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 which aims at 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.

7th May 2019

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 the 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 7th May 2019 . . . . . .

Citation of this thesis

Lakshmanan, Pragalbha. *Enrichment of the DBpedia NIF dataset.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstract

DBpedia je komunitní úsilí založené na davu, jehož cílem je získávat informace z článků Wikipedie a poskytovat tyto informace ve strojově čitelném formátu. Datový soubor DBpedia NIF poskytuje obsah všech článků Wikipedie ve 128 jazycích. Konečným cílem práce je obohatit datovou sadu o další informace, jejichž hlavní výzvou je velikost datové sady. Implementace spočívá v předběžném zpracování souboru dat oddělením obsahu jednotlivých článků Wikipedie do samostatných souborů, protože soubor dat NIF obsahuje obsah všech článků v jednom obrovském souboru. Předběžné zpracování textu napomáhá k využití datové sady pro školení různých úloh zpracování přirozeného jazyka. Po provedení následuje provedení NLP úkolů, a to rozdělení vět, Tokenizace, část značkování řeči. Nakonec přispět do komunity DBpedia přidáním dalších odkazů na články wikipedia. Konečně, vyhodnocení výsledků a kontrola správnosti výsledků statisticky.

Klíčová slova DBpedia, datový soubor NIF, obohacení datové sady, předzpracování textu, úlohy NLP, další odkazy

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 NIF dataset provides the content of all Wikipedia articles in 128 languages. The ultimate goal of the thesis is to enrich the dataset with additional information where the main challenge is the size of the dataset. The implementation comprises of pre-processing the dataset by segregating the contents of individual Wikipedia articles into separate files, as the NIF dataset comprises the contents of all the articles in one huge file. The text pre-processing helps in order to use the dataset for training different Natural language processing tasks. The implementation is followed by performing NLP tasks namely sentence splitting, Tokenization, Part of speech tagging. Eventually contribute to the DBpedia community by adding additional links to the wikipedia articles. Finally, evaluating the results and checking the correctness of the results statistically.

Keywords DBpedia, NIF dataset, enrich dataset, text pre-processing, NLP tasks, additional links

Contents

List of Figures

Chapter 1

Introduction

The DBpedia community project[1] extracts knowledge from Wikipedia and makes it widely available via established Semantic Web standards. It extracts structured content from the information available in the Wikipedia project. DBpedia community provides this information in a machine readable format. It allows users to semantically query relationships and properties of Wikipedia resources, including links to other related datasets. DBpedia is one of the most famous parts of the decentralized Linked Data effort. One of the core datasets behind DBpedia is the DBpedia NIF dataset which provides the content,structure and links of all Wikipedia articles.

The DBpedia community uses a flexible and extensible framework to extract different kinds of structured information from Wikipedia. DBpedia Extraction Framework gets the most recent revision from the Github repository - git://github.com/dbpedia/extraction-framework.git, the master branch contains the latest version.

DBpedia focuses on representing the knowledge that are present in the Wikipedia Infoboxes. Majority of the information is contained in unstructured Wikipedia articles text. In order to broaden and deepen the amount of structured DBpedia data, they have gone a step further. 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: the full text of a page as context (including begin and end index)
2) nif-page-structure: the structure of the page in sections and paragraphs (titles, subsections etc.)
3) nif-text-links: all in-text links to other DBpedia resources as well as external references

The DBpedia NIF dataset is stored in RDF. Resource Description Framework is realized with the concept of triples (Subject - Predicate - Object). The NIF dataset is available in a variety of formats like .TQL, .TTL , .HDT. This ensures that the data is stored in a structured manner and the exact piece of information required from a Wikipedia page could be retrieved without much struggle.

Key features of NIF dataset[2] -
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
4) Exploited within the TextExt - DBpedia Open Extraction challenge
5) Available for large-scale training NLP methods

The extracted knowledge from DBpedia, comprising more than 1.8 billion facts, is structured according to an ontology maintained by the community[3]. The knowledge is obtained from different Wikipedia language editions, thus covering more than 100 languages, and mapped to the community ontology. The resulting datasets are linked to more than 100 other datasets in the Linked Open Data (LOD) cloud. The DBpedia project was started in 2006 and has meanwhile attracted large interest in research and practice.

NIF dataset is parsed using a python library called RDFlib. According to wikipedia - "Parsing, syntax analysis, or syntactic analysis is the process of analysing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar". Once parsed, it becomes easier to handle and process the data for performing various tasks. Natural Language Processing, usually shortened as 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.

The Natural Language Processing tasks that I used in my thesis includes:
1) Sentence splitting - Sentence splitting means to split a given paragraph of text into sentences, by identifying the sentence boundaries.
2) Tokenization - Tokenization is the operation of chopping up of 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 each tokens

or words it detects, such as noun, verb, adjective, etc.
4) Enrichment of Links - Provide additional links to the Wikipedia articles. Reading the text of Wikipedia articles word by word in order to check if an additional link could be provided (when the word which is given a link is clicked, redirects browser to another Wikipedia page).

Apart from these NLP tasks there are few pre-processing tasks as well :
1) Separation of every wikipedia article from each other which were all originally stored in one huge file namely NIF context. This helps in faster processing of the dataset.
2) Converting the machine readable format NIF text links dataset into a CSV file by taking only the required parameters. This is done with the mindset that search operation is faster in csv file rather than RDF file which is realized using the concept of triples.

The combination of pre-preprocessing tasks and the NLP tasks would be my main focus point of the thesis. I have discussed in detail on upcoming chapters about how the data is processed and how efficiently it is done considering the size of the dataset which is exceptionally huge.

1.1 Motivation

Wikipedia is an online encyclopedia in about 290 languages with exclusively free content, based on open collaboration through a model of content edit by web-based applications like web browsers called wiki. Wikipedia has over 5 million articles in English covering all the important topics in almost all the fields. It has over 40 Million articles in over 290 languages[4]. Wikipedia has become part and parcel of our everyday life. DBpedia is a Large-scale, multilingual Knowledge Base Extracted from Wikipedia. The DBpedia project was started in 2006 and has meanwhile attracted large interest in research and practice. Since, it is an interesting topic and good field to do research on, this would be my main reason for motivation. I am motivated to contribute to the NIF dataset provided by DBpedia community, in terms of increasing the readability and performing some interesting tasks on it for analysing their results. Performing tasks like sentence splitting on the dataset allows to compare the average number of sentences in a wikipedia article whereas part-of-speech tagging allows to find the commonly used POS in various articles. There are various other interesting insights found from the analysis (Refer section 4.3).

Linking Wikipedia articles to each other is quite important. These links allows users to jump to new article from the original article, for understanding 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 the same article. There are many Researches researching in this field but not many people concentrate on enriching the NIF dataset. So, I believed this would be something good for me to carry out. Data pre-processing and Natural Language processing tasks would help me to improve my knowledge in the Artificial Intelligence field which has a really good market value right now. The ability to process a huge dataset and retrieve the information that we need from it, is typical requirement for this thesis topic which I believe is interesting and the fact that I get to contribute to DBpedia community gives me motivation to pursue this.

1.2 Objectives and Goals

The objective of my thesis is to study the DBpedia's effort on extracting information from Wikipedia that is provided in a machine readable format. Enhance my understanding of the NIF dataset provided by them. The objectives of the thesis are:
1) Performing sentence splitting on individual Wikipedia articles
2) Performing Tokenization on the articles
3) Part-of-speech tagging for every token on the articles
4) Find out all the words in the articles that has link to another wikipedia page(s) and store them in a CSV file.
5) Add more links in the articles by reading the article word by word in order to check if a link could be possibly added to the word.

The goals of the thesis include:
1) Contribute to the DBpedia community by enriching the NIF dataset.
2) The output of tasks like sentence splitting, Tokenization and Part of speech tagging would serve as a stepping stone to perform more complicated NLP tasks.

Apart from these goals and objectives, I also focus on analysing the results of sentence splitting, tokenization, part-of-speech tagging and enrichment of links, to gain insights from them. Also evaluate the correctness of the results and depict them statistically.

Chapter 2

Background and Related works

2.1 Background

2.1.1 Resource Description Framework

2.1.1.1 What is RDF?

Resource Description Framework (RDF) , is a family of specifications developed by the organization World Wide Web Consortium (W3C), originally designed as a metadata model[5]. It is used as a general method for modelling information in different syntaxes. It is a general framework of data that describes the source document so that its description is readable both humanly and mechanically. RDF is a standardized format that lets you express descriptive information about Web resources. At the same time, it is also a graph format, so all data in RDF can be written using a graph with oriented edges that can be written as a set of triads. The source that can be described by RDF can be any resource that can be uniquely identified by a Uniform Resource Identifier (URI). This identifier uniquely determines which particular resource is involved in web content that is composed of different types of documents.

2.1.1.2 Triples

A semantic triple, or simply triple, is the atomic data entity in the Resource Description Framework (RDF) data model. As its name indicates, a triple is a set of three entities that codifies a statement about semantic data in the form of subject, predicate and object expressions. An example could be "Pragalbha is 23" or "Pragalbha is from India". The main advantage of having it in such a format is that machines can read them easy or the so called machine readable format. 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, has the colour

as predicate and black as object. In the Object oriented design we have a similar notation that is entity attribute and value model where in the entity is elephant, attribute is the colour and value is black.

What makes RDF triples special is that EVERY PART of the triple has a URI associated with it, so the everyday statement "Mike knows John " might be represented in RDF as:

```
1 uri://people#MikeSmith12 http://xmlns.com/foaf/0.1/knows uri://people#
      JohnDoe45
```

2.1.1.3 Various Serialization Formats

Turtle : It is a syntax and file format for expressing data in the Resource Description Framework (RDF) data model. Turtle syntax is similar to that of SPARQL, an RDF query language. It is a compact human friendly format. RDF in Terse RDF Triple Language (Turtle) format is much easier as you can define prefixes at the beginning of the .ttl file, shortening each triple. Another feature of turtle is that multiple triples with the same subject are grouped into blocks for example:

```
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)
5 nif.beginIndex "0"
6 a nif.Context
7 nif.endIndex "2294"
8 nif.predLang <http://lexvo.org/id/iso639-3/eng>
```

Here, all the triples have the common subject "dbr:Animalia_(book)". One unique shortening is the predicate <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> which is so common, it is represented simply with the letter 'a' (Line 6).

N-triples : Storing and reading RDF as N-Triples is simple as every line of a .nt file is a single triple (<subject> <predicate> <object>) that together form 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. It is very simple and easy-to-parse that is not as compact as Turtle. 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" .
```

RDF/XML : It is an XML-based syntax that was 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 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."

Using linked data, statements encoded in triples can be spread across different websites. On Website A we can present the entity John and the fact that he knows Stephan. On website B we can provide all the information about Stephan and on the Website C we can find information about Stephan's hometown.

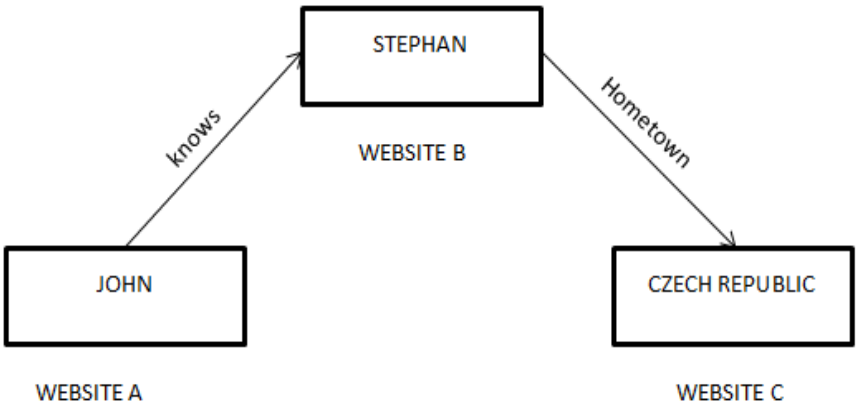

Fig. 2.1. Linked Data example

Each page contains the structured data to describe an entity and the link to the entity that could be described on a different website. 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." In computing linked data describes a method of

2.1.3 DBpedia

2.1.3.1 What is DBPedia?

It 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. 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 projects. Wikimedia is a global movement whose mission is to bring free educational content to the world. 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 in the sum of 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. Google uses a similar approach to create those knowledge cards during search.

The DBpedia hopes that this work will make it easier for the huge amount of information in Wikimedia projects to be used in some new interesting ways. DBpedia data is served as Linked Data, which is revolutionizing the way applications interact with the Web. One can navigate this Web of facts with standard Web browsers, automated crawlers or pose complex queries with SQL-like query languages (e.g. SPARQL). The DBpedia community has a very good intention of converting the wikipedia articles into a machine readable format. However, there are quite some challenges as the Wikipedia contents keep increasing and that should be updated in DBpedia as well, or modifying a particular article which also needs to be updated. The DBpedia dataset provided in RDF triples is hosted and publishes by using OpenLink Virtuso. The access to the DBpedia's RDF dataset is provided through a Sparql endpoint, alongside HTTP support for any Web client's standard GETs for HTML or RDF representations of DBpedia resources.

The DBpedia Knowledge Base

A knowledge base is used to store complex structured and unstructured information used by a computer system. The initial use of the term was in connection with expert systems which were the first knowledge-based systems. It helps in improving the intelligence of Web and enterprise search.

Information integration is one of the main features which Knowledge base supports. Nowadays most of the knowledge bases created are for only specific domains and are build by small groups of knowledge engineers and are very cost intensive to keep up-to-date as domains change. On the contrary, Wikipedia has now become one of the main source for knowledge source for all human being. Wikipedia is maintained by thousands of contributors. The DBpedia project leverages this gigantic source of knowledge by extracting structured information from Wikipedia and by making this information accessible on the Web under the terms of the Creative Commons Attribution-ShareAlike 3.0 License and the GNU Free Documentation License.

DBpedia ontology

It has been manually created from the infoboxes inside the Wikipedia. This ontology is shallow and available across multiple domains. This ontology currently uses 685 classes which are described by about 2800 different properties. 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.The table below lists the number of instances for several classes within the ontology:

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

The DBpedia Ontology can also be queried via the DBpedia SPARQL endpoint and can be explored via the DBpedia Linked Data interface. Examples: Class Place, property elevation.

2.1.3.2 The English version of the DBpedia knowledge base

The DBpedia's knowledge Base for the English version consists of about 4.58 million things. In the 4.58 million 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. These is in contrary with 5.7 million wikipedia articles which are in English[7].

DBpedia provides localized versions of DBpedia in 129 languages. Now if we look at all the versions, then it provides about 38.3 million things, out of which 23.8 million are localized descriptions of things that also exist in the English version of DBpedia. The full DBpedia data set features 38 million labels and abstracts in 125 different languages. It provides 25.2 million links to images and 29.8 million links to external web pages. Usually the external web pages links are provided in the Related links section in every Wikipedia article. But however most of the links point to another wikipedia article itself and there are about 80.9 million links to Wikipedia categories, and 41.2 million links to YAGO categories. DBpedia is connected with other Linked Datasets by around 50 million RDF links.The DBpedia 2014 releases consists of 3 Billion piece of information in the form of RDF tripels out of which 580 million were extracted from the English edition of Wikipedia, 2.46 billion were extracted from other language editions. If we like notice carefully only 11.7% of the total wikipedia articles belongs to English. About 10.7% belongs to Cebuano, a language spoken in Phillipines, 7.5% belongs to Swedish and 4.5% in German and 4.2% in French. Thus even in DBpedia there are 4.58 million things in English comparison to overall 38.3 million things in all the languages is about 13% which is close to 11.7%. To get into more details, there are articles in languages like Cebuano consisting of just one or two sentences. Probaby they had created some bots and is doing automatically for few articles.

Now, lets take a look at the advantages of DBpedia knowledge base as a comparison to other knowledge bases
1)It covers many domains
2)It represents real community agreement
3)It automatically evolves as Wikipedia changes
4)It is truly multilingual.

The DBpedia knowledge could help you get interesting information for very complex queries unlike Wikipedia. For example we can retrieve thes information from DBpedia whereas Wikipedia doesn’t provide answers for these queries: Display all the Cities in the world with a population more than 10 million people or give me 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.

2.1.4 DBpedia NIF Dataset

DBpedia's primary focus is on retrieving the factual knowledge from the Wikipedia infoboxes. Predominantly large amount of data is available in the unstructured format of texts in the Wikipedia articles. They have thereby going a step further to broaden and deepen the unstructured DBpedia dataset. With the representation of wiki pages in the NLP Interchange Format (NIF) we provide all information directly extractable from the HTML source code divided into three datasets:
1) NIF context: the full text of a page as context (including begin and end index)
2) NIF Page Structure: the structure of the page in sections and paragraphs (titles, subsections etc.)
3) NIF Text Links: all in-text links to other DBpedia resources as well as external references

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 provide the whole wiki page text in the NIF format.

The NLP Interchange Format (NIF) aims to achieve interoperability between Natural Language Processing (NLP) tools, language resources and annotations. To extend the versatility of DBpedia, furthering many NLP-related tasks, they decided to extract the complete human- readable 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

The NIF 2.0 Core Ontology[8] provides classes and properties to describe the relations 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. The subclass of nif:String is the nif:Context OWL class. This class is assigned to the whole string of the text (i.e. all characters). The purpose of an individual of this class is special, because the string of this 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 the text in every wikipedia article.

Combination of nif:word could be either a nif:title, nif:phrase or nif:paragraph which are defined in the nif:Structure. Thus, nif:word is a subclass of

The name of the Wikipedia article is Anthropology as it is evident from every line after "dbr:" in the subject of all the triples, and ends before the "?". There are 6 triples in this small example of nif-context.ttl. All these triple correspond to a single wikipedia article that is Anthropology and as mentioned, each article has 6 triples. Each triple implies the following:
1) It is a part of NIF context file.
2) The actual words present in the Wikipedia article 'anthropology' with a predicate indicating it is the string.
3) The third triple indicates the begin index of the article which is always 0.
4) The fourth triple indicates the end Index of the wikipedia article.
5) The fifth triple gives the URL link to access this Wikipedia article which when typed in a browser opens the Anthropology Wikipedia article.
6) The sixth triple indicates the language of the content.

2.1.4.3 NIF Page Structure

From the name it is quite obvious to guess that this file gives the structure of each Wikipedia article. The structure of the Wikipedia page as NIF:Structure instances, such as Section, Paragraph and Title. Generally there are many sections and paragraphs in a wikipedia article. If you take a sportsman's wikipedia page , there are many sections like - Early age, personal life, achievements etc. Within each section there could be multiple paragraphs. It is necessary to keep track of this sections, titles and paragraphs in DBPedia for storing this unstructured text articles into a machine readable format. I haven't used this file for my thesis as I don't need to segregate anything by paragraphs or sections for performing my NLP tasks. But this is also a very important dataset provided by DBpedia which comes in two extensions .TTL and .TQL . So lets see a small example of this file and I would try to explain it below:

```
1 dbr:Anthropology?dbpv=2016-04&nif=context    nif:hasSection    dbr:
      Anthropology?dbpv=2016-04&nif=section_0_634    .
2 dbr:Anthropology?dbpv=2016-04&nif=section_0_634    nif:beginIndex    "0"^^<
      http://www.w3.org/2001/XMLSchema#nonNegativeInteger>    .
3 dbr:Anthropology?dbpv=2016-04&nif=section_0_634    nif:endIndex    "634"^^<
      http://www.w3.org/2001/XMLSchema#nonNegativeInteger>    .
4 dbr:Anthropology?dbpv=2016-04&nif=section_0_634    nif:referenceContext
      dbr:Anthropology?dbpv=2016-04&nif=context    .
5 dbr:Anthropology?dbpv=2016-04&nif=section_0_634    nif:hasParagraph    dbr:
      Anthropology?dbpv=2016-04&nif=paragraph_0_330    .
6 dbr:Anthropology?dbpv=2016-04&nif=section_0_634    nif:hasParagraph    dbr:
      Anthropology?dbpv=2016-04&nif=paragraph_331_634   .
7 dbr:Anthropology?dbpv=2016-04&nif=section_0_634    nif:firstParagraph
      dbr:Anthropology?dbpv=2016-04&nif=paragraph_0_330    .
8 dbr:Anthropology?dbpv=2016-04&nif=section_0_634    nif:lastParagraph    dbr
      :Anthropology?dbpv=2016-04&nif=paragraph_331_63    .
```

We can infer the following from these triples (each point corresponds to the line number):

1) The wikipedia article 'Anthropology' has a section
2) The section's begin index is 0
3) The section has end index of 634
4) The content's of this section could be seen in NIF-context file as it gives the reference to it.
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

Also 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"^^<http://www.w3.org/2001/XMLSchema#nonNegativeInteger>    .
3 dbr:Anthropology?dbpv=2016-04&nif=paragraph_0_330    nif:endIndex
    "330"^^<http://www.w3.org/2001/XMLSchema#nonNegativeInteger>    .
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    .
```

This describes the Paragraphs properties and this is linked to the section as well with a superstring attribute indicating that this paragraph is a part of the section that has index 0 to 634. We can understand the following from each of the triples:
1) This is a paragraph in the article Anthropology
2) The Begin index of the paragraph is 0.
3) The end index of the paragraph is 330.
4) The reference to see the content of 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 article. It can be noticed from the predicate being superString.

2.1.4.4 NIF Text Links

This dataset contains the words that has link to another page. However, this words are recorded in a highly structured format. This is again a huge dataset with about 60GB containing hundreds of millions of lines. This comes in two extensions namely .TTL and .TQL . I have thus used .TTL extension for my thesis due to same reasons that I had mentioned for NIF context file. Just to clarify , in the screenshot below taken from the wikipedia article 'Animalia book', the words 'children's book' and 'Graeme Base' consist of links to other Wikipedia articles. The words which when clicked redirects the browser to another wikipedia page are called as Surface forms. In this dataset, the details of these surface forms are recorded. However, there is a separate way of recording whether the link

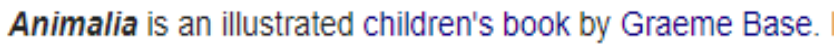

Fig. 2.4. Surface Forms are the words in blue colour

exists on a single word or a collection of words. If the link exists on only one word, then it is mentioned in the triples clearly that the link exists on only one word. So lets take a look at a small example of this dataset:

```
dbr:Anthropology?dbpv=2016-04&nif=word_29_37    a    nif:Word .
dbr:Anthropology?dbpv=2016-04&nif=word_29_37    nif:referenceContext    dbr
    :Anthropology?dbpv=2016-04&nif=context .
dbr:Anthropology?dbpv=2016-04&nif=word_29_37    nif:beginIndex    "29"^^<
    http://www.w3.org/2001/XMLSchema#nonNegativeInteger> .
dbr:Anthropology?dbpv=2016-04&nif=word_29_37    nif:endIndex    "37"^^<http
    ://www.w3.org/2001/XMLSchema#nonNegativeInteger> .
dbr:Anthropology?dbpv=2016-04&nif=word_29_37    nif:superString    dbr:
    Anthropology?dbpv=2016-04&nif=paragraph_0_634 .
dbr:Anthropology?dbpv=2016-04&nif=word_29_37    <http://www.w3.org/2005/11/
    its/rdf#taIdentRef>    dbr:Human .
dbr:Anthropology?dbpv=2016-04&nif=word_29_37    nif:anchorOf    "humanity"
    .
```

The important information that could be gathered from this is :
1) The link exists on only one word. If the link had existed on more than word, we could see that instead of nif:Word there would be nif:Phrase.
2) The content is available in NIF context dataset with a reference given to it.
3) The word that has the link on it has a begin index of 29.
4) The word that has the link on it has a end index of 37.
5) It is a part of the paragraph with index 0 to 634 on the Anthropology Wikipedia article.
6) When this word is clicked, the page that needs to load is the Wikipedia article of Human. Just to clarify things, here "dbr:" is a namespace. It just needs to be defined in the beginning once that dbr corresponds to https://en.wikipedia.org/wiki/ . Everywhere within the script we could just use dbr instead of the whole URL. So here the URL for the new page that should open is "https://en.wikipedia.org/wiki/Human".
7) The content of the word that has the link from index 29 to 37 is "humanity".

So, if the link had existed on more than one word, it would appear like this:

```
dbr:Anthropology?dbpv=2016-04&nif=phrase_65_84    a    nif:Phrase    .
dbr:Anthropology?dbpv=2016-04&nif=phrase_65_84    nif:anchorOf    "social
    anthropology"
```

This specifies that with index from 65 to 84, there exists a link on words ’social anthropology’ pointing to another Wikipedia article. In the first line, you could see that it is a phrase which indicates that a single consists

of more than one word. However, in the previuous case, it was nif:Word instead of nif:Phrase.

2.1.5 Natural Language Processing

2.1.5.1 What is NLP?

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 things like Machine Translation and text-filtering. NLP is a very much benefited from the recent advancement of machine learning esp 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.

2.1.5.2 Syntactic and Semantic Analysis

Syntactic Analysis (Syntax) and Semantic Analysis (Semantic) are the two main techniques that are part of 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.

2.1.5.3 Techniques to understand Text

Parsing is a kind of formal analysis where every word in the sentence is looked word by word that is breaking into individual constituents of a sentence, which results in a parse tree that shows their syntactic relation to each other in visual form. This can be used for further processing and understanding.

Stemming is a technique that comes from morphology and information retrieval which is used in NLP for preprocessing and efficiency purposes.

Basically, Stemming is the process of reducing words to their word stem but what is actually meant by stem? A ’stem’ is that part of a word that remains after the removal of all affixes. So for example, if you take a look at the word ’punched’, it’s stem would be ’punch’. ’Punch’ is also the stem of ’punching’ and so on.

Text Segmentation 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’.

With Sentiment Analysis, we want to determine the attitude (e.g the sentiment) of, for example, a speaker or writer with respect to a document, interaction, or event. Therefore it is a natural language processing problem where text needs to be understood, to predict the underlying intent. The sentiment is mostly categorized into positive, negative and neutral categories.

Relationship Extraction takes the named entities of ’Named Entity Recognition’ and tries to identify the semantic relationships between them. This could mean for example finding out who is married to whom, that a person works for a specific company and so on. This problem can also be transformed into a classification problem where you can train a Machine Learning model for every relationship type.

2.1.5.4 NLP Tasks used in the thesis

Sentence splitting - Sentence separation, means, to split a given paragraph of text into sentences, by identifying the sentence boundaries. 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. Store the result in a huge list. This is an open ended challenge to which there are no perfect solutions. Try to break up given paragraphs into text into individual sentences. Let’s look at the challenges faced:
1)Abbreviations: Dr. H.N.Abraham is the author of Animalia. In this case, the first and second "." occurs after Dr (Doctor) and H.N(initial in the person’s name) and should not be confused as the end of the sentence.
2)Sentences enclosed in quotes: "What good are they? They’re led about just for show!" remarked another. All of this, should be identified as just 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.

Tokenization- Given a string or a paragraph or a sequence of characters and a defined document unit, tokenization 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. This is a requirement in natural language processing tasks. Usually word tokenization is done by separation of spaces. Let's look at the challenges faced:
1) The use of abbreviations can prompt the tokenizer to detect a sentence boundary where there is none.
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.

Part of speech - POS tagger is a piece of software that reads text and assigns parts of speech to each tokens or words it detects, such as noun, verb, adjective, etc., although NLTK uses POS tags like 'noun-plural' or 'noun-singular' etc. The main issue is that most of the word has more than one meaning and they could be used in different context having completely different part of speech. Lets take a look at the following example:
Let us look 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.)

Enrichment of additional links - Provide additional links to the Wikipedia articles. Reading the text of Wikipedia articles word by word in order to check if an additional link could be provided. when the word which is given a link is clicked redirects browser to another Wikipedia page. Please refer section 3.8 for in detail explanation of this task and the challenges faced.

Chapter 3

Data Processing

Data processing is, generally, the collection and manipulation of items of data to produce meaningful information. Carrying out of operations on data, especially by a computer, to retrieve, transform, or classify information. With respect to my thesis, the NIF-context and NIF-text-link is the dataset on which various operations are performed (Sentence splitting, Tokenization, Part of speech tagging and addition of links). The information retrieved could be used to perform complex NLP tasks. Also obtained some useful insights by analysing the results of this operations.

3.1 Requirements

I have downloaded the NIF datasets(context,structure and links) in .TTL extension in a compressed format. I decompressed them and these are unbelievably huge files. The nif-context is approx 6GB, nif-links is approx 60GB. I haven't used the structure file for any of the purpose, so lets not concentrate about it as of now. I will enumerate the steps that I carried out and lets look in depth on each step in this Chapter later on.

- **STEP 1** : The first target is to separate the NIF-context.ttl file into individual wikipedia articles as this helps to pre-process the data and perform various nltk operations in a much easier fashion. The goal is to separate the triples in nif-context.ttl file corresponding to individual wikipedia article. So basically I need to create about 4.8 million individual ttl files(each corresponding to one wikipedia article) as there are about 4.8 million wikipedia articles in english. Lets discuss about the pseudocode and sample of a generated file in the section 3.3 . I have saved each file in this format : <name of the wikipedia article>.ttl .

end index of each token along with the content/text of each token. So lets say if there are 6000 words in a wikipedia article, then each word is a token and the resultant ttl file generated should consist of 6000 tokens. These files are named <name of the wikipedia article>-tokenization.ttl .

- **STEP 4** : Once I have the tokens individually separated in every wikipedia article, my next goal is to find the corresponding part of speech for these tokens. However, in this case I should find the part of speech of every token by using sentences because part of speech varies depending upon the context it is used. For eg:- The word 'next' could be used as Adjective,Adverb, Preposition or Noun depending upon the context it is used. I would discuss this in detail on the upcoming sections. For now, the goal is to identify the part of speech of every word/token used in the wikipedia articles and store the final results in <name of the wikipedia article>-pos.ttl. This is done with the nltk library using pos tagger. The generated ttl file should contain all details that appear in tokeniztaion file along with the three more triples. One triple indicating the short form of part of speech like NNP (this is basically the result of pos tagger via the nltk library), another triple converting short form to the full form basically informing that NNP is Noun and the final triple indicating the type of Noun that is in our case Proper Noun. This will be handy to compare while creating new links for a new article.

- **STEP 5** : Here comes the need of nif-text-links.ttl file. This file approximately is about 60GB. My ultimate goal of this step is to create a csv file that consists of three columns. First column should consist of all the words that has link to some other page in all the wikipedia article(The words that have link to another page is called as surface forms). So, first column should have all the surface forms. The second column should consist the link to the page that will load when surface form is clicked. So, the second column should consist of this URL link to the new page. The third column basically is the count implying the number of times surface form and its corresponding link repeats throughout all the articles in wikipedia. This is just a single csv file generated and named surfaceforms.csv .

- **STEP 6** : Now that we have everything we need to add more links, in this step we process the text of every wikipedia article(available from step 1) and process it word by word in attempt to provide additional link. If there is a match between the word and the surface-form

column in the surfaceforms.csv file obtained from the previous step, then a link(obtained from Links column in surfaceforms.csv file) is provided to this word. Also, the longest possible match is taken into consideration. For eg:- colouring book , here colouring has a match in the surface form on the csv file and also colouring book is present in the surface form list. So, we should ignore colouring and provide a link to colouring book.

This is the stepwise procedure that I have carried out in my thesis. I have discussed all theses steps in detail along with snippets. I used python programming language for the implementation.

3.2 Python libraries

3.2.1 RDFlib

RDFLib is a pure Python package helps in working with RDF files. Let's take a look at some important features and what RDFlib supports:
1)RDFLib contains most things that need to work with RDF, including: parsers and serializers for RDF/XML, N3, NTriples, N-Quads, Turtle, TriX, RDFa and Microdata.
2)a Graph interface which can be backed by any one of a number of Store implementations.
3)store implementations for in memory storage and persistent storage on top of the Berkeley DB.
4)a SPARQL 1.1 implementation - supporting SPARQL 1.1 Queries and Update statements.

RDFLib is a Python library for working with RDF, a simple yet powerful language for representing information. The library contains an RDF/XML parser/serializer that conforms to the RDF/XML Syntax Specification (Revised). The library also contains both in-memory and persistent Graph backends.

RDFlib basically reads the triples and stores it as a graph. parse() converts into a graphical form with nodes corresponding to each subject. It sometimes takes huge amount of processing time to parse huge files with millions of triples but once parsed it becomes really friendly to handle it. Parsing a RDF file automatically segregates the subject, predicate and object of the triples in the file. After parsing the file, just a loop is sufficient to iterate over every triple and the output of such script would look something like this (for the given input file):

```
1 (rdflib.term.URIRef('http://bigasterisk.com/foaf.rdf#drewp'),
```

```
2  rdflib.term.URIRef('http://example.com/says'),
3  rdflib.term.Literal(u'Hello'))
4 (rdflib.term.URIRef('http://bigasterisk.com/foaf.rdf#drewp'),
5  rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-
       ns#type'),
6  rdflib.term.URIRef('http://xmlns.com/foaf/0.1/Person'))
```

Here you could see there are 2 triples separated by brackets and each triple has individual subject predicate and object. RDFlib is a python client that allows me to read the NIF datasets which are available in Turtle format.

3.2.2 NLTK

3.2.2.1 General Information

NLTK is a leading platform for building Python programs to work with human language data. 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.

There are lot of cool features provided by nltk in processing a dataset with various operations such as separating the content of a file with individual sentences, or separation of individual words, or find the part of speech of each tokens, or even separate words by spaces and so on.

3.2.2.2 Example with explanation

```
1 import nltk
2 from nltk import word_tokenize
3 text="I am human being"
4 tokens=word_tokenize(text)
5 print(tokens)
```

nltk package is imported and a wordtokenize subpackage derived from NLTK as well. This basically splits the string into individual tokens and displays the output as shown below:

['I', 'am', 'human', 'being']

3.2.3 Pyspark

PySpark helps data scientists interface with Resilient Distributed datasets in apache spark and python.Py4J is a popularly library integrated within PySpark that lets python interface dynamically with JVM objects (RDD's). Apache Spark comes with an interactive shell for python. SparkContext:

Main entry point for Spark functionality. RDD: A Resilient Distributed Dataset (RDD), the basic abstraction in Spark SparkConf: For configuring Spark. SparkFiles:Access files shipped with jobs.

It generally allows you to run the program in multiple cores which means processing huge dataset should be much faster. However I tested pyspark but it didn't help me much with the performance as the laptop that I use has only 2 cores. However, I tried this and though so to include in my thesis.

```
pyspark —master local[2]
```

The master parameter is used for setting the master node address. Here we launch Spark locally on 2 cores for local testing. A small snippet of how pyspark is used:

```
import pyspark
from pyspark import SparkContext
sc=SparkContext()
sc.range(5).collect()
[0, 1, 2, 3, 4]
```

3.3 Separation of Wikipedia articles

The input file for this step is nif-context.ttl. We separate the nif-context.ttl file which comprises triples of all the Wikipedia articles, into individual wikipedia article by storing the triples of individual articles into separate files. This is done because it helps to process the dataset quicker and perform various operations, NLP tasks faster and easier on the individual articles. This is a preprocessing step to perform the NLP tasks on NIF dataset.

```
num_lines = sum(1 for line in open('nif_context_en.ttl',
    encoding='UTF-8'))
```

This python script counts the total number of lines in the NIF context file. This is helpful to cover all the lines in the file without missing any of them by iterating via a loop specifying the total number of iterations to be equivalent to num_lines.

By running this command, the output is 29642397, which means that there are 29642397 lines present in this context file. Each wikipedia article comprise of 6 lines or 6 triples in the nif-context.ttl, so there are 4.8 million files. Thus by multiplying them with 6 (4.8m * 6) we get 29 million lines or 29 million triples. The output we received by running the command should be similar to 29 million triples and it is "29642397". Now the goal is to

read the context file line by line until the end of the file is reached which can be detected by checking if total lines read is equivalent to num_lines found above. I have added the pseudocode for separation of articles below and beneath it I have explained how this works:

```
1  previousSubject = None
2  file=open('nif_context_en.ttl',encoding="utf-8")
3  for i in range(29642397):
4      fileline=file.readline()
5          s=fileline.find('?')
6          thisSubject=fileline[29:s]
7          if previousSubject==thisSubject:
8              f.write(fileline)
9          else :
10             try:
11                 f=open("Context/"+thisSubject+".ttl",'a',
                       encoding="utf-8")
12                 f.write(fileline)
13             except:
14                 pass
15     previousSubject=thisSubject
```

In the first line, the previousSubject keeps track of the name of wikipedia article read in the previous line. Then the file nif-context is opened with UTF-8 encoding so that it allows to read all the special characters as well, which might appear in the file. A for loop that runs the code within it for 29,642,397 times to read all the lines of NIF context file. The variable fileline stores the currently read line on the particular iteration, as the readline() functions reads a line(lines 2-4). One great thing about this dataset is that, the triples of all the articles are in a sequential order. They aren't mixed up. All the 6 triples of each article occurs one after the other. So this allows to create a new file if previousSubject is not same as the current one meaning that all the triples of a particular article is read. Let's take a look at one typical example of a read line to understand the further parts of this python script:

```
1  <http://dbpedia.org/resource/Antarctic?dbpv=2016-10&nif=context> <http://
       persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core#endIndex>
       "6039"
```

Here the name of the Wikipedia article is the Antartic which appears from index 29 and continues till a question mark(?) appears. This is the typical technique for detecting name of every wikipedia article of all the lines starting from index 29 and ends before '?'. The 's' variable in line 5, stores the index of question mark which appears in the readline. So

thisSubject stores the name of the article by retrieving the index from 29 to s, on the line read currently.

There is an if condition(line 7) checking if the line read has the same article's name(which is termed as 'subject' of the Wikipedia article) as compared to the line that is read currently, if they both are same then this line is written on the same file as the triples with same article name should be in the same file . Otherwise a new file is created with the name equivalent to the thisSubject (name of current article) on append mode allowing more lines to be added into the file in upcoming iterations. These files are created under a folder called Context. Again there is an encoding of UTF-8 just to allow some special characters which might be necessary to write within that file.

You could find the try except keywords (Lines 10-14) within the else condition because Windows doesn't support certain file names that includes characters like '/', '*', ':', '?', '|' as these characters are reserved in Windows. Also there are some reserved keywords - CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, and LPT9. However, these names and characters are very unlikely to appear as the subject of wikipedia artice, just to prevent the stoppage of execution of the source code in case of these reserved keywords appear I had added the try block. If these keywords or special characters appear in thisSubject then the file isn't created and basically the read line isn't written anywhere. Again, there could hardly be 1000 files that has these characters or keywords out of the 5 million overall files.

So the triples belonging to one subject are written within one ttl file saved in the name of the subject of the article. I think this is a convenient way of storing, so that in the future it could be accessed easily. However, there is an issue to access the folder containing millions of small files as it takes a lot of time to detect the files and eventually open them. But there is not a lot of option other than waiting for it to open the files in a while.

As discussed earlier about RDFLIB, it has a subpackage Graph for parsing the triples. An RDF Graph is a set of RDF triples, and I try to mirror exactly this in RDFLib, and the graph tries to emulate a container type. This above snippet parses the ttl file that is generated. On parsing one of the wikipedia articles 'Animalia(book)', followed by segregating and iterating over its subject, predicate and object displays the output as :

```
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 "Animalia is an illustrated children's book by Graeme Base. It
      was originally published in 1986, followed by a tenth anniversary
      edition in 1996, and a 25th anniversary edition in 2012. 'text of
      Animalia (book) article goes on....'"
```

Every generated ttl file within Content has 6 triples inside it. They all follow the same pattern. I will explain the above output with each point corresponding to each line respectively:
1) A namespace named "nif" is assigned with the URL. This prevents from repeating the URL everytime within the triples. Instead of repeating the URL, the word nif could be used which is short and conserves lot of space and therefore data. In huge files, this saves space in a huge manner.
2) Another prefix named rdf for the same reasons as nif.
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) The begin index of the content for this wikipedia
4) Informing that this triple is a part of NIF context file
5) The end index of the content for this wikipedia article
6) The content of this wikipedia article is in English
7) The URL for accessing this wikipedia article on a browser with predicate SourceURL.
8) The actual content of this article always appears with predicate in nif:isString.
This sums up the first step of my thesis work. Let's move to the next section.

3.4 Sentence splitting

In this section, we would be discussing the second step by focussing on performing the sentence split for each of the files in the content folder created above. According to Merriam Webster a sentence is "a word, clause, or phrase or a group of clauses or phrases forming a syntactic unit which expresses an assertion, a question, a command, a wish, an exclamation, or the performance of an action, that in writing usually begins with a capital letter and concludes with appropriate end punctuation, and that in speaking is distinguished by characteristic patterns of stress, pitch, and pauses".

I need to separate the sentences in the article which is a NLP task in order to perform tokenization and part of speech tagging later on. We could say that it is a processing step for finding part of speech, for every

word in the article. Sentence separation is to split a given sequence of text (which could be a paragraph or sections or combination of paragraphs within a article) into sentences, by identifying the sentence boundaries. The sentence boundary in most of the cases is full stop, however the sentence splitting is not all that simple. The general challenges regarding the sentence splitting can be found on section 2.1.4 under NLP tasks used in the thesis.

3.4.0.1 Implementation

Lets take a look at the snippet of python script for performing the sentence split and later I would try to explain how it works:

```
1 import nltk
2 from rdflib import Graph,term
3 from rdflib.namespace import RDFS,RDF, FOAF, NamespaceManager
4 nif=rdflib.Namespace("http://persistence.uni-leipzig.org/
    nlp2rdf/ontologies/nif-core#")
```

These are the imports required and RDFLib provides mechanisms for managing Namespaces. In particular, there is a Namespace class that takes as its argument the base URI of the namespace. So whenever I use 'nif' inside a triple, it means that "http://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core". Moving ahead in the implementation part:

```
1  graph2=rdflib.Graph()
2  name="Animalia_(book)"
3  graph2.parse(name+'.ttl',format='nt')
4  g=Graph()
5  s=graph2.serialize(format="nt")
6  count=0
7  for s,p,o in graph2:
8      if type(o)==rdflib.term.Literal and nif.isString in p:
9          sentences = nltk.sent_tokenize(o)
10         for i in sentences:
11             try:
12                 BI=o.index(i)
13                 EI=o.index(i)+len(i)
14             except:
15                 pass
16             )
```

I parsed one of the individual ttl file(line 3), generated from the previous step. named "Animalia(book)" which corresponds to a wikipedia article. Then another empty graph is created to store the output of sentence-split in triples format. And now we split the graph into triples and iterate over

```
    =2016-10&nif=sentence_"+str(BI)+"_"+str(EI)),nif.endIndex,rdflib.term.
    Literal(str(EI))]
g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
    =2016-10&nif=sentence_"+str(BI)+"_"+str(EI)),nif.anchorOf,rdflib.term.
    Literal(i)])
g.add([rdflib.term.URIRef("http://dbpedia.org/resource/"+name+"?dbpv
    =2016-10&nif=sentence_"+str(BI)+"_"+str(EI)),nif.referenceContext,
    rdflib.term.URIRef("http://dbpedia.org/resource/Animalia_(book)?dbpv
    =2016-10&nif=context")])
g.bind("nif",nif)
print(g.serialize(format="turtle"))
print(g.serialize(destination=name+"sentencesplit.ttl",format="turtle")
```

Here, nif.Sentence implies http://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core#Sentence since nif is a namespace created above. And similarly RDF.type is http://www.w3.org/1999/02/22-rdf-syntax-ns#type . I don't need to define RDF namespace as I could directly import it from RDFlib.Namespace . The 5 triples appear for every sentence in a particular article and implies the following:
1) This generated file contains sentences and details about every sentence
2) The begin index of the sentence
3) The end index of the sentence
4) The content of the sentence
5) This is derived from NIF-context file

And finally I store the output in a serialized form on a ttl format file. Binding the graph is to have the name nif for the namespace nif. Otherwise, python automatically gives the name for the namespace we create mostly as ns1. We cannot access it as nif.Sentence as it is ns1.Sentence, to overcome this confusion it is better to bind it by giving your preferable name to the namespace. Before wrapping up this section, I just want to show how the output file looks like. The following is taken from Animalia(book)_sentencesplit.ttl file :

```
<http://dbpedia.org/resource/Animal?dbpv=2016-10&nif=sentence_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> .
```

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, it might appear in a different order and is perfectly fine.

3.4.0.2 Challenges

As far as the challenges are concerned, NLTKs's sent_tokenize function is good enough to detect the salutation, initial of people etc. Let me display

one of the sentences from Animalia_(book)-sentencesplit.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."
And also this nltk package can detect question marks as the end of the sentence and few other punctuations as well..

However, This is supposed to be a single sentence -
"Each creature (A is for alligator, B is for butterfly, etc.) is unique."
but it appears 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 would only increase the complexity of the program as this needs to be checked in every loop. Thus I have not added any additional condition to solve this problem.

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 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.

3.5 Tokenization

As the name suggests, tokenize means breaking into individual linguistic units. Given a string or a paragraph or a sequence of characters and a defined document unit, tokenization is the operation of cutting it up into pieces, called tokens , perhaps at the same time throwing away certain characters, such as punctuations. Tokens are individual linguistic units which are obtained once we tokenize text. Word tokenization is the process of splitting a large sample of text into words. This is a requirement in natural language processing tasks. Usually word tokenization is done by separation of spaces. The general challenges regarding the tokenization can be found on section 2.1.4 under NLP tasks used in the thesis.

3.5.0.1 Implementation

I made use of the same libraries as used for sentence split. Here I will be importing word_tokenize from nltk additionally. Also string needs to be imported for various purposes. And the next part is to parse the graph of the Animalia_(book).ttl .Then serialise it followed by creating an empty

graph to store the triples in order to write the triple into a new file. All these are pretty similar to the first few lines of sentence-splitting. Let's see the function that tokenises when a string or a sentence or a paragraph is given to it as the input.

```
1 def spans(txt):
2     tokens=nltk.word_tokenize(txt)
3     offset = 0
4     for token in tokens:
5         offset = txt.find(token, offset)
6         yield token, offset, offset+len(token)
7         offset += len(token)
```

The function spans here takes any string as input. It uses the word_tokenize package to separate the string into individual tokens. Let's analyse this word_tokenize in detail and how I overcame the challenges, after the explanation of the code. The result of the individual tokens from the given string is stored in the form of a list on the variable 'tokens' (line 2). The offset variable is used to keep track of the index that it is reading. So the index of any string starts from 0, so it is assigned 0. The next part iterates a loop for every token present in the list of all the tokens in order to keep track of the begin and end index of each token. The find operation tries to search for the first occurence of the token from the index that is equivalent to offset. And the index where it finds the first occurence is assigned to offset. This allows retrieval of correct index of every token even if a particular token appears multiple times within the list of tokens.

The idea is that the first token will have the index o to the length of the token and second token from length of the first_token+1 to the length of the second token and so on. So,here this function 'spans' will return the token, begin index and end index of all the tokens in the input text. From line 6, it is evident that it returns token, offset(begin index) and offset+ length of the token(end index) . And finally the offset is incremented by the length of the token.
Let's continue further on the implementation part:

```
1 for s,p,o in graph2:
2     if type(o)==rdflib.term.Literal and nif.isString in p:
3         sentences = nltk.sent_tokenize(o)
4         for i in range(len(sentences)):
5                 print(sentences[i])
6                 try:
7                    content of try block
8                 except:
```

```
9                    pass
```

The graph2 is the parsed Animalia_(book).ttl file. It is iterated over its subject,predicate and object and if condition detects the text part of Animalia_(book) wikipedia article which is same as it was implemented for sentence splitting. The variable 'sentences' stores all the sentence in the form of a list. Moving ahead, yes the sentence split is done. Followed by in line number 4, we have a loop iterating over every sentence in the list of all sentences. The print statement displays every sentence on the output window in python IDE.
The content of the try block:

```
1 BII=o.find(sentences[i])
2 for token in spans(sentences[i]):
3     print(token)
4     assert token[0]==sentences[i][token[1]:token[2]]
5     BI=BII+token[1]
6     EI=BII+token[2]
7     print(str(BI)+str(EI))
8     if token[0] not in string.punctuation:
9         <add the triples to the new empty graph>
```

The try block attempts to find the begin index of the sentence, it is highly unlikely that two sentences in a wikipedia article would be exactly same. So the find function returns the begin index of first occurence of the sentence that is currently being iterated and stores in the variable BII. A nested loop in line 2 is iterating for every token within the current sentence which is sentence[i] placed as a input parameter for spans function. The spans function would return all the tokens in the current sentence along with their begin and end index. Token is displayed (Line 3) where it is a list for eg- (book,9,12) where the token is "book"(token[0]) with begin index "9"(token[1]) and end index "12"(token[2]).

The 'assert' keyword helps detect problems early in your program, where the cause is clear, rather than later as a side-effect of some other operation. We check if the token returned by spans function exist on the given begin and end index of the sentence currently being iterated. Also this is the reason why this piece of code is within a try-except block, if the assert condition fails, it just passed to next token without any interruption. The index of the token in a wikipedia article needs to be calculated as the begin and end index obtained from spans function is confined to the particular sentence. In order to calculate the index of token wrt to the entire article, adding the begin index of sentence to the begin index of the value returned by spans function would give us that. Same is applicable for end index as well (Line 5,6).

String is imported to this script at the beginning which allows to use string.punctuation -includes String of ASCII characters which are considered punctuation characters in the C locale. In line 8, the if condition checks if only a punctuation exist as an entire token returned by spans function for eg '.', then we prevent it from writing into the final output file as it isn't a valid token. However, we need it to increment its length in begin and end index. So I process the punctuation token and I just prevent it from writing into the output file. This is a good way to sought out lot of problems.

Finally, the triples should have the following information in order to store it in a structured manner:
1) The following set of triples correspond to a token which in this case is a word.
2) The word should be displayed with the predicate as nif:anchorOf.
3) The begin index of the word.
4) The end index of the word.
5) The reference of where this is extracted from.

A sample of the output file with added triples for one token looks like this:

```
1 <http://dbpedia.org/resource/Animal?dbpv=2016-10&nif=word_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> .
```

3.5.0.2 Challenges

The general challenges regarding the tokenization can be found on section 2.1.4 under NLP tasks used in the thesis. However, lets see the challenges concerned to my task in this section.

1) **Separation of stop words** such as full stops, comma, open brackets and few other punctuations from the token. If a text "Base." is given as input to the spans function, then it separates "Base" and "." and gives output as two different tokens. Therefore, this function basically separates punctuation if it appears at the END or BEGINNING of a token. Here, however "." isn't a token. So, while writing the output triples I have an "if condition" checking if an entire token is equivalent to a punctuation. If so, I don't write the triple into the output file. I didn't ignore it at the very beginning because I want to keep note of the Begin and end index of punctuation as well. So, I ignored it right as the end just before writing into the output file.

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 Tweet tokenizer we could eliminate the "/" in between the words and consider them as two different token. Also "children's" would be considered as single token in Tweet tokenizer. So, Tweet Tokenizer gives better results in terms of separating tokens more accurately instead of word_tokenize. I have used the tokenize function from TweetTokenizer() in my final source code after initially testing with word tokenizer.

4) **Reading all the files within a folder** - In my case I need to read all the files within the Context folder. So it is bit tricky as an extra loop needs to be added as the outermost loop for it to iterate over every file within a given particular folder. The following piece of code was used in my python program for reading all the files from Context folder and generate tokenization for all of them.

```
import os
for filename in os.listdir('Context/'):
        name=filename.split(".")[0]
        <code for tokenization>
        <store in Tokenization folder>
```

The entire tokenization source code appears within loop(ofcourse excluding the imports and other functions that need not be called everytime). The filename has the extension in it, so in order to strip the extension I have used a split function and retrieved the part before the full stop basically.

3.6 Part of speech Tagging

This section is about creating a piece of software that reads text and assigns parts of speech to each tokens or words it detects, such as noun, verb, pronoun, etc., although NLTK uses POS tags like NNP for 'Proper Noun' or NNS for 'noun-plural' etc.

We need to find the part of speech of every token in order to provide new links only if both the word's part of speech matches. Let's say a word " work " appears more than a billion times in the wikipedia articles. Example- "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. So, depending the part of the speech the new link assigned will be varied.

3.6.0.1 Implementation

From the implementation's perspective, it is same as Tokenization until separating the words in a paragraph. I need to find the Part of speech for

each of those tokens. Before I start with the implementation of POS, there is a pre-processing step which needs to be done. The output of the POS tagger for a token appears as "NNP" or "PRP". Here, NNP signifies Noun, plural . PRP - Personal Pronoun. So we need to map the output of POS tagger to their corresponding expansion as it is hard to interpret that PRP corresponds to Personal Pronoun.

So, the result of the pre-processing should contain the mapping. I had created an excel sheet with 3 columns - The first column should contain all the possible short form outputs from the NLTK pos tagger. The second column should consist of the corresponding full forms like Noun, verb, adjective etc. The third column should contain the type like singular, plural or present or past etc.. It could be done manually by referring the documentation of the POS tagger in python's official website. Let's see a sample of this file :

POS	Fullform	type
NNP	Noun	ProperNoun
JJ	ADJ	Adjective
VBZ	Verb	VerbSingularPresent
PRP	Pronoun	PersonalPronoun
NNS	Noun	PluralNoun
VBD	Verb	VerbPastTense

The first column "POS" is the result that I get from NLTK POS tagger, which is mapped to Fullform and Type. Most of the changes that needs to be done are within the try block on the python script of tokenization, to obtain the part of speech. Let's see the snippet of the implementation:

```
import pandas as pd
data=pd.read_excel("pos–mapping2.xlsx")
for s,p,o in graph2:
    if type(o)==rdflib.term.Literal and nif.isString in p:
        sentences = nltk.sent_tokenize(o)
        tokens = [nltk.word_tokenize(sent) for sent in
            sentences]
        tagged = [nltk.pos_tag(sent) for sent in tokens]
        for i in range(len(sentences)):
                count=0
                try:
                        print(tagged[i])
                        <body>
                except:
                        pass
```

I had used pandas to import the excel sheet for identifying the full form of part of speech. The sentences are taken in order to find the POS as it gives more accurate result of Part of speech rather than searching for an individual word(Line 6). The part of speech changes for a word depending upon which context it is used. The 'tagged' variable in Line 6 consists finds the part of speech for every token in a sentence and stores in a 2-D nested list. Each sentence is a list within which each token's POS is in a list. Iteration of every sentence takes place and at the beginning of every sentence count is assigned 0. The variable 'count' is used to keep track of number of tokens in a sentence.
Just to depict how tagged variable stores the content:
tagged[i] should display the list of all words in the particular sentence along with their POS tags like this - [('Over', 'IN'), ('three', 'CD'), ('million', 'CD'), ('copies', 'NNS'), ('have', 'VBP'), ('been', 'VBN'), ('sold', 'VBN'), ('.', '.')].
tagged[i][count] must display the pair of word with POS for the particular token, When count=0 -('Over', 'IN') when count=1 ('three', 'CD') and so on.
tagged[i][count][1] should display only the part of speech, when count =0 - ('IN') and when count = 1 - ('CD') and so on..
The body of try block has the following content which is focussed on how to convert the POS obtained from the POS tagger (NLTK library) to the human readable full form and it's type :

```
1  for token in spans(sentences[i]):
2          val=data.posfullform1[data.posshortcut1==tagged[i][
               count][1]]
3          for jumbo in val:
4          posfullform=jumbo
5          POSFF="http://purl.org/olia/olia.owl#"+ posfullform
6          value=data.posfullformtype1[data.posshortcut1==tagged[
               i][count][1]]
7          for jumbos in value:
8          posfullformtype=jumbos
9          POSFFT="http://purl.org/olia/olia.owl#"+
               posfullformtype
10         if token[0] not in string.punctuation:
11         <add the triples to the new empty graph>
12         count=count+1
```

The iteration for every token within a sentence is performed to retrieve the POS of every word. The indexes are found in the same way as tokenization. The variable 'val'(Line 2) attempts to find the corresponding POS full form from the excel sheet for the short form obtained from

tagged[i][count]. Here the name 'posfullform1' and 'posshortcut1' are the heading in the excel sheet 'pos-mapping2.xlsx'. Let's say if there is a match for tagged[i][count][1] and the column posshortcut1 , then variable 'val' gives output like this -
"9 Verb
Name: posfullform1, dtype: object"
Here, 9 is the index in the excel sheet where a match is found. The corresponding full form is a verb. It displays the name of the column where it obtained value from and also the datatype of that particular column. However, we just need the information of Verb from these two lines. In order to retrieve it, there is a nested loop run in the val to obtain only the POS and assigning it to the variable 'posfullform'. Printing 'posfullform' would display 'verb' for the example.

In a similar fashion, I retrieve the type of POS from the excel sheet for the corresponding short form obtained from NLTK POS tagger. The variable 'posfullformtype' stores the type of POS. The variables 'POSFF' (Line 5) and 'POSFFT' (Line 8) stores the URL form for writing the output into triples and storing them in the .TTL file.

If the token is a punctuation then they aren't written into the output file. Finally at the end of the loop the count is incremented to obtain the POS of next token in the next iteration. Let's take a look at the sample of one token from the output file:

```
1 <http://dbpedia.org/resource/Animalia_(book)?dbpv=2016-10&nif=word_0_8> a
      nif:Word ;
2     nif:anchorOf "Animalia" ;
3     nif:beginIndex "0" ;
4     nif:endIndex "8" ;
5     nif:oliaCategory <http://purl.org/olia/olia.owl#Noun>,
6         <http://purl.org/olia/olia.owl#ProperNoun> ;
7     nif:oliaLink <http://purl.org/olia/penn.owl#NNP> ;
8     nif:referenceContext <http://dbpedia.org/resource/Animalia_(book)?dbpv
          =2016-10&nif=context> .
```

All the triples have the same subject. The triples that were added to the output files are as follows:
1) Lines (1-4) are same as the output of tokenization file.
2) Line 5 informing the full form of the part of speech for the anchor "Animalia" as "Noun" (at the end of URL) under the predicate OLIA Category.
3) Line 6 informing the type of POS, that it is a "ProperNoun" under the same predicate as OLIA Category.
4) Line 7 displaying the short form obtained from the output of NLTK pos tagger under predicate Olia Link.
5) Line 8 to give the reference of where this data is obtained from.

There is a structured way of storing the triples - The begin index of anything should have nif:beginIndex as predicate. Similarly, the POS should have predicate as nif:oliaLink . The list of all the predicates are found in "http://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core/nif-

core.html#d4e436" . So while storing the triples , I should give appropriate predicates to be added to maintain standard form which have been followed.

3.6.0.2 challenges

The general challenges regarding the Part of speech tagging can be found on section 2.1.4 under NLP tasks used in the thesis. The main challenges that I had faced with POS tagging in my thesis are :

1) Finding the part of speech by using sentence as input instead of individual token. If we give input as individual token, the NLTK POS tagger returns the POS that has higher confidence value. But ignores the context it is used. The confidence value is given by calculating the number of times the word appears as the given POS divided by the total number of times the word used in wikipedia articles. For example "They refuse to permit us to obtain the refuse permit" , if I provide every token individually to NLTK 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 displays both as VERB because confidence value of verb is higher for refuse as in most cases refuse is used as verb. But this is not the correct result in our case. So, I had overcome this challenge by providing sentences as input to the NLTK POS tagger.

2) The part of speech for "." is given as "." which is incorrect as punctuations don't have any POS. So, I just prevented this from writing into the final output file. If an entire token is a punctuation, it automatically assigns the POS as the same punctuation by default which is quite weird. Also, anyways this step was skipped in the Tokenization phase as well.

3) I tried to run this with Pyspark by increasing the number of cores to 2. However, it took more time than running normally. So, I didn't use this implementation. However, the runtime for processing one file is approx 0.1 sec for each file because there is also search operation going on the excel sheet.

3.7 creation of links dataset

This is a pre-processing step for enrichment of additional links on wikipedia articles. I want to have all the words that has link to other article in a CSV file. When attempting to add new links in a article, the words in the article are searched on this CSV file if they exist. Well, if they exist, then this word is given the link. That is the reason for this pre-processing step. NIF link dataset(refer section 2.1.4.3) is extremely huge when unzipped it is of

60GB. I need "surface_form- Link-count" from this dataset and store the result in a .CSV file. First column should consist of all the words that has link to some other page in all the wikipedia articles(The words that have link to another page is called as surface forms, basically the 'word' which when clicked redirects to another page). So, first column should have all the surface forms. The second column should consist the link to the page that will load when surface form is clicked. So, the second column should consist this URL link to the new page. The third column basically is the count implying the number of times surface form and its corresponding link repeats throughout all the articles in wikipedia. This is just a single csv file generated and named surface-forms.csv. I need the results of this task to perform the enrichment of additional links(section 3.8).

It is problematic for performing search operation in .TTL file as opposed to .CSV file which explains why we need to store the result in csv. As far as the implementation is concerned, need to parse the NIF text links file as a Graph. Then create a new excel sheet using python. I used "xlwt" for creating an empty excel sheet. I wrote the headings for the three columns in the excel sheet namely Surface_form, Link, count respectively. Let's take at further part of implementation:

```
1  for s,p,o in g:
2      if "http://persistence.uni-leipzig.org/nlp2rdf/ontologies/
           nif-core#anchorOf" in p or (pref in o and "http://
           persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core
           #taIdentRef" in p):
3          if i==0:
4              word=o
5              i=1
6          elif i>0 and pref in o:
7              link=o.replace(pref,"")
8              i=0
9              sheet1.write(row,0,word)
10             sheet1.write(row,1,"https://en.wikipedia.org/wiki/
                   "+link)
11             sheet1.write(row,2,"1")
12             counter=counter+1
13         else:
14             i=2
15             word=o
16     row=row+1
```

There are 7 triples for each word in the links dataset, however only two are needed for us to process. The other lines or triples are ignored by the if condition in line 2. We need the triple which contains the word itself and we need the triple that gives the link to another page. Pref is assigned

a URL "http://dbpedia.org/resource/" and we check if this is contained in the object to make sure that the new link is a wikipedia article as I am not interested in links pointing to other websites(you could find links to other websites in the External Links section in most of the wikipedia articles.

The object of this predicate: "http://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core#anchorOf" is the surface form which needs to be stored in first column. Initially i,counter is assigned to 0. The word and link are assigned empty string in the beginning. However, there are two ways to enter inside this condition. So we need to consider the other possibility of entering via pref in o and taIdentRef in p, so before exiting "i==0" nested if condition, we make i=1. Similarly while in the condition i>0 and pref in o , we assign i=0. The link and word are retrieved for a set of 7 triples which are both written inside the excel sheet. The third column is given "1" by default for all the entries, later they can be summed if two rows have same surface form and Link. I don't need "http://dbpedia.org/resource/" in Link because in order to access this new page on a browser it has to be "'https://en.wikipedia.org/wiki/'+link". And finally the row is increased by one to write into next row in the excel sheet.

It's time to go through the snippet for summing the third column on surfacegroup.csv to find the count of how many same surface groups and links occurs in entire wikipedia article list. The count could also helps to give correct links on a new dataset because a particular surface-form is more likely to have a link that has higher confidence rate. Higher confidence rate is the one with higher count value. Let's say if a surface form has more than one Link, then the Surface-form, Link pair with highest count is given higher priority when assigning link to this word if it appears in a new article.

```
1 import pandas as pd
2 dataframe = pd.read_csv('surfaceforms.csv', names=['Surface{\_
      }form', 'Link', 'count'])
3 (dataframe.groupby(['Surface{\_}form','Link'])['count'].sum())
      .to_csv('surfaceform.csv')
```

I have used pandas for this implementation as it makes the code extremely simpler. After reading the file, I try to store it as a data frame. Pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). The groupby function allows me to check the rows that has the similar pair of surface form and Link. If there are multiple rows with the same pair of Surface form and Link, then I summed their count and merged the multiple rows into a single row. There are lot of merges happening and the final output is stored in a new file "surfaceform.csv". The output file looks like this:

Surface-form	Link	Count
Phil McGraw	https://en.wikipedia.org/wiki/Phil_McGraw	116
Ngapali	https://en.wikipedia.org/wiki/Ngapali	7
Kalidasa	https://en.wikipedia.org/wiki/Kalidasa	283
Wuhan	https://en.wikipedia.org/wiki/Wuhan	1243
Anglin	https://en.wikipedia.org/wiki/Anglin	24
Anglosphere	https://en.wikipedia.org/wiki/Anglosphere	341

Another approach

There is an another approach I tried which I would like to mention in my thesis. With the NIF text links file, I retrieved the triples of an individual wikipedia article. Similar to what I did with the NIF context file in the beginning, I did the same with NIF text links file. This gives a set of 7 triples for each link in the wikipedia article. From this I have the begin and end index of the link, the actual word which when clicked goes to a new page, the URL of the new page (where the link points to). I created an excel sheet containing surface-form, Link, Part-of-speech for individual wikipedia article. Ofcourse I didn't do it for all the wikipedia articles, I had done it for only few files to test and see if it gives good results. Obtaining the part-of-speech is really helpful because two words having same content and having same part-of-speech are more likely to have same link rather than choosing the link that has higher count for two words with similar content.

Looking at the implementation of this approach, first an empty excel sheet needs to be created. The column names needs to be defined manually. For simplicity, I named them Surface form, POS and Link. After which the parsing of the newly created links ttl (consisting of links in individual wikipedia article, basically a subset of NIF text links file). Later extraction of the name of the wikipedia article from the subject of newly created links file is mandatory. This allows to load the corresponding part-of-speech file for this wikipedia article. Now the goal is to find the POS by parsing that file with the available begin and end index along with the anchor. The final result was however stored in a xlsx file itself. So, this is how the implementation is done:

```
1 for su,pr,ob in graph1:
2       if "http://dbpedia.org/resource/"+name+"?dbpv=2016-10&
          nif=word_"+str(BII)+"_"+str(EII) in su and "phrase"
          not in s:
3           if nif.oliaLink in pr:
4                   print(ob.split("#")[1])
5                   sheet1.write(row,1,ob.split("#")[1])
```

```
6       if "http://dbpedia.org/resource/"+name+"?dbpv=2016-10&
            nif=word_"+str(BII)+"_"+str(EII) in su and "phrase"
            in s:
7           if nif.oliaLink in pr and wordcount==1:
8                   print(ob.split("#")[1])
9                   sheet1.write(row,1,ob.split("#")[1])
10          row=row+1
11  book.save(name+'surfaceforms.xls')
```

The graph1 is the parsed version of <name-of-the-article>-pos.ttl file. Inside the loop every triple is checked if they have a subject corresponding to the one in <name-of-the-article>-links.ttl file. The 's' in line 2 is the subject of links file. So, if both the subjects matches next thing is to find out if it is a word or phrase. In the POS file, we have part-of-speech for individual tokens, so if it is a phrase I need to break down the phase into individual tokens before looking for POS. The POS is always stored like this in ttl format:

```
1  nif:oliaLink <http://purl.org/olia/penn.owl#NNP> ;
```

In line 4, "ob.split("#")[1]" means that I split the object to retrieve the part after hash tag which in the above case would give me "NNP" which is exactly what I need. The predicate "nif:OliaLink" has the POS which prevents reading other unnecessary triples , the reason behind having an if condition on Line 3.

The retrieval of POS is considerably simpler if it is a single word. Now lets consider the other if condition which is passed through if a link appears on a phrase(more than one word), the phrase is split into individual words. And a loop iterates over number of individual words in order to attempt looking for their corresponding POS. The variable wordcount is used to keep track of it. The final output is stored as <name-of-the-article>-surfaceforms2.xls . But considering the size of articles, loading every corresponding POS file is very time consuming. It takes infinitely huge time to process although this is expected to give better results than providing a new link by the confidence of the word to link (higher the count, higher the confidence).

3.8 Enrichment of additional links

The final section of the Data Processing part is the final step on the flow-chart. The preprocessing step for this task is to have a CSV file consisting of Surface_Form-Link-Count pairs all over the wikipedia article. There is a possibility that a particular surface-form can have multiple links.
For example : The surface form 'it' has link to 'Information Technology' and the 'pronoun' it on separate ocassions. So, when I read the word 'it'

in a new article, there is a confusion on what link should be given either "IT" or "pronoun". As discussed in the end of the last section there are two ways in which a new link could be provided while reading a new article.
The first approach is :
If the word in the new article matches the 'surface-form' column in the surface-form.csv file (explained in the previous section) more than once, then the link having higher count could be provided for this word (which in most of the cases is likely to be correct but not ALWAYS). This step is easier to process as it isn't really time consuming. This is my initial approach for implementation.

The second approach is :
If the word in the new article matches the 'surface-form' column in the surface-form.csv file (explained in the previous section) more than once, then the link that has same part of speech could be provided which is highly likely to be a correct link. In the example, IT in information technology is a NOUN , however it is PRONOUN. So, this would give a better result in terms of the correctness of the new link provided. However, it is really time consuming as I need to parse the corresponding article's part of speech to find it. I have used this approach of implementation to get better results. I have partially implemented this. This approach is very time consuming because of obtaining POS for every surface-form in my previous task(refer section 3.7 under another approach part).

Now my goal is to enrich more links on the individual wikipedia articles that were generated within the Context folder(refer section 3.3). Within each file, I will read word by word and search if a link is possible for the word, by comparing it with the surface-form.csv file created in the previous section. However from implementation point of view, both the approaches are very similar. Let's look at the implementation:

```
1  stop = set(stopwords.words('english'))
2  flag=0
3  for s,p,o in graph2:
4      if 'http://persistence.uni-leipzig.org/nlp2rdf/ontologies/
           nif-core#isString' in p:
5          count=len(nltk.word_tokenize(o))
6          for i in range(count):
7              test=""
8              a=o.split(' ')[i]
9              a=re.sub(r'[^\w\s]','',a)
10             b=o.split(' ')[i+1]
11             c=o.split(' ')[i+2]
12             string1=a+' '+b
```

```
13              string=a+'␣'+b+'␣'+c
14              csvreader = csv.reader(codecs.open('surfaceform.
                    csv','r',encoding='mac_roman',errors='ignore'))
15              fields = next(csvreader)
16              <search for the string,string1,a in the CSV file>
```

This is the first part of the implementation focussing on the variable creation and segregating the words in order to search in the csv file. The variable 'stop' consists a list of all stopwords in English like 'A' ,'the' etc. We don't need to look for these words to search for a link as they are quite common words. The graph2 is the parsed graph of individual articles from context folder. After extracting the content of the article from the triples in each article, we separate each token in the content (similar to section 3.5). A loop iterates for the number of tokens in each article, the variable 'a' (Line 8) is the first token, 'b' is the next token. In the second iteration 'b' becomes first token and 'c' becomes second token and so on..! Every word after a space is the next token. That is the mindset behind splitting token with space as a parameter.

The re.sub command in line 9 is to remove the fullstop or comma or any other unnecessary punctuations from the token because you cannot find any surface form with '(' or ',' along with the words. So it is better to remove them while searching for a link. The variables string 1 are string are the combination of first two words and first three words respectively. After this, we shall open the surface form csv file for searching if the tokens exist within the file. Once opened, we can iterate over the rows. The variable 'flag' keeps track of the number of words that link is assigned to while reading this articles. Let's go ahead with the implementation on the search technique:

```
1  for row in csvreader:
2      if a in stop:
3          break
4      if flag>0:
5          flag=flag-1
6          break
7      row[0]=re.sub(r'[^\w\s]','',row[0])
8      fword=row[0].split("␣")[0]
9      try:
10         sword=row[0].split("␣")[1]
11         tword=row[0].split("␣")[2]
12     except:
13         pass
14     if fword.lower()==a.lower():
```

```
15          if sword.lower()==b.lower():
16          if tword.lower()==c.lower() and row[0].lower()==string.
                lower():
17          print(row)
18          flag=2
19      if row[0].lower()==string1.lower():
20          print(row)
21          flag=1
22      if row[0].lower()==a.lower():
23              if test=="" or row[2]>test:
24              print(row)
25              test=row[2]
26              g.serialize()
27  counter=counter+len(a)
28  counter+=1
```

Jump out of this loop on two conditions:
1) If the first token is a stop word
2) If the first token is a part of a link from previous iteration(s). Here, row[0] corresponds to the first column on the excel sheet which is nothing but 'surface-form'. I had removed the punctuations in it as well to find the exact match and not cause any ambiguity. The fword (Line 8) is the first word which is available in each row of the surface-form column on the CSV file. There could be multiple words in a surface form which is why it tries to find the second and third word in that column as it isn't sure that they will exist everytime. If there is an exact match for the three words read to first column in surface form, then the corresponding link is retrieved and the flag is assigned 2 meaning the next two words are already a part of this link.

Similar approach if the first two tokens have an exact match on the CSV file. The flag is set one and the corresponding link is stored in the final output. The trickier part is when one word matches on the CSV file. The search operation cannot be halted, it needs to go on trying if another match occurs. The 'test' variable stores the count value of Link provided to a particular token. The idea is that if a Link with better count is found on the upcoming iterations, then the link needs to be modified. The row[2] corresponds to count of surface-form and link pair as discussed in previous section. The final result is however stored in the TTL format. When flag=1, a set of triples are added with displaying for string as the surface form and their corresponding link :

```
1  <http://dbpedia.org/resource/Animalia_(book).ttl?dbpv=2016-10&nif=
       word_27_41> a nif:Phrase ;
2      nif:anchorOf "childrens book" ;
3      nif:beginIndex "27" ;
4      nif:endIndex "41" ;
```

```
5     nif:referenceContext <http://dbpedia.org/resource/
          Animalia_(book).ttl?dbpv=2016-10&nif=context> .
6         nif:taIdentRef "https://en.wikipedia.org/wiki/Children's_Book" .
```

The triples explained (with each point corresponding to the line number):
1) It is a phrase ie. comprising more than one word (Since flag=1 has two words).
2) The word that has the link on it is "Children's book"
3) Begin index of "children's book" in Animalia book article is 27.
4) End index of "children's book"(surface form) in Animalia book article is 41.
5) This content is part of the context file.
6) The link to the new Wikipedia page when "Children's book" is clicked is https://en.wikipedia.org/wiki/Children's_Book.

Similarly when flag is 2, another set of triples with nif:anchorOf equivalent to 3 words is added and it's corresponding link. The surface-form, link pair that has the highest count is displayed irrespective of the flag. When there is a match for 'a' on surfaceforms.csv without having a match for 'string1' then the triple added has nif:Word instead of nif:Phrase and nif:anchorOf has one word on it.

3.8.0.1 Challenges

Repetition of links for the same word in the article. If a surface form repeats 5 times within an article, my script would provide link for the same surface form all the 5 times even though given once perfectly serves the purpose. This leads to more number of links and it is unnecessary.

Not detecting surface forms that has more than 3 words in it. My code checks for an exact match upto 3 words in an article. If there is a possible surface form having more than 3 words, my code doesn't detect it.

Incorrect links provided - For example: There is a surface form "It was" which is a music album. So whenever "It was" is detected in any article, it provides the link to the wikipedia page of that music album which in most cases is likely to be wrong. There is a possibility that a particular surface-form can have multiple links and I choose to give the link that has higher count of surface-form and Link pair. This choice in most of the cases is likely to be correct but not Always. Let me take this as an opportunity to explain my work on the second approach (the link that has same part of speech could be provided rather than the link that has the highest count for the given surface form) which would obviously reduce the number of incorrect links. Instead of count, I have POS for the corresponding surface-form. The surface-forms2.xlxs looks like this :

Surface-form	Link	Count
Phil McGraw	https://en.wikipedia.org/wiki/Phil_McGraw	NNP NNP
Ngapali	https://en.wikipedia.org/wiki/Ngapali	NN
Kalidasa	https://en.wikipedia.org/wiki/Kalidasa	NNP
Wuhan	https://en.wikipedia.org/wiki/Wuhan	NN
Angling	https://en.wikipedia.org/wiki/Angling	VBG
Anglosphere	https://en.wikipedia.org/wiki/Anglosphere	RB

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.

Providing links for words like "The" , "was" etc.. doesn't makes sense. So, I skipped these words from getting links by preventing the search operation on the CSV file, if the word to be searched is a part of Stopwords in english.
I have explained in detail on next chapter regarding the percentage of correct links provided.

Chapter 4

Experiments

4.1 Metrics

The results obtained from the tasks I had performed needs to be analysed. For this purpose there is an F1 score which is used to rate the correctness of my results and could be used as a metric to compare with others works. 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. Lets look at the formulae of the various metrics with which my work could be measured:
1) Precision= TP/(TP + FP)
2) Recall=TP/(TP+FN)
3) F1=(2*Precision*Recall) /(Precision + Recall)

In other words, an F1-score (from 0 to 1, 0 being lowest and 1 being the highest) is a mean of an individual's performance, based on two factors i.e. precision and recall. The F1 score is also known as the Sørensen–Dice coefficient or Dice similarity coefficient (DSC).

I have calculated all these metrics for the enrichment of Links. There is a slightly different method to calculate Precision and Recall for Tokenization and sentence splitting : The correctly segmented words are regarded as true positives (TP). To obtain precision, TP is normalised by the prediction positives (PP), which is equal to total number of words returned by the system. For recall, we divide TP by the real positives (RP), the total number of words in the reference. The complement of RP is referred to as real negatives (RN).

4.2 Evaluation

To determine quality of results 2 experiments are done. First is to determine F1 Score. Second experiment is to determine the quality of results comparing to similar project - Wikifier.

4.2.1 F1 score

For the sentence splitting 5 random output files were taken into consideration.
True Positives(TP) - 268 (Correctly detected sentences in 5 files)
Prediction positives(PP) - 272 (total sentences returned by system)
Real Positives(RP) - 270 (total sentences in the reference)
Precision = 98.5% , Recall=99.25% , F1= 98.9% The error here is that it detects the words ending with 'etc.' and 'Warner Bros.' inside the brackets as a separate sentence, although it should have been one sentence consisting of the content of brackets within it. This issue is discussed in challenges part of sentence splitting.

For the Tokenization 2 random output file were taken into consideration.
True Positives(TP) - 2070 (Correctly detected tokens)
Prediction positives(PP) - 2076 (total tokens returned by system)
Real Positives(RP) - 2082 (total tokens in the reference)
Precision = 99.7% , Recall=99.4% , F1= 99.57% My script has made mistakes in separating words having ':' '#' between them without spaces. Instead of separating them as two different words, my script gives them as one word. This issue was also discussed in the challenges part of Tokenization.

Regarding enrichment of links, one of the files 'Animalia_(book)-newLinks.ttl' was checked manually :

	Positive	**Negative**
True	136	-
False	8	35

Here, the total number of links originally present in this wikipedia article is 22. Out of this 22, 17 were detected correctly (other 5 links had more than 3 words in their surface form, my script detects surface forms with maximum of 3 words). The new links TTL file detected 171 links totally out of which 136 are correct and 35 are incorrect or repetition of links to the words everytime it appears in the article.
Precision = 94.8%, Recall=79.5%, F1= 86.4%

ted by my results as well. In addition 15 other links are provided by my script. Out of the 21 links provided in this paragraph, 4 are repeated and 2 is incorrect ('over','sold'). so 15 correct links provided with 2 incorrect and 4 repetitive links.
Let's take a look at Wikifier F1 score:

	Positive	**Negative**
True	6	-
False	9	0

They have missed out correct links to 9 words in this paragraph. And they have provided 6 correct links.
Precision = 40%, Recall=100%, F1= 57.1%

Let's take a look at my F1 score for this particular paragraph:

	Positive	**Negative**
True	15	-
False	0	6

Precision = 100%, Recall=71.4%, F1= 83.5%

However, wikifier would have wanted to avoid giving links to basic terms, so I don't know their exact motivation behind their project.

4.3 Statistics

The analysis has been performed on 100 files that are provided on the CD for each NLP task.
1) The total number of sentences on the 100 files is 5126. The average number of sentences in a file is approximately 51. It was detected by counting the number of nif.Sentence in each file, the count would give overall number of sentences in 100 files :

```
1 count=0
2 graph2.parse('Sentencesplit/'+filename,format='turtle')
3 for s,p,o in graph2:
4     if "http://persistence.uni-leipzig.org/nlp2rdf/ontologies/
          nif-core#Sentence" in o:
5         count=count+1
```

I have loaded all the files within Sentencesplit(enclosed in the CD). In each file, an attempt to find nif.Sentence is made, if found count is incremented.

2) The total number of tokens in the 100 files is 109206 which means that every article has an average of 1092 tokens. The script used searches

This is the script used to detect the count of the part of speech :

```
1 def in_list(item,L):
2     for i in L:
3         if item in i:
4             return L.index(i)
5 for s,p,o in graph2:
6     if "http://persistence.uni-leipzig.org/nlp2rdf/ontologies/
        nif-core#oliaLink" in p:
7         P=o.split('#')[1]
8         check=in_list(P,show)
9         list-pos[check][1]+=1
```

Here, the variable list-pos(Line 9) is a 2D list which has all possible POS (output from NLTK) and the count is initially set as 0 for all the POS tags. The format in which the list is stored - [[pos-1,count-1],[pos-2,count-2],...]. The functionin_list return the index of the particular POS tag in the list. I detect the position of the particular POS tag in the list and increment its corresponding counter. The variable 'P' in Line 7 stores the POS, extracted after #. The function accepts the list and the POS to be searched as parameters.

Chapter 5

Conclusion and Future work

The master thesis is performed to enrich the DBpedia NIF dataset with additional information. By parsing and combining the DBpedia NIF datasets, I have created separate datasets for Context,Sentence-split, Tokens, POS and Link Enrichment. Results are these 5 folders containing their corresponding data for each wikipedia article.
I would like to list out the results:

- DBpedia NIF datasets are analysed thoroughly
- Already existing approaches for basic NLP operations on wikipedia contents are analyzed. However, it is not possible to get to know the implementation details of such approaches.
- Segregated the contents of individual Wikipedia articles into separate files, as the NIF dataset comprises the contents of all the articles in one huge file.
- Performed basic NLP tasks like sentence splitting, Tokenization and POS tagging on the segregated contents of articles and stored the result of each task in triples.
- Enriched links to Wikipedia articles and stored the result in triples. As an improvement, surface forms occuring multiple times within an article must be given only one link at the first time of occurence.
- Analyzed the results of these tasks statistically and compared results with similar work.

As a future work, better search algorithm for finding surface forms from CSV file could be implemented for link enrichment task and implement complete script for comparing POS of the word in the new article to the POS of surface form, for better results.

Bibliography

[1] DBpedia. DBpedia [online]. April 2019. Available from:
https://wiki.dbpedia.org/about/dbpedia-community

[2] AKSW. DBpedia NIF Dataset: Open, Large-Scale and Multilingual Knowledge Extraction Corpus[online]. Available from :
http://aksw.org/Projects/DBpediaNIF.html

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

[4] Wikipedia. Wikipedia[online]. Available from:
https://en.wikipedia.org/wiki/Wikipedia:Size_comparisons

[5] W3C. RDF [online]. Available from:
https://www.w3.org/RDF/

[6] The Linked Open Data Cloud[Online]. Available from:
https://lod-cloud.net/

[7] DBpedia About[Online]. Available from :
https://wiki.dbpedia.org/about

[8] NIF 2.0 Core Ontology[Online]. Available from:
http://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core/nif-core.html

[9] BabelNet. Available from : **http://live.babelnet.org/**

[10] WordNet. WordNet Search [online]. April 2019, [Cited 2019-04-28]. Available from: http://wordnetweb.princeton.edu/perl/webwn

[11] Extracting and Annotating Wikipedia Sub-Domains — Towards a New eScience Community Resource. Utrecht University Repository. Available from:
https://dspace.library.uu.nl/handle/1874/296810

[12] Annotating documents with relevant Wikipedia concepts[PDF] Authors-J Brank, G Leban, M Grobelnik - Proceedings of SiKDD, 2017. User Interface -
http://wikifier.org/

Appendix A

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

Appendix **B**

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `Scripts` ..... the source code of NLP,preprocessing tasks
  - `Scripts of analyzing tasks` .. Scripts used to analyze the results statistically
  - `thesis` ..... the directory of LaTeX source codes of the thesis
    - `*.text` ..... the LaTeX source codes of the thesis
- `Output` ..... 100 samples of results of the tasks
  - `Content` ..... 100 samples of separation of wikipedia articles
  - `Sentencesplit` ..... 100 samples of sentence splitting on articles
  - `Tokens` ..... Tokenization results for 100 articles
  - `Part of Speech` ..... POS tagging results for 100 articles
  - `Enrichment of additional Links` ..... 100 samples
- `DP_Lakshmanan_Pragalbha.pdf` ..... the thesis text in PDF format