicate doesn't appear on Line 6, it just has the object.
4) Line 7 displaying the POS obtained from the result of POS Tagger as an object to the predicate 'nif:oliaLink'.
5) Line 8 gives reference to where the data is obtained from.

### 3.6.1 Challenges

I have identified several challenges related to POS NLP task:

1) Incorrect part of speech obtained while supplying tokens as an input to "postags" function. It is solved by finding the part of speech by using sentence as an input to the function rather than using individual token as an input. If individual tokens are provided as an input to various NLP libraries, the POS taggers returns the POS that has higher confidence value but ignores the context under which the token is used. The confidence value of a token's POS is calculated based on the primary meaning of the token and the primary usage of the token in general. For example "They refuse to permit us to obtain the refuse permit", if I provide every token individually to NLTK library's POS Tagger then both the "refuse" would have verb as their part of speech. However, the correct result should have first refuse as noun and second one as verb. It chooses both as VERB because confidence value of verb is higher for refuse as in most cases refuse is used as verb. The primary usage of refuse is verb over noun. But this is not the correct result in our case. So, this challenge is overcome by providing sentences as an input to the various libraries POS taggers.

2) The part of speech for "." is assigned as "." which is incorrect as punctuations don't have any POS. So, I just prevented these tokens from writing to the final output file. If an entire token is a punctuation, the tool automatically assigns the POS as the same punctuation token by default which is quite strange. Anyway, this step was skipped in the tokenization task as well.

3) Finding the part of speech in different languages. NLTK's accuracy for POS is good only in English language. Thus, StanfordPOSTagger is used to detect POS for German, Spanish and French languages and Nagisa

explicitly for Japanese. spaCy and TextBlob do support wide variety of languages for detecting POS. StanfordPOSTagger provides fast POS Tagger for German and it has the highest precision rate for German language.

4) Tracking the begin and end index of each token. If a particular word/-token appears more than once in a particular sentence that is given as input to 'Postags' function, then this has to be dealt carefully to assign correct begin index to the token. The variable 'offset' is maintained explicitly to deal with this problem which is incremented at the end of each token inside the loop. This variable is supplied to the find operation as start index, from where the operation has to start its search on the input sentence. So if the same token appears the second time on a given sentence, then the index of second appearance of the token is searched to get the correct begin index.

## 3.7 Links Enhancement

Surface form analysis is a pre-processing step for enhancing additional links. NIF text links dataset is required for generating surface forms CSV file. After downloading and extracting the NIF text links datasets in English (nif_text_links_en.ttl), French (nif_text_links_fr.ttl),
German (nif_text_links_de.ttl), Spanish (nif_text_links_es.ttl) and Japanese (nif_text_links_ja.ttl) languages, the goal is to generate the list of all words having link(s) to other DBpedia resource(s). The list of such generated surface forms are stored on a CSV file. This process has to be done for all 5 languages ie. on all the 5 versions NIF text links files. The objective is to extract the details of surface forms from these huge files and store them in a format that supports quick search operation. The generated CSV file must contain the following attributes :

- Surface Form - The list of all the words that has link to other DBpedia resource(s). All in-text links to other DBpedia resources as well as external references. In Wikipedia, the words that holds a link on them which when clicked redirects the browser to a different Wiki page. Such words that has a link is called as Surface Form. The first column should have the token(s) of these surface forms.

- Link - The second column should consist the link to DBpedia resource which should load the contents of new DBpedia resource. It should posses the URL link to load the new DBpedia resource.

- Part of Speech - The third column should have the part(s) of speech for the token(s) of the surface form. If the surface form comprises multiple tokens, each token should have part of speech.

There are 5 CSV files generated, one for each language. This is a pre-processing step for "Links Enhancement" NLP task. The naming convention of these CSV files are "LinkDataset<language>_with_duplicates.csv" as the generated output file will have duplicate values. It is problematic for performing search operation on a TTL file as opposed to a CSV file which explains why we need to store the result in CSV. The pseudocode for generation of links dataset is as follows:

```
Get the path to NIF Text Links file
Store the language of NIF Text Links file provided
Define the columns of CSV file
Create and Open the CSV file on 'write' mode
Open NIF Text Links file from the given path
  Write the column names onto CSV file
  Loop with line from 1 to end of file
    Check if "nif-core#anchorOf" is part of the line
      Store the name of the surface form
      Store the link to the DBpedia resource for surface form
      Replace empty spaces by '_' on surface form
      Store the tokens on the surface form
      Store the POS of the tokens
      Declare an empty list
      Loop over every token on surface form
        Add the POS of token to the empty list
      Write Surface Form, Link, POS to CSV file
```

Listing 3.22: Pseudocode for generating links dataset

Explanation about the pseudocode of the Python script on my tool for generating the surface form dataset:
1) The user has to run my tool by specifying the path to the location of NIF text link file as an argument. The shell script validates the path to NIF text links file. If the validation is successful, the shell script would then pass the location of NIF text links file to the python script whose pseudocode is mentioned above in the listing.

2) The language of the NIF text links file is mentioned as a part of its name - "nif_text_links_<language>.ttl", similar to that of NIF context file. Storing the language is necessary for the naming the generated CSV file. The columns of the CSV file are surface form, link and POS. A new CSV file is created as "LinkDataset<lang>_with_duplicates.csv" inside the "Files/" directory. This CSV file is opened on write mode with encoding style "UTF-8". Write the defined column names into created CSV file.

3) Iterate over each line present in the NIF text links file. There are 7 triples for each surface form in the text links file. The triple that contains the tokens of the surface form has the predicate "nif-core#anchorOf". The

object of the triple consists the tokens of the surface form. The link to the DBpedia resource is obtained from the object of the triple with predicate "taIdentRef". Store the tokens present on the surface form and the link to its DBpedia resource. Create an empty list to append the POS of these tokens into the list. Finally, write every set of Surface Form - Link - POS into the CSV file. A sample output taken from "Files/LinkDataseten_with_duplicates.csv" file:

| Surface Form | Link | POS |
|---|---|---|
| Graeme Base | https://dbpedia.org/resource/Graeme_Base | NNP NNP |
| Synopsis | https://dbpedia.org/resource/Synopsis | NN |
| alliterative | https://dbpedia.org/resource/Alliterative | JJ |
| alphabet | https://dbpedia.org/resource/Alphabet | NN |
| alligator | https://dbpedia.org/resource/Alligator | NN |
| butterfly | https://dbpedia.org/resource/Butterfly | NN |

Table 3.2: Sample data from generated CSV file

However, this CSV file consists of duplicate records. In order to remove duplicity, another script is created to delete the duplicate records on the CSV file. The pseudocode for removing the duplicate records on the surface forms CSV file generated from NIF text links dataset is as follows:

```
Create a new CSV file on write mode with 'UTF-8' style
Open the CSV file with duplicate records on read mode
    seen = set() //set for fast O(1) amortized lookup
    Loop for every record on file with duplicate records
        if record is seen
          continue //skip duplicate record
        Set record as seen
        Write the record into new CSV file
```

Listing 3.23: Pseudocode for removing duplicate records

This creates a new CSV file and stores unique records within it by skipping the already seen records. Open the CSV file with duplicate records on read mode and open the newly created empty CSV file on write mode. Loop over every record on the CSV file with duplicate records, and set the record as seen for the records appearing first time on the file. Immediately after making it seen, write this record onto the new CSV file. If an already seen record appears again during the future iterations of the loop, then this record is skipped from writing onto the output file. Naming convention of the CSV file without duplicates is "LinkDataset<language>.csv". For instance, French language surface forms are stored with the name "Files/LinkDatasetfr.csv". Removing the duplicate records helps the size of the CSV file reduce drastically and therefore the search operation could be

done faster. It is not necessary to generate all the surface forms from NIF text links dataset to the CSV file. The execution of "separate_scripts.sh" for generating the surface forms CSV file could be terminated at anytime after the execution starts. The longer it runs, more records are generated on surface forms CSV file. Now that the pre-processing step is completed, next step is to enhance the links.

```
Initialize counter as 0
Loop over every sub, pred and obj on the input file
  If type of obj is Literal and pred is nif:isString
    Tokenize the obj(full text) through NLP library
    Store the (Tokens,POS) pair
    store the count of the total tokens in the input file
    Loop with i less than count of total tokens
      Initialise flag2 and flag as 0
      store the token[i]
      if the token[i] is neither a punctuation nor a stopword
        Store the combination of token[i], token[i+1] & token[i+2]
        Store the combination of token[i] & token[i+1]
        Open the generated CSV file with removed duplicates
          Read the records on CSV
          Check if token[i] and POS[i] matches a record on CSV file
            Check if combination of 3 tokens matches an entry on CSV
              Find begin index of these 3 tokens on full text
              Provide link to these 3 tokens obtained from CSV
              Check if begin index is greater than counter
                Write the triples onto output file
              Increment counter by length of token[i+1] & token[i+2]
              Increment i by 2
              Break
            Check if combination of 2 tokens matches entry on CSV
              Store the begin index and link of tokens
              flag2=1
            Check if token[i] matches the record on CSV
              Store the begin index and link of token
              flag=1
        Check if flag2=1 and begin index of tokens > counter
          Provide stored link to these 2 tokens obtained from CSV
          Write the triples onto output file
          Increment counter by length of token[i+1]
          Increment i by 1
        Check if flag2=0 and flag=1 and begin index > counter
          Provide stored link to the token obtained from CSV
          Write onto output file
        Increment counter by length token[i]
      Increment 'i' and counter by 1
```

Listing 3.24: Pseudocode for link enhancement

The final part of my thesis is enhancing links on NIF context dataset. The goal is to enhance the amount of links on each or a subset of output files generated from "separation of Wiki articles" task. The pseudocode is presented in the listing as follows:

Linking Wikipedia articles to each other is important. The links allows users to move to a new article from the current Wiki article helping to understand the original article in depth, greatly adding to Wikipedia's usefulness. But at the same time adding too many links can be distracting. This could however be avoided by providing link to the same word again and again within one article. The list of all surface forms already existing are available in the CSV file generated from the previous step.

Read the full text of Wiki pages token by token in order to detect the tokens that are surface forms. If the content and part of speech of read token(s) matches the content and part of speech of a surface form entry on the CSV file, then a link is provided to these read tokens(s) for their respective DBpedia resource.

The input files of this task are the files present inside "Input<language-name>/" directory. These files are read one by one as input to this task.
Explanation about the pseudocode of the Python script in my tool for performing Link enhancement NLP task:
1) The "counter" variable is initialised as "0" and it's main usage is to prevent giving links to same word more than once on a particular Wiki article. The implementation is followed by a loop over all the triples present in the input file. Store the tokens and POS pairs for the full text of the Wiki page. Loop with variable 'i' from 1 to total number of tokens present in the full text of this input Wiki article. Inside the loop store the content of token that is being iterated and check if it is a punctuation or a part of stop words. A stop word is a commonly used word such as 'a','the','an','in','would' etc. that a search engine has been programmed to ignore, both when indexing entries for searching and when retrieving them as the result of a search query. The goal of checking if the token is a part of stop word is to prevent assigning links to such words. Well, if the token is either a punctuation or a stop word, then increment the counter by length of the token and move to the next token.

2) My tool should be able to provide links for surface forms containing upto 3 tokens. So, the combination of the current token being iterated along with the next two upcoming tokens are stored on a variable. Similarly, the combination of current token being iterated along with the next upcoming token is stored on another variable.

3) Open and load the appropriate CSV file with duplicates removed, based on the input language provided by the user. If token currently being iterated matches the first word of an entry of surface form column on CSV file,

then their corresponding parts of speech are compared. If POS are same, then my tool compares if this record on CSV file matches with the combination of first 3 tokens along with their parts of speech. If so, link is provided to all 3 tokens and the triples are written onto output file. 'i' incremented by 2 to prevent iterating the next two tokens as they are already provided with the link. Counter is incremented by the length of next two tokens. Break statement is executed and the control of the program moves outside the loop which reads records of CSV file. The length of the current token is incremented to counter, before iterating the third token from the current one.

4) Before writing the surface form provided with a link onto the output file, the begin index of the surface form is checked if it is greater than counter. If yes then the link is provided to the three tokens and written onto the output file. If begin index of the token is lesser than the counter then the surface form is not provided with a link because it means that this particular surface form has already appeared on the full text of this Wiki page. My tool would have provided a link on its first occurrence as the begin index of the surface form would have been greater than counter. If begin index of the token is lesser than the counter, then it prevents giving link to the same surface form again within the same Wiki article.

5) If the combination of first 3 tokens isn't a match on this record of CSV file, then it attempts to search for the combination of first 2 tokens and its POS on the same record of CSV file. If there is a match between the 2 tokens and their POS on CSV file, then the variable "flag2" is set as 1. The break statement is not executed here because there is a possibility that another record on CSV file could have a match for 3 tokens. The goal is to provide link to the longest surface form. So, the variable 'flag2' is set to 1 and the begin index and link are stored before reading the the next record on the CSV file. The point of setting "flag2" as 1 is to remember that there is a match for 2 tokens which is eligible to be given a link. At the end of searching the whole CSV file, if there isn't a match for the combination of 3 tokens on CSV, then the link is provided to the 2 tokens from the "if" condition on line 29 of the listing. The begin index is checked if it is greater than counter before writing onto. 'i' incremented by 1 to prevent iterating the next token as it is already provided a link. Counter is set to the begin index of tokens and incremented by the length of last token ie. token[i+1]. Before moving onto the second token from the current one increment counter by length of token[i] and increment 'i' by 1.

6) If the combination of first 2 tokens isn't a match on the record of CSV file, then checks if one token matches the whole surface form and its POS on CSV. If so, stores the link and begin index of the token. The variable "flag" is set to one. And the search operation continues on the next records

of the CSV file. After reading the whole CSV file and if it doesn't consists of links to either combination of 2 tokens or combination of 3 tokens, then the "if" condition on line 34 kicks in. A link is provided to one token after checking if its begin index is greater than the "counter". This surface form details are noted down on the output file. The counter is incremented by the length of the token. 'i' incremented by 1 to make sure next token is being iterated. The next token is searched for a possibility to be provided a link. This process continues until all the tokens in the input file are iterated.

The necessary parameters to be stored on the output file are begin index of surface form, end index of surface form, token(s) of surface form, DBpedia resource link to the surface form. A sample taken from the output file of enhancing links task on "Actrius" article:

```
1 <http://dbpedia.org/resource/Actrius?dbpv=2016-10&nif=
      phrase_offset_100_112> a nif:Phrase ;
2       nif:anchorOf "Ventura Pons" ;
3       nif:beginIndex "100" ;
4       nif:endIndex "112" ;
5       nif:referenceContext <http://dbpedia.org/resource/Actrius?dbpv
            =2016-10&nif=context> ;
6       itsrdf:taIdentRef dbp:Ventura_Pons .
```

Listing 3.25: Links Enhancement result

The triples explained (with each point corresponding to the line number):
1) Surface form is a phrase ie. comprising more than one token. If it had only one token, then the object appears as "nif:Word".
2) The word that has the link on it is "Ventura Pons". The tokens on surface form appear as "Ventura Pons".
3) Begin index of surface form in Actrius article is 100.
4) End index of "Ventura Pons"(surface form) in Actrius article is 112.
5) This content is part of the context file.
6) The DBpedia resource for surface form is "dbp:Ventura Pons". The prefix dbp points to "<http://dbpedia.org/resource/>".

### 3.7.1 Challenges

Repetition of links to the same surface forms in the article. If a surface form repeats 5 times within an article, my tool would provide link to the first occurrence of this surface form. This reduces unnecessary number of links. It is achieved by using "counter" variable. This variable keeps track of the begin index of the token being iterated currently. If the begin index of the surface form on full text of Wiki article is lower than the index of the token currently being iterated, then this surface form had past its

first occurrence of its tokens. That’s why it can be safely ignored that this surface is already provided with links.

Detecting surface forms that has more than one token. My tool can provide links for surface forms upto 3 tokens. If there is a possible surface form having more than 3 tokens, my tool doesn’t detect it. However it is possible to increase this number by checking it token[i+4] matches the 4th token on CSV and subsequently increasing this number. But the initial version of my tool doesn’t support this now. It will be released in the future versions of my tool.

Incorrect links - For eg: There is a surface form "It was" which is a music album. A sentence "It was a beautiful day." is detected in an article. In this case it should not provide link to the wikipedia page of that music album "It was". The part of speech has to be compared and the link should be prevented in this case. Now, lets compare the same example - ’It_was’ is the name of the album, thus it is assigned "NN NN" as POS whereas the normal english words ’It was" is assigned "PRP VBD" as part Of speech. So the link is prevented from providing for these words. Incorrect links are overcome by comparing the parts of speech.

Providing links for words like "The" , "was" etc.. doesn’t makes sense. A reader will be irritated to see links provided for basic words. The basic words are detected by importing stopwords package on each language. Every language has a set of stopwords which covers all the basic word of the language. These words when appearing on the full text of Wiki pages are skipped from being searched on the generated CSV file. The length of the token is added to counter and the next token is being iterated.
This sums up the Data processing chapter. All the tasks implemented in my thesis are covered.

Chapter 4

# Usage

This chapter provides documentation of the processing steps required for generating the result-set of NLP tasks through the developed tool. Also, this chapter explains integration new language(s) and new library(s) to my tool. All the scripts and necessary directories are stored in my GitHub repository [34]. my tool for enrichment of DBpedia NIF dataset is a compilation of Python3 and shell scripts that enables to perform various Natural Language Processing tasks on Wikipedia on normal off-the-shelf hardware (e.g., a quad-core CPU, 8 GB of main memory, and 512 GB hard disk storage).

## 4.1 Requirements

The requirements to run the developed tool :
Python3
Rdflib >= 4.0 [30]
Numpy >=1.16.3
Pandas >= 1.0
NLTK >= 3.0 [17]
Spacy >=2.0 [18]
TextBlob >=0.15.2 [20]
Pattern >=3.6 [19]
StanfordPOSTagger >= 3.9.0 [21]
Konoha,Nagisa (for Processing Japanese language) [31]

## 4.2 Processing steps

The processing steps are:

- **STEP 1** : Download the NIF context file from the official DBpedia page [9] in the TTL format. Languages supported in this project

are :
English (nif_context_en.ttl),
French (nif_context_fr.ttl),
German (nif_context_de.ttl),
Spanish (nif_context_es.ttl) and
Japanese (nif_context_ja.ttl).
Download the language of NIF context version(s) as per the requirement. Extract them after downloading. A minimal version of these files are created and are stored on "NIF_Dataset_Minimal_Version/" directory located at the home of my GitHub repository [34]. Similarly download NIF Text Links file in TTL format for any of the above mentioned languages. Extract them after downloading. NIF text links file is required only if you would like to perform the 'Links Enhancement' NLP task. The minimal version of the NIF Text Links dataset is stored at "NIF_Dataset_Minimal_Version/" directory.

- **STEP 2** : Clone my git repository [34] on your local system. In order to execute the shell scripts, run the following 4 commands from the home directory:

```
1 sed -i 's/\r//' run.sh
2 sed -i 's/\r//' separate{\_}scripts.sh
3 chmod +x run.sh
4 chmod +x separate{\_}scripts.sh
```

Listing 4.1: Commands to permit execution of Shell scripts

- **STEP 3** : Run "separate_script.sh" with an argument -p specifying the path where NIF context file is stored in the system. The language is automatically detected by the script and the result will be saved in "Files/Input<lang>/" directory.
Positional argument:
-p PATH, Specify the stored location to extracted NIF context file.
Optional argument:
-s SEARCH, Specify the subset of article(s) that needs to be extracted from NIF context file.

**Examples:**
"./separate_scripts.sh -p F:/Master_thesis/nif_context_de.ttl" (Extracts all the articles in German language and stores in Files/Inputde/ directory)
"./separate_scripts.sh -p F:/Master_thesis/nif_context_en.ttl -s St" (Extracts all articles that starts with "St" in English Language and stores in Files/Inputen/ directory)
"./separate_scripts.sh -p F:/Master_thesis/nif_context_es.ttl -s Apocopis" (Extracts the article Apocopis in Spanish Language and stores in Files/Inputes/ directory)

For performing enhancing links NLP task, run the "separate_script.sh" again with an argument -p specifying the path to NIF text links file stored. The result is saved as 'Files/LinkDataset_with_duplicates.csv'.
"./separate_scripts.sh -p F:/Master_thesis/nif_text_links_fr.ttl" (Creates a CSV file with all the surfaceforms-Links-POS for French Language and stores it in Files/LinkDatasetfr_with_duplicates.csv)
This might contain duplicate records. So just to speed up the NLP task, you should run the python script LinkDataset_remove_duplicates.py located at scripts/preprocessing_scripts.
Python scripts/preprocessing_scripts/LinkDataset_remove_duplicates.py (Duplicates are removed and result is stored at Files/LinkDataset.csv)

- **STEP 4** : Perform Sentence-splitting, Tokenisation, Part-of-speech tagging and Enhance Links by running the script "run.sh" with the following argument(s) :
  1) Language - en for english, fr for French, de for German, ja for Japanese, es for spanish. Default language is English.
  2) NLP task - SEN for sentence splitting, TOK for Tokenisation, POS for Part of speech tagging, LINK for enrichment of additional links
  3) Instance size - Number of articles for which the NLP task(s) should be performed. Default integer is 1.
  Search - Specify a particular article name for which the NLP Task(s) has to be performed.
  4) Library name - NLTK for Natural Language Tool Kit package, TTB for using TextBlob , SIO for using SpacyIO and PAT for Pattern. Default is NLTK.

**Usage**
"./run.sh [ -l LANGUAGE] [ -n INSTANCE SIZE] [ -t NLP TASK] [-e TOOL NAME] [-s SEARCH]"

Positional arguments:
-t NLP TASK, Specify SEN, TOK, POS or LINK
-n INSTANCE SIZE , Specify an integer. (Default: 1)

Optional arguments:
-s SEARCH, Specify the name of an article. You have an option to specify "-t ALL" to have all NLP tasks performed for this article.
-e TOOL, Specify NLTK, SIO, TTB or PAT. (Default: NLTK)
-l LANGUAGE, Specify en, de, fr, es or ja. (Default: en)

**Examples:**
"./run.sh -t SEN -n 100" (Performs Sentence splitting on 100 English articles through NLTK)

"./run.sh -t ALL -s Apollos" (Performs all 4 NLP tasks for the article Apollos)
"./run.sh -t TOK -n 100 -l de -e TTB" (Performs Tokenisation on 100 German articles through TextBlob)
"./run.sh -t POS -n 10 -l es -e SIO" (Performs Part-of-Speech tagging for 10 Spanish articles through SpacyIO)
"./run.sh -t LINK -n 10 -l fr -e NLTK" (Enhances Links for 10 French Articles through NLTK)

**Output :**
Results of sentence-splitting task gets stored in "Files/Sentences/" directory in RDF triples.
Results of Tokenization task gets stored in "Files/Tokens/" in RDF triples.
Results of Part of speech tasks gets stored in the Files/POS in RDF triples on the same name as the article.
Results of Link Enrichment task gets stored in "Files/Links" in RDF format.
Results of Search tasks gets stored on "Files/Search" with name of the article followed by task in RDF format.

## 4.3 Integrating new language to the tool

This section describes the steps for integration of new language to my tool. I have used integrating Dutch language to my tool as an example for better explanation of each step.
The steps to be carried for a new language to be integrated with my solution are:
1) Download the NIF context dataset for the new language to be integrated from the official DBpedia download page [9]. For Dutch language, download nif_context_nl.ttl and extract this dataset.

2) Create new empty directory under "Files/" with the name of directory having the format as "Input<language>/". This is the directory on which all the output files of "Separation of Wiki articles" task are stored. For eg. From the home directory of my tool, create an empty directory "Files/Inputnl/".

3) Separate the Wiki articles from NIF context dataset. Execute the "separate_scripts.sh" shell script with the path of NIF context dataset as an argument:
"./separate_scripts.sh -p <Location of NIF dataset>"
For example:
"./separate_scripts.sh -p F:/Master_thesis/nif_context_nl.ttl"
The output gets stored in the "Files/Inputnl/" directory.

4) Perform the various NLP operations . Execute the "run.sh" shell script with necessary parameters. Use the argument "-t" for choosing the NLP task to perform. SpaCy library supports the NLP operations for over 50 languages. Check if the language to be integrated is either supported by spacy [18] or TextBlob [20] or NLTK [17] or CoreNLP [21] from their official site. Together these 4 libraries covers all the languages that are supported by DBpedia. In case, the language is not covered by any of these libraries, then integration of new library supporting this language has to be performed in order integrate the language to my tool.
Examples:
"./run.sh -t SEN -n 100 -l nl -e SIO" (Performs sentence splitting for 100 Dutch articles through SpacyIO)
"./run.sh -t TOK -n 1000 -l nl -e SIO" (Performs Tokenization for 1000 Dutch articles through SpacyIO)
"./run.sh -t POS -n 10 -l nl -e SIO" (Performs Part of Speech Tagging for 10 Dutch article through SpacyIO)
"./run.sh -t ALL -s Apollos -l nl -e SIO" (Performs all 4 NLP tasks for the Dutch article Apollos)

5) However keep in mind that NIF text links dataset has to be downloaded and surface forms CSV file has to be generated for performing the links enhancement NLP task on the language to be integrated. Example:
"./separate_scripts.sh -p F:/Master_thesis/nif_text_links_nl.ttl" (Generates the CSV file containing surface forms - links - POS for Dutch language)

## 4.4 Integrating new library to the tool

Decide the library through which you would like to implement NLP tasks. I would use the NLP library "Gensim" as an example to explain each step. Make sure to read the documentation of the library that you would like to integrate to the tool, from their official site.
The steps to be followed for integrating a new library to my tool:
1) A new library that has to be integrated will have to be installed. This library becomes a requirement to the tool. For importing Gensim, running "pip install gensim" on command-line downloads the Gensim package. Similarly download the library whichever way it suits you.

2) Create Python scripts for performing NLP task(s) through the library to be integrated with the naming convention - "<NLP-Task>_<library-name>.py". These scripts must be created on "scripts/" directory. For performing sentence splitting through Gensim, create "sentence_gensim.py" on "scripts/" directory.

3) Copy the source code of scripts for performing the respective NLP tasks from NLTK or Spacy and paste it on the newly created Python scripts. NLP tasks for NLTK or Spacy are located at "scripts/" directory such as "sentence_nltk.py" or "token_sio.py" or "pos_nltk.py" or "links_sio.py". Choose the scripts to be copied depending upon the NLP task you would like to perform. For sentence splitting through Gensim, copy the contents of "sentence_nltk.py" and paste it on "sentence_gensim.py".

4) The library imported must be changed from nltk to the library integrated on the newly created Python script. That is, change 'import nltk' to 'import gensim' on "sentence_gensim.py" at the top of script. The command used for splitting sentences through Gensim has to be replaced with the command used for performing sentence splitting in NLTK. For eg. the lines which perform sentence splitting through NLTK:

```
1 import nltk
2 sentence_result = nltk.sent_tokenize(o)
```

These lines must be replaced as follows in order to perform sentence splitting through Gensim:

```
1 import gensim
2 from gensim.summarization.textcleaner import split_sentences
3 sentence_result = split_sentences(o)
```

Save the modified scripts. Similarly the commands used for performing other NLP task through NLTK has to be replaced with the commands used for performing the respective NLP task through the integrated library.

5) Update "main.py" located on "scripts/" directory. Importing the newly created scripts on "main.py". For eg. import sentence_gensim at the beginning of "main.py" is sufficient. Add a condition on "main.py" to call these newly added scripts based on the tool and library chosen.

6) Execute the shell script "run.sh" with appropriate parameters to perform various NLP tasks through the newly integrated library.

Chapter 5

# Experimentation

The result-set generated from the tool is evaluated and analysed in this chapter. The results generated from different libraries are compared. The metrics of comparison vary upon the NLP task. Section 5.2 provides the statistics of the result-set.

## 5.1 Evaluation

To determine quality of results, 2 experiments are performed. First is to find the runtime of results obtained through various libraries. Second experiment is to determine the accuracy of results among various libraries.

### 5.1.1 Sentence splitting

This section compares the results of sentence splitting task among various libraries. It was implemented through 5 different libraries namely NLTK, Spacy, TextBlob, Pattern and Konoha. The criteria considered for evaluation are accuracy of the results (in terms of segregating the sentences on Wiki articles) and the runtime for performing this NLP task on Hundred Thousand files. These criteria are applied on all 5 languages focussed on my thesis.
The comparison between the libraries in terms of accuracy of the results of sentence splitting task:
1) Section titles of the Wiki pages - Spacy considers the section titles as a separate sentence in the full text of Wiki articles. Whereas NLTK and TextBlob considers the section title as a part of the first sentence in their respective sections. Pattern doesn't segregate the section titles as a separate sentence. Konoha considers only Japanese sentence boundaries meaning if the section titles don't have sentence boundary, then they are considered to be a part of the first sentence of the section.

2) Contents listed in points - Wiki articles tend to have text listed out in points. Every point don't have full stop towards the end. Spacy detects each of these points as separate sentences. However, the other libraries don't. The other libraries picks up all the points in the listing as one whole sentence.

3) Length of sentences - Pattern adds one additional index to each sentence making its end index one more than what is expected. An example of a sentence detected by Pattern :
"the right to make a plea in mitigation is absolute ."
It adds an unnecessary space between the last token of the sentence and the sentence boundary ie. between the word "absolute" and sentence boundary "." . The full text of Wiki page doesn't have this space. This problem could be overcome by reducing one to the length of each sentence in order to get the correct end index. Spacy adds a couple of indexes to the last sentence in each section. It detects the section boundary and adds two to the length of the sentence.

Hundred thousand files generated from separation of Wiki articles task are considered to compare the runtime of all the libraries. Same files were considered as an input to all the libraries. The table below gives the runtime (in mins) for performing sentence splitting on 100,000 files for each library:

| | English | French | German | Spanish | Japanese |
|---|---|---|---|---|---|
| NLTK | 17 | 22 | 20 | 19 | - |
| Spacy | 28 | 33 | 23 | 20 | - |
| TextBlob | 28 | 24 | 19 | 23 | - |
| Pattern | 20 | 29 | 24 | - | - |
| Konoha | - | - | - | - | 19 |

Table 5.1: Sentence splitting runtime (in mins)

NLTK is the fastest in terms of execution time. It just takes 17mins for NLTK to perform sentence splitting on 100,000 files. For all the 5 million Wiki texts in English language, it takes 5 hours for sentence splitting to be performed. It can be noted that TextBlob is slower for English language but it is quite fast for languages other than English. Spacy is really slow for French language but the accuracy is comparatively good. The reason Spacy takes longer time is because it has to create more triples for each file as it detects more sentences. The section titles are taken as separate sentence in Spacy. NLTK only splits text by sentences, without analysing the semantic structure. Konoha is one of the fastest NLP libraries for Japanese. Pattern has results very similar to that of NLTK but it has an

additional operation to be performed - the end index has to be recalculated as the length of sentences detected by Pattern are 1 more than the correct length. So the time taken is more than the NLTK. My tool helps the user to decide which library to run based on their requirements.

### 5.1.2 Tokenization

This section compares the results of tokenization task among various libraries. It was implemented through 4 different libraries namely NLTK, Spacy, TextBlob and Nagisa. The criteria considered for evaluation are accuracy of the results generated (in terms of tokens generated from full text of Wiki articles) and the runtime for performing this NLP task on 100,000 files. These criteria are applied on all 5 languages focussed on my thesis.
Comparison between the libraries in terms of results from tokenization :
1) White spaces - Spacy detects white spaces in between the tokens that appear on listings. For instance:
'1 Geography
2 Earthsea'
Spacy detects the white space as tokens in between '1' and 'Geography'. Whereas the other libraries don't consider these white spaces as tokens. These white spaces have to be prevented from writing onto the output file for tokenization.

2) Hyphens or slashes in between a word - A word 'well-known' is considered as one token by NLTK, however spacy detects this as 2 different tokens 'well' and 'known'. Spacy ignores '-' as a token. TextBlob detects 'ipod/iphone' as two different tokens and it ignores '/'. Even if '-' or '/' appears at the end of word, TextBlob ignores these characters as a part of the token.

3) Apostrophe or bracket at the beginning or end of a word - NLTK detects these characters as a part of the token. For eg. '(Hello World)' - NLTK detects 2 tokens as '(Hello' and 'World)'. However Spacy detects the tokens as 'Hello' and 'World' and ignores the brackets.

Comparison based on runtime for the Hundred Thousand files:

TextBlob is the fastest in generating the resultset for tokenization task. On the other hand, Spacy is slow for all the languages because it creates more tokens such as additional white spaces (between the tokens inside the listings). NLTK library's runtime is good but it detects special characters as a part of some tokens. For the tokenization task, it is recommended to use TextBlob because of faster runtime and accurate results. Nagisa is quite slow for detecting the tokens. Spanish NIF context file has 1.3million articles compared to 5million articles for English language, so time taken

| | English | French | German | Spanish | Japanese |
|---|---|---|---|---|---|
| NLTK | 38 | 35 | 34 | 34 | - |
| Spacy | 46 | 48 | 47 | 44 | - |
| TextBlob | 33 | 30 | 29 | 28 | - |
| Nagisa | - | - | - | - | 53 |

Table 5.2: Tokenization runtime (in mins)

for performing all NLP tasks on Spanish is considerably lower due to lesser number of articles.

### 5.1.3 Part of speech tagging

This section compares the results of POS tagging task among various libraries. It was implemented through 5 different libraries namely NLTK, Spacy, TextBlob, StanfordPOSTagger and Nagisa. The criteria considered for evaluation are accuracy of the results generated (in terms of parts of speech detected on full text of Wiki articles) and the runtime for performing this NLP task on 100,000 files.
Comparison between the libraries in terms of results from part of speech tagging :
1) Detailed part of speech - NLTK and TextBlob are able to find if a part of speech is singular or plural. For eg. 'islands' is detected as 'NounPlural' by these two libraries, however spacy just detects it as a 'Noun'. Similarly 'has' is detected as 'VerbSingular' by NLTK and TextBlob whereas 'Verb' by Spacy. They are also capable of finding the tense of part of speech. For instance, NLTK and TextBlob detects the token 'derived' as 'VerbPastParticiple' whereas spacy detects it as 'Verb'.

2) Part of speech for white spaces and digits - Spacy assigns 'TABSPACE' as a part of speech for white spaces that are detected in between the tokens on points of listings. There are no part of speech for spaces in NLTK and TextBlob as they don't detect spaces as tokens. Spacy considers part of speech for digits as 'NUMBERS' whereas NLTK and TextBlob detects their part of speech as 'Cardinal Digits'.

3) Difference in part of speech tags - NLTK has way too many tags for part of speech compared to Spacy. For eg. the token 'to' is detected as preposition by Spacy which is the correct POS. NLTK detects the POS for 'to' as 'TO' which is the short form of 'ADVERB when used to go somewhere'. The primary POS for 'to' is preposition but Adverb is the second widely used POS.

NLTK is used along with StanfordPOSTagger for French, German and Spanish languages as the POS tags detected by StanfordPOSTagger is more reliable than NLTK for these languages. NLTK separates the sentences and StanfordPOSTagger detects the tokens and their POS. Comparison between runtime of the libraries:

| | English | French | German | Spanish | Japanese |
|---|---|---|---|---|---|
| NLTK | 57 | 160 | 118 | 143 | - |
| Spacy | 71 | 57 | 50 | 51 | - |
| TextBlob | 54 | 61 | 56 | 58 | - |
| Nagisa | - | - | - | - | 71 |

Table 5.3: Part of speech tagging runtime (in mins)

All the triples appearing on tokenization plus three more additional triples for each token gives the result-set for POS tagging. NLTK and TextBlob doesn't produce good results for POS in languages other than English. StanfordPOSTagger has been used along with NLTK for languages other than English. This is the reason for NLTK to take longer time for execution of French, German and Spanish languages. StanfordPOSTagger is really slow. The time taken for StanfordPOSTagger to perform POS tagging on German (118 mins) is quicker than French (160 mins) or Spanish (143 mins) because it has 'fast tagger' developed for German. But still this is considerably slower than Spacy. NLTK or TextBlob should be preferred for English language and Spacy for French, German and Spanish languages.

### 5.1.4 Links enhancement

The accuracy and runtime vary drastically upon the generated surface forms file 'LinkDataset<lang>.csv'. As mentioned earlier, it is not necessary to generate all the surface forms from NIF text links dataset onto the CSV file. The execution of "separate_scripts.sh" for generating the surface forms file could be terminated at anytime after the execution starts. The time taken for generating all surface forms from English NIF context dataset is about an hour. The longer it runs, more records are generated on surface forms CSV file. More records on the CSV file implies more number of links enhanced and slower the runtime. Fewer records on CSV file leads to lower number of enhanced links and faster the runtime.

One way to evaluate this task is through the F1 score. To measure the accuracy, F1 score is used in the statistical analysis of binary classification. F1 score is defined as the harmonic mean between precision and recall. It is used as a statistical measure to rate performance. F1 score is from 0 to 1 with 0 being lowest and 1 being the highest. It is a mean of an individual's

performance based on two factors i.e. precision and recall. Lets look at the formulae of the various metrics with which this task could be measured:
1) Precision=(TP)/(TP + FP)
2) Recall=(TP)/(TP+ FN)
3) F1=(2*Precision*Recall)/(Precision + Recall)
Here,
TP - Number of True Positives (model correctly predicts the positive class)
FP - Number of False Positives (model incorrectly predicts the positive class) and
FN - Number of False Negatives (model incorrectly predicts the negative class)

I have calculated these metrics for a sample output file generated for Link enhancement task. 'Links/Animalia}(book)-links.ttl' was checked manually. The total number of links originally present in the this article is 22. Out of the 22, 19 are detected correctly by my tool. Rest 3 of the surface forms has more than 3 tokens, my tool in current situation detects surface forms with maximum of 3 tokens. The generated output file from my tool detects 122 links overall as shown:

| | Positive | Negative |
|---|---|---|
| True | 114 | - |
| False | 3 | 5 |

Table 5.4: Evaluation of an output file with enhanced links

My tool detects 100 links more than the already existing links (22). Out of 122, 114 of them have correct links assigned to their respective DBpedia resource. However, 3 surface forms which should have been assigned a link was not assigned by my tool. These 3 links have more than 3 tokens present. My tool is not able to detect the surface forms with more than 3 tokens. However, a plan to increase this threshold upto 10 tokens are considered in the future releases of my tool. And 5 of the links assigned are wrong. These 5 links were assigned twice to the same surface form on full text of the Wiki page. If a surface form appears twice immediately one after the other, then the links are assigned to both the surface forms. This is occurred because of the condition checking if 'begin index > counter' is passed and the output is written onto the output file. In every iteration, the token is read along with its upcoming tokens. So the counter variable doesn't get increased. But the chances for same token to occur either as next token or the token after that is really low. On calculating the F1 score:
Precision = 114/(114+3) =0.974 =97.4%
Recall=114/(114+5) =0.957 =95.7%
F1= (2*0.974 *0.957)/(0.974+0.957) = 0.965 =96.5%

The longer the generated CSV file, longer the runtime of this task. For the completely generated surface forms file, it takes few seconds for enhancing link to the full text of one Wiki page. However, the beauty of the tool is that it allows to generate even half the surface forms from NIF text links dataset and use this to enhance links. This gives a good runtime and a really decent amount of enhanced links. The results of above mentioned 'Links/Animalia}(book)-links.ttl' file used the completely generated surface forms CSV file as an input. The total number of links detected were 96 when a partially generated CSV file was served as an input. The size of the CSV file supplied as an input to the script was half the complete size. The execution time was about twice as fast. And the total links detected is 96 out of overall 114 which is 84%. Even generating half the surface forms CSV file gives good result.

## 5.2 Statistics

The analysis has been performed on 100,000 files that was generated for detecting the runtime on the previous section. These generated files for each language are stored on separate directories as "Sentences/<lang>/". The 100,000 files for each language are stored on their appropriate directories and are used for the analysis. Some important insights figured out:
1) The total number of sentences is 4,380,027 obtained by NLTK whereas 5,091,245 by Spacy for 100,000 English language articles. The average number of sentences in an article is approximately 44 for NLTK and 51 for Spacy. This is expected because Spacy considers the section title as a separate sentence. It was detected by counting the number of 'nif.Sentence' in the generated output files for Sentence splitting task. The pseudocode for finding the overall number of sentences present in 100,000 files is:

```
1 Assign the count to 0
2 Loop with files one by one in "Sentences/<lang>/" directory
3 Parse the graph with file as a parameter
4 Serialize the graph with the parsed file
5         Loop over every sub, pred and obj on serialized graph
6                 if "nif-core#Sentence" in obj
7                         count=count+1
```

Listing 5.1: Total number of sentences

All the TTL files in the given directory are iterated one by one. The graphs are parsed and serialised. The output of "sentence splitting" task has 5 triples created for each sentence - Each sentence in the file has 5 triples with each having type,begin index, end index, content and derived data of a word unit. The triple which has the type of word unit is filtered out in 6th line. The word unit is Sentence in this case.
The 'count' provides the total number of sentences present in 100,000 files.

Average number of sentences in French, German, Spanish and Japanese are 33, 31, 24 and 28 respectively. NLTK was used to calculate the number of sentences for French, German and Spanish languages whereas Konoha was used to calculate the average sentences in Japanese languages

2) The total number of tokens in the 100,000 files are 54,300,206 in English through NLTK which means that every article has an average of 540 tokens. Average number of tokens in French, German and Spanish are 384, 397 and 268 respectively detected through TextBlob. The script used searches for 'nif.Word' in the object and everytime the condition is satisfied, the count of token is increased by 1.

3) POS obtained from various languages are compared to find the occurrence of individual part of speech for each language. Most commonly used part of speech is Noun and it comprises of about 35% of overall tokens in English. The complete table of all the POS are represented below :

| Part of speech | English | French | German | Spanish |
|---|---|---|---|---|
| Noun | 35.6 | 33.2 | 30.2 | 40.1 |
| Verb | 14.6 | 18.1 | 20.8 | 19.4 |
| Preposition | 12.6 | 11.2 | 9.1 | 5 |
| Adjective | 8.2 | 10.2 | 12.2 | 7.3 |
| Adverb | 5.5 | 6.2 | 4.1 | 7.9 |
| Conjunction | 4.9 | 5 | 7.2 | 6.1 |
| Digits | 3.8 | 4.2 | 5.5 | 2.1 |
| Determiner | 11.6 | 10.1 | 7.5 | 10.1 |
| Others | 3.2 | 1.6 | 4.3 | 1.7 |

Table 5.5: Distribution of POS for multiple languages

This pie chart depicts the distribution of part of speech for English language:

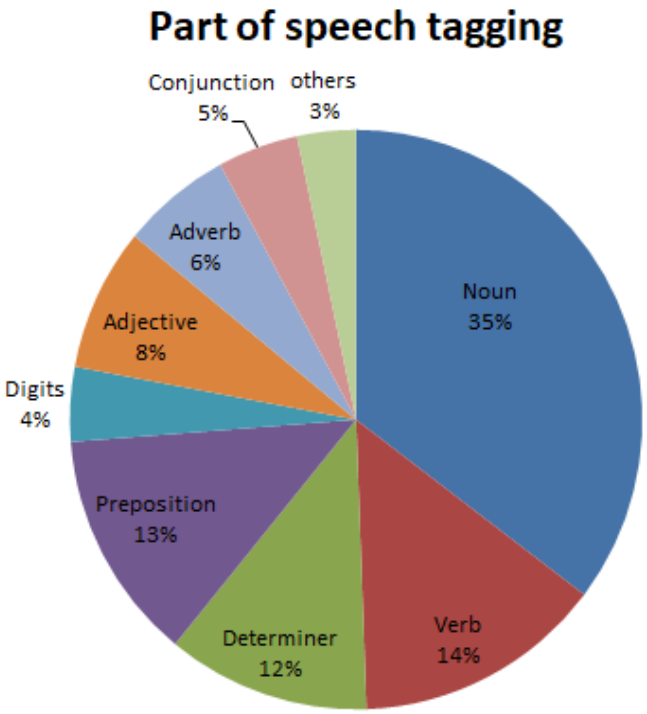


Fig. 5.1. POS tagging results for English

Chapter 6

# Conclusion and Future work

Created a tool to enrich the DBpedia NIF dataset with additional information. The output files generated by the tool for sentence splitting, tokenization, part of speech and link enhancement tasks enriches the NIF dataset. The individual output files generated from the tool are compressed to one zip file for every language and every task, which is the enriched DBpedia NIF dataset.
Result-sets generated by the tool consists of:
1) Sentences split on the text of Wiki articles
2) Segregated tokens on the text of Wiki articles
3) Part of speech for all the tokens on Wiki articles
4) Enhanced links on the text of Wiki articles
I have created a topic on DBpedia Forum [35] explaining the working of my tool and its capability for enriching the NIF dataset.

To conclude, DBpedia NIF datasets are studied in order to obtain the required details such as obtaining the full text of Wiki pages from NIF context file, obtaining the details of surface forms from NIF text links file. The obtained information from NIF datasets are trained with various NLP tasks to produce meaningful results which helps DBpedia in increasing the amount of its structured data. Various NLP operations that are applied on DBpedia NIF datasets are studied. The implementation followed with storing the triples of individual Wiki articles from the NIF context file separately. On the text of Wiki pages for multiple languages, NLP tasks namely sentence splitting, tokenization, POS tagging and link enhancement are performed. The results of these NLP tasks are compared and analysed for different libraries used to build my tool. A documentation on integration of new language or a new library to my tool is specified.

As a future work, the tool should be able to assign links to surface forms having more than 3 tokens. The algorithm to enhance links could be

improved by not providing links to same surface forms occurring consecutively and also improve the execution time of this task, if possible. The tool could be improved by its ability to perform more NLP tasks. The tool should be able to support more languages in its future releases.

# Bibliography


[1] NLP2RDP, *NIF Core 2.0 ontology Image*. https://github.com/NLP2RDF/ontologies, 2020.

[2] DBpedia, *DBpedia About [online]*. https://wiki.dbpedia.org/, 2019.

[3] Wikipedia, *DBpedia*. https://en.wikipedia.org/wiki/DBpedia, 2020.

[4] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," in *The semantic web*, pp. 722–735, Springer, 2007.

[5] W3C, *Semantic Web Standards*. https://www.w3.org/2001/sw/wiki/Main_Page, 2019.

[6] DBpedia, *DBpedia NIF Dataset*. https://wiki.dbpedia.org/dbpedia-nif-dataset, 2019.

[7] LOD, *Linked Open Data Cloud [online]*. https://lod-cloud.net/, 2019.

[8] DBpedia, *DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia*. http://www.semantic-web-journal.net/system/files/swj499.pdf, 2019.

[9] DBpedia, *DBpedia NIF Dataset Download [online]*. https://wiki.dbpedia.org/downloads-2016-10, 2018.

[10] AKSW, *DBpedia NIF Dataset: Open, Large-Scale and Multilingual Knowledge Extraction Corpus*. http://aksw.org/Projects/DBpediaNIF.html, 2019.

[11] W3C, *Linked Data*. https://www.w3.org/standards/semanticweb/data, 2019.

[12] W3C, *RDF [online]*. https://www.w3.org/RDF/, 2015.

[13] Wikipedia, *Linked Data*. https://en.wikipedia.org/wiki/Linked_data, 2020.

[14] T. B. Lee, *Linked Data*. https://www.w3.org/DesignIssues/LinkedData.html, April 2019.

[15] N. Ontology, *NIF 2.0 Core Ontology [online]*. https://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core/nif-core.html, 2015.

[16] D. D. Palmer, *Text Segmentation*. https://s3.amazonaws.com/tm-town-nlp-resources/ch2.pdf, 2015.

[17] NLTK, *Natural Language ToolKit*. https://www.nltk.org/, 2020.

[18] spaCy, *Spacy io*. https://spacy.io/, 2017.

[19] Pattern, *Python for NLP: Introduction to the Pattern Library*. https://stackabuse.com/python-for-nlp-introduction-to-the-pattern-library/, 2020.

[20] TextBlob, *TextBlob: Simplified Text Processing*. https://textblob.readthedocs.io/en/dev/, 2018.

[21] S. University, *Stanford POS Tagger*. https://nlp.stanford.edu/software/tagger.shtml, 2019.

[22] NLPforHackers, *Sentence Splitting*. https://nlpforhackers.io/splitting-text-into-sentences/, 2019.

[23] S. University, *Tokenization*. https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html, 2018.

[24] Wikifier, *Annotating documents with relevant Wikipedia concepts*. http://wikifier.org/, 2019.

[25] T. H. Ta and C. Anutariya, "A model for enriching multilingual wikipedias using infobox and wikidata property alignment," in *Joint International Semantic Technology Conference*, pp. 335–350, Springer, 2014.

[26] G. Ytrestøl, D. Flickinger, and S. Oepen, "Extracting and annotating wikipedia sub-domains—towards a new escience community resource," *LOT Occasional Series*, vol. 12, pp. 185–197, 2008.

[27] talend, *Data Processing*. https://www.talend.com/resources/what-is-data-processing/, 2017.

[28] NIF, *Provenance and Confidence for NIF annotations*. https://nif.readthedocs.io/en/latest/prov-and-conf.html, 2015.

[29] W3C, *Internationalization Tag Set (ITS) Version 2.0*. https://www.w3.org/TR/its20/, 2019.

[30] RDFLib, *RDFLib Project Description*. https://pypi.org/project/rdflib/, 2019.

[31] Nagisa, *Project description*. https://pypi.org/project/nagisa/, 2019.

[32] S. University, *Part Of Speech*. https://nlp.stanford.edu/software/tagger.shtml, 2018.

[33] P. Taggers, *Categorizing and Tagging Words*. http://www.nltk.org/book/ch05.html, 2020.

[34] P. Lakshmanan, *Enrichment of DBpedia NIF Dataset*. https://github.com/pragal18/Enrichment_of_DBpedia_NIF_Dataset, 2020.

[35] DBpedia, *DBpedia Forum*. https://forum.dbpedia.org, 2020.

# Acronyms

**NLP** Natural Language Processing

**NIF** NLP Interchange Format

**RDF** Resource Description Framework

**POS** Part of Speech

**TTL** Terse RDF Triple Language

**TQL** Terse RDF Quad-Turtle Language

**HDT** Header, Dictionary, Triples

**W3C** World Wide Web Consortium

**LOD** Linked Open Data

# Contents of enclosed CD

- `readme.txt` ..... File with CD contents description
- `Files` ..... Directory with files generated from various tasks
  - `Input<lang>` ..... Files generated from Separation of Wiki articles
  - `Links` ..... Output files generated from Links Enhancement
  - `POS` ..... Output files generated from Part of speech tagging
  - `Search` Output files generated from all 4 NLP tasks for specific file(s)
  - `Sentences` ..... Output files generated from Sentence splitting
  - `Tokens` ..... Output files generated from Tokenization
  - `LinkDataset<lang>.csv` ..... Generated CSV file for each language
- `NIF_Dataset_Minimal_Version` ..... subsets of NIF dataset
  - `nif_context_minimal_version_<lang>.ttl` ... NIF context dataset
  - `nif_text_links_minimal_version_<lang>.ttl` . Text links dataset
- `scripts` ..... All python scripts regarding the implementation
  - `Pre-processing scripts` .. Python scripts related to analysing and pre-processing tasks
  - `<NLP Task>_<library>.py` scripts wrt implementation of NLP tasks
- `run.sh` ..... script performing all NLP tasks
- `separate_scripts.sh` ..... script performing pre-processing tasks
- `readme.md` ..... Markdown consisting the steps for reproducing results
- `DP_Lakshmanan_Pragalbha.pdf` ..... Thesis text in PDF format