Semantics extraction

Final Project - Tecnologie Semantiche

Group 6

Santella Alessandro 0622701526
Salsano Carmine 0622701527
Tedesco Christian 0622701528
Oro Carmine 0622701529

{a.santella2, c.salsano8, c.tedesco19, c.oro}@studenti.unisa.it

Contents

1 Study of the problem/idea

The project's objective is to extract relevant semantic informations from natural language text. To do so we have used a lexical database known as $WordNet$ and an API to get data from Wikipedia's database known as $Wikifier$.

Since we were dealing with a natural language text, it was necessary to apply some steps of the Natural Language Preprocessing (NLP), such as Coreference solution, Named Entity Linking, Tokenization, Stepwords removal, POS Tagging, Lemmatization, etc. These steps will be shown later.

In order to get the most significant representations of the information, we thought of extracting as many significant triples as possible, focusing both on the subjects and predicates.

A first set of triples that we decided to extract is related to the descriptive classes of each subject that the Wikifier API recognizes automatically from the text. The second set we decided to extract is related to the classic 'Subject-Predicate-Object' structure, so such triples are extracted looking at the text.

Examples of triples that we extracted are:

- **For the first type:** *'Frodo Baggins is a Hobbit'*;
- **For the second type:** *'Ringwraiths search Frodo Baggins'*;

So the system we designed must extract a large number of triples, which are not too long or elaborate, but that carry the most information.

The entire system is available at this link.

2 System overview/architecture

In this chapter we focus on the pipeline that our text will have to go through, describing the main modules of the system and the I/O of each component.

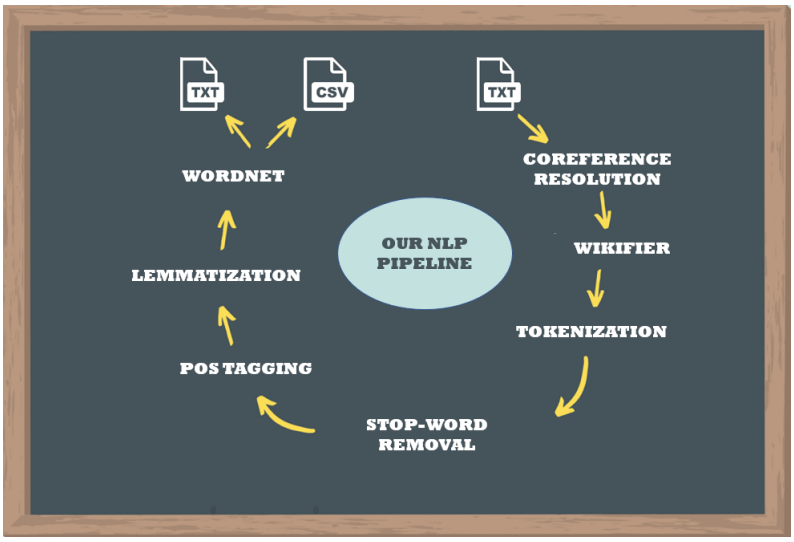

Figure 1: Pipeline

2.1 Coreference resolution

Starting from the loaded text, it will be given as input to the coreference resolution step, whose main goal is to determine linguistic expressions that refer to the same real-world entity in natural language, in order to replace the possessive pronouns with the entities in question. So, at the end of this phase a text will be obtained in which every possessive pronoun is substituted with the proper nouns it refers to.
An example is the following: the sentence of the input text 'Frodo Baggins knew the Ringwraiths were searching for him' becomes the sentence 'Frodo Baggins knew the Ringwraiths were searching for Frodo Baggins' in the output text.

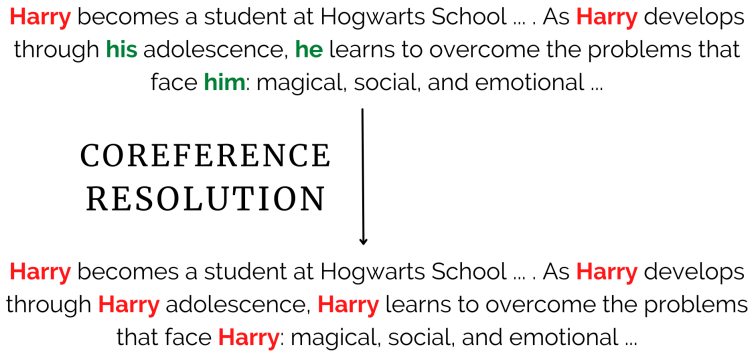

Figure 2: Coreference Resolution

2.2 Named Linking Entity with Wikifier

The coreferenced text is now processed by Wikifier API by an apposite *POST* request in which the entire text is passed. However, this was possible since we knew the text to work on and we knew it was a few lines, but if we had a large document (for example of many pages), it would probably have been more appropriate and efficient to divide it into various parts and analyze the various parts individually. Recalling the patterns of the literature, an example of subdivision of the text could be to take the various paragraphs individually, since in each paragraph a separate topic is explained.

Once the response is obtained, it's possible to build 'isA' triples (and associated URIs) by linking the references found in text with the Wikipedia classes labels associations found by the Wikifier API. It has been decided to use text reference as triple subject since it can happen that the text reference and the associated Wikipedia page have some slightly grammatical differences, and therefore it was preferred to preserve the words used in the analyzed text.

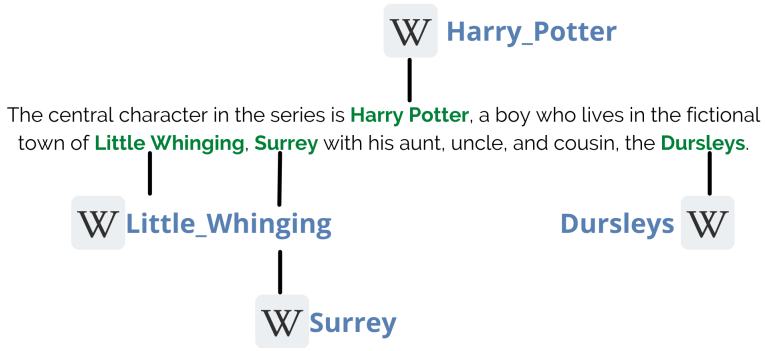

Figure 3: Named Linking Entity with Wikifier

2.3 Sentences Tokenization

Considering the text output from the coreference resolution step, it will be input to the Tokenization step, that is the process of tokenizing or splitting a string/text into a list of tokens.
In this step, the entire text is divided into various sentences (considering the '.' as a separator). Subsequently, every single sentence was divided into tokens, considering every single word as a token.

2.3.1 Merging tokens of subjects

One of the main problems with tokenization is that it splits the sentence into tokens without considering the semantics of them, so for example names like 'New York' and 'Harry Potter' are split into two tokens.
This thing is not good for our system, so to remedy this, we saved a list of main subjects (extrapolated from Wikifier) whose names are composed of more than two tokens and, subsequently, the tokens that are part of it are merged, to regain the subject in a single token.

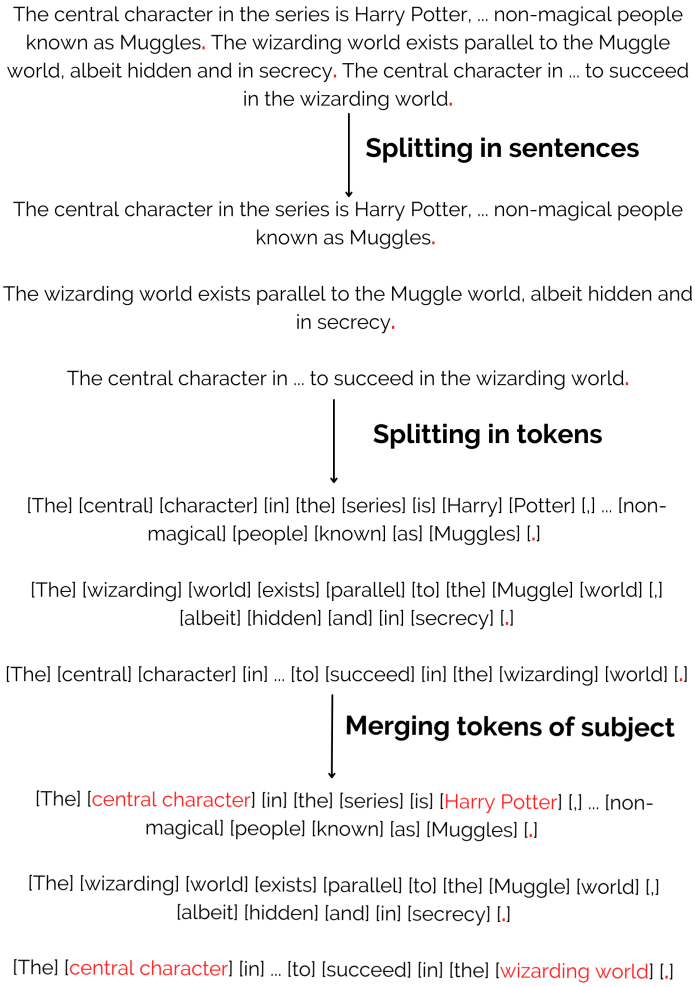

Figure 4: Tokenization

2.4 Stopwords removal

The Stopwords removal step takes as input the sequences obtained from the tokenization step, on which this step is carried out individually. The stopwords removal step involves the removal of the so-called stopwords, i.e. those words in natural language that have a very little meaning, such as 'an', 'the', etc.
Since stop words occur in abundance, hence providing little to no unique information that can be used for analysis, the output of this step will simply be the sentences without the stopwords.

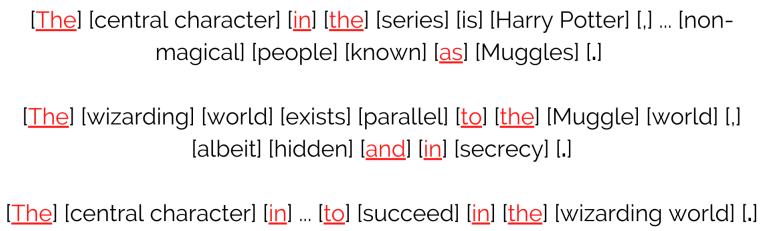

Figure 5: Stopwords removal

2.5 POS Tagging

The POS Tagging step is the process of marking up a word in a text as corresponding to a particular part of speech, based on both its definition and its context. A simplified form of this is commonly the identification of words as nouns, verbs, adjectives, adverbs, etc.

In this case the POS Tagging labels each token but, in order to obtain triples, we add some filtering so as to obtain only the tokens labeled as nouns and verbs. However there are various nuances of labels inherent in verbs and nouns, in our case we have labeled as nouns all the tokens that had as labels 'NN', 'NNS', 'NNP', 'NNPS' and as verbs the tokens that had as 'VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ' labels. The outputs are the labelled tokens for each sentence.

Figure 6: POS Tagging

2.6 Lemmatization

Immediately after the POS Tagging step, there is the Lemmatization step that normally aims to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the *lemma*.
This step takes the sentences with the labels as input and outputs the same sentences but with the words transformed into a lemma, that is the base form.

Figure 7: Lemmatization

2.7 Taking synset from WordNet

Having obtained the phrases so done and almost ready to become triple, we decided to take another step to be able to expand the extraction of the triples. To do that, in our project we also made use of a well-known large lexical database of English, WordNet. In WordNet the nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (*synsets*). Synsets are interlinked by means of conceptual-semantic and lexical relations.

In our project we thought of using synsets to be able to extrapolate as many triples as possible. Specifically, we decided to create new triples starting from the original triples and replacing them with the verb with synonyms. In order to do this, the verb was extrapolated from the triple, the synset associated with it was searched and, subsequently, filtered to obtain only the synonyms that are verbs.
Having obtained the list of synonyms, we create a triple for each of them. It is important to note that we only take distinct synonyms, so we avoid repetition.

So, in this step, we extrapolate a dictionary of synonyms for each verb present. Specifically, we extrapolate the first synonym of each synonym set. Taking the figure 8 into consideration, '*exist*', '*populate*', '*survive*', '*know*', '*be*' will be taken as a synonym for *live*.

Verb

- S: (v) populate, dwell, **live**, inhabit (be an inhabitant of or reside in) *"People lived in Africa millions of years ago"; "The people inhabited the islands that are now deserted"; "this kind of fish dwells near the bottom of the ocean"; "deer are populating the woods"*
- S: (v) **live** (lead a certain kind of life; live in a certain style) *"we had to live frugally after the war"*
- S: (v) survive, last, **live**, live on, go, endure, hold up, hold out (continue to live and avoid dying) *"We went without water and food for 3 days"; "These superstitions survive in the backwaters of America"; "The race car driver lived through several very serious accidents"; "how long can a person last without food and water?" "One crash victim died, the other lived"*
- S: (v) exist, survive, **live**, subsist (support oneself) *"he could barely exist on such a low wage"; "Can you live on $2000 a month in New York City?"; "Many people in the world have to subsist on $1 a day"*
- S: (v) be, **live** (have life, be alive) *"Our great leader is no more"; "My grandfather lived until the end of war"*
- S: (v) know, experience, **live** (have firsthand knowledge of states, situations, emotions, or sensations) *"I know the feeling!"; "have you ever known hunger?"; "I have lived a kind of hell when I was a drug addict"; "The holocaust survivors have lived a nightmare"; "I lived through two divorces"*
- S: (v) **live** (pursue a positive and satisfying existence) *"You must accept yourself and others if you really want to live"*

Figure 8: Synsets of Live

2.8 Finding Triples

This last step is the one that deals with extracting the triples. Starting from the dictionary of synonyms and from the phrases obtained from the pipeline just mentioned, we build the triples (and related URIs) of the 'Subject-Predicate-Object' type.

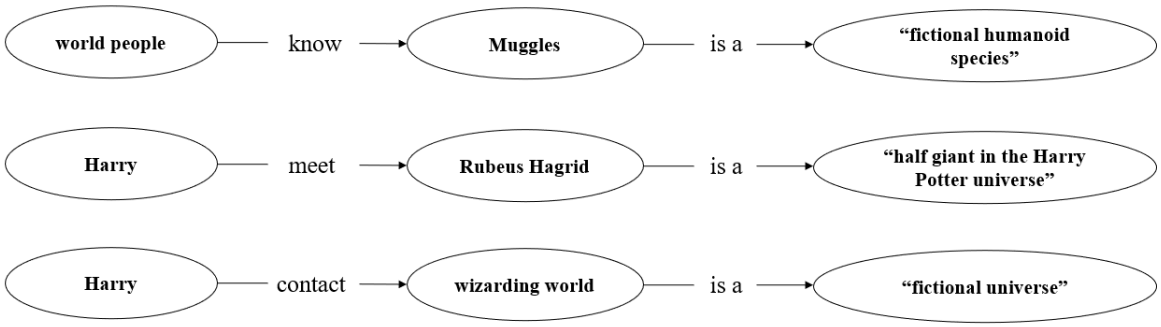

Figure 9: Triples

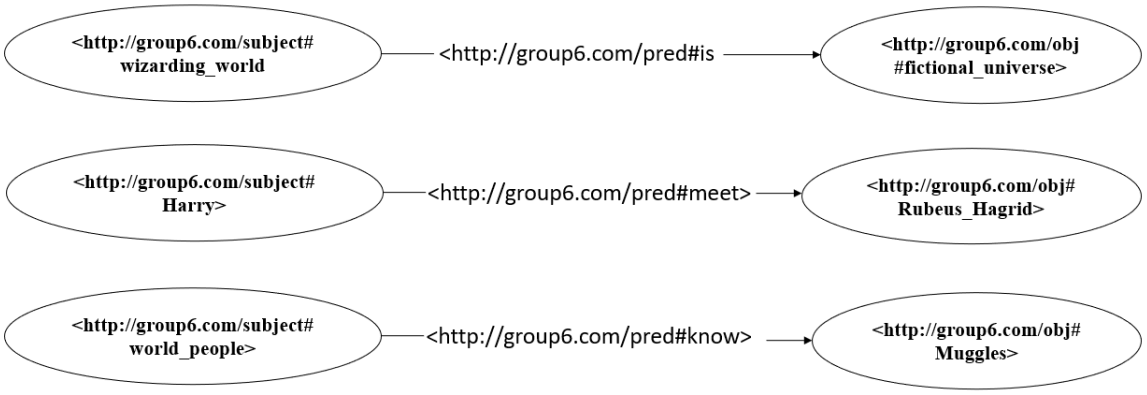

Figure 10: URIs

3 Implementation

In this chapter the focus is on how each step of the pipeline described above has been implemented in code. In particular the choices made and the functions used are illustrated.
Due to the vastness of existing libraries and ease of use, it was decided to use *Python*. In order to share the workspace among the group and to be able to modularize the code, *GoogleColab* was used.

3.1 Coreference resolution

This step consists of loading the English ('*en*') pre-trained spaCy model to the NLP pipeline, adding the Coreference resolution ('*neuralcoref*') module to it, then feeding the loaded text to the pipeline and applying operations on it.
The aforementioned operations are:

- the extraction of tokens with whitespaces from the spaCy document;
- the extraction of tokens from the representative cluster name for each cluster;
- the replacement of each coreference text element with the representative element of the cluster, if they are not equal and none of the coreference element words are in representative element;

However, there may already be errors in this step. In fact, it can rarely happen that single pronouns are substituted with whole sentences or that mistakenly replaces pronouns with subjects they did not refer to. Moreover, sometimes the clusters to which some possessive pronouns belong are not correctly identified, leading to further errors. It has been decided to accept this rare wrong cases, since in the most of times the replacing is done correctly.

```python
import spacy
import neuralcoref

# Load SpaCy
nlp = spacy.load('en')

# Add neural coref to SpaCy's pipe
neuralcoref.add_to_pipe(nlp)

#Function that executes coreference resolution on a given text
def coref_resolution(text):

    doc = nlp(text)

    # fetches tokens with whitespaces from spacy document
    tok_list = list(token.text_with_ws for token in doc)
    for cluster in doc._.coref_clusters:

        # get tokens from representative cluster name
        cluster_main_words = set(cluster.main.text.split(' '))
        for coref in cluster:
            if coref != cluster.main:

                # if coreference element is not the representative element of that cluster
                if coref.text != cluster.main.text and bool(set(coref.text.split(' ')).intersection(cluster_main_words)) == False:

                    # if coreference element text and representative element text are not equal and none of the coreference element
                    # words are in representative element. This was done to handle nested coreference scenarios
                    tok_list[coref.start] = cluster.main.text + doc[coref.end-1].whitespace_
                    for i in range(coref.start+1, coref.end):
                        tok_list[i] = ""

    return "".join(tok_list)
```

Figure 11: Coreference resolution code

3.2 Named Linking Entity with Wikifier

As mentioned before this processing step is performed using the $Wikifier$ API. It consists of:

- preparing the URL for the API call (specifying the text, the language and other parameters useful to $Wikifier$);
- calling the API through a $POST$ request and reading the response;
- building 'isA' triples (and associated URIs) having as subjects the titles of the API responses (subjects recognized by $Wikifier$), 'is' as verb and as objects each one of the $Wikipedia$ classes obtained from the call;

Analyzing the results obtained with Wikifier, we noticed that many triples are extrapolated for each entity, but that some of them are very forced. However, not having available a threshold to be able to modify the degree of acceptability of the triple, we had thought of selecting the best triple a-posteriorly.

Looking at the triples, we noticed that for some subjects (the most "popular" ones) the acceptable triples are almost all, while for the less popular ones some of them should be discarded. But, in order to do this, we should do a specific analysis on the triples obtained from the specific text and, therefore, specific domain, but in doing so we lose the much desired automation of the process. Therefore, it was taken into account to accept this trade-off, also taking some "forced" triples but still leaving the total automation of the process.

A similar approach was applied in the step of manual extraction of the triples made after the NLP, in which we divide the text into sentences (each point identifies the end of a sentence) and we analyze each sentence individually, in order to extrapolate the triples to them. associated, and then only at the end put together all the triples thus obtained.

```
import urllib
from string import punctuation
import nltk
import json

# Function that fetches entity linking results from wikifier.com API
def wikifier(text, lang="en", threshold=0.8):
    uris_is=[]
    URI="http://group6.com/"

    # Prepare the URL for API call
    data = urllib.parse.urlencode([
        ("text", text), ("lang", lang),
        ("userKey", "tgbdmkpmkluegqfbawcwjywieevmza"),
        ("pageRankSqThreshold", "%g" %
         threshold), ("applyPageRankSqThreshold", "true"),
        ("nTopDfValuesToIgnore", "100"), ("nWordsToIgnoreFromList", "100"),
        ("wikiDataClasses", "true"), ("wikiDataClassIds", "false"),
        ("support", "true"), ("ranges", "false"), ("minLinkFrequency", "2"),
        ("includeCosines", "false"), ("maxMentionEntropy", "3")
    ])
    url = "http://www.wikifier.org/annotate-article"
    # Call the Wikifier and read the response
    req = urllib.request.Request(url, data=data.encode("utf8"), method="POST")
    with urllib.request.urlopen(req, timeout=60) as f:
        response = f.read()
        response = json.loads(response.decode("utf8"))


    results = list()
    subjects_found = set()

    # Output the results
    print(response["annotations"])

    # Scroll through the results building "isA" triples and associated URIs
    for annotation in response["annotations"]:
        if ('wikiDataClasses' in annotation):

          text_reference = ""
          text_reference_new = ""
          for el in annotation["support"]:

            # Getting text reference by looking at starting character and ending character of each reference found in the text
            # The longest reference (highest number of charachter) is taken as final reference
            ch_start = el['chFrom']
            ch_stop = el['chTo']
            if ch_start < len(coeref_text) and ch_stop < len(coeref_text):
              text_reference_new = coeref_text[ch_start:ch_stop+1]
              if text_reference_new[-2] == '\'':
                text_reference_new = text_reference_new[0:-2]

              if len(text_reference_new) > len(text_reference):
                text_reference = text_reference_new

          # If the text reference and the title of the annotation doesn't match, another "is a" triple
          # (and an associated URI) is added with the right annotation name
          if text_reference != annotation['title']:
            results.append(str(text_reference + " is a " + annotation['title']))
            uris_is.append(["<"+URI+"subject#"+text_reference.replace(" ","_")+">", "<"+URI+"pred#is", "<"+URI+"obj#"+annotation['title'].replace(" ","_")+">"])
          else:
            text_reference = annotation['title']

          # (ex. if we have "Professor Snape" in the text and the annotation is "Severus Snape", it's build the triple "Professor Snape is a Severus Snape")

          subjects_found.add(text_reference)

          # Saving "isA" triples and URIs
          for label in annotation['wikiDataClasses']:
            results.append(str(text_reference + " is a " + label['enLabel']))
            uris_is.append(["<"+URI+"subject#"+text_reference.replace(" ","_")+">", "<"+URI+"pred#is", "<"+URI+"obj#"+label['enLabel'].replace(" ","_")+">"])


    # Returning "is a" triples, URIs and subjects (text reference with matching annotations on Wikipedia)
    return results, uris_is, subjects_found
```

Figure 12: Named Linking Entity with Wikifier code

3.3 Sentences Tokenization

In this part of the pipeline, there is the splitting of the loaded text in sentences (each one of them recognized by the '.' at the end) using the *sent_tokenize* method of the *nltk* module, and the splitting of the sentences in tokens (words) using the *word_tokenize* method of the *nltk* module.

3.3.1 Merging tokens of subjects

To avoid separating subjects composed of two or more words, this processing step looks at the list of those, obtained previously, and brings the words together to form the correct subjects as in figure 14.
We consider 5 as the maximum number of tokens of which a single subject can be composed.

```
from nltk.tokenize import sent_tokenize, word_tokenize

# Tokenize sentences
print("\nSplitting in sentences\n")
tokenized_sentences = sent_tokenize(coeref_text)
print(tokenized_sentences)

# Tokenize words by sentences
print("\nSplitting in tokens\n")
i = 0
while i < len(tokenized_sentences):
  tokenized_sentences[i] = word_tokenize(tokenized_sentences[i])
  i += 1
print(tokenized_sentences)
```

Figure 13: Sentence Tokenization code

```
# Function to merge words belonging to the same subject composed of more words,
# that have been divided by tokenization
def merge_subjects(words):
  i = 0
  new_tokenized = words.copy()
  while(i < len(new_tokenized)):
    subject = ""
    j = 0

    while (subject not in subject_more_words) and (j < 5) and (i+j < len(new_tokenized)):
      if j == 0:
        subject += new_tokenized[i+j]
      else:
        subject += " " + new_tokenized[i+j]
      j += 1

    if j != 5 and i+j < len(new_tokenized):
      new_tokenized[i+j-1] = subject
      j -= 1
      while j > 0:
        del new_tokenized[i+j-1]
        j -= 1

    i += 1

  return new_tokenized

# Merging words for obtaining composed subjects for each sentence
i = 0
while i < len(tokenized_sentences):
  tokenized_sentences[i] = merge_subjects(tokenized_sentences[i])
  i += 1

print(tokenized_sentences)
```

Figure 14: Merging Tokens code

3.4 Stopwords removal

Processing step used to remove the so called Stopwords from the tokenized sentences obtained previously, using the *stopwords* module of *nltk.corpus* to get the English Stopwords (not including punctuation and pronouns). Other words were also added to the module as they were not present but considered necessary, while we removed words from it that we consider important and that if removed caused loss of information.
Additionally, we also removed the tokens representing punctuation, using *string.puntuaction*.

```
from nltk.corpus import stopwords
import string

# Function to remove stopwords from sentence
def remove_stopwords(sentences):
  stop_words = set(stopwords.words('english'))
  not_stop_words = ('is', 'having', 'be',  'do', 'own', 'am', 'are', 'were', 'had', 'been', 'have', 'does', 'did', 'has', 'being', 'doing', 'was')
  pronouns = ['i', 'we', 'it', 'you', 'she', 'he', 'they', 'me', 'us', 'her', 'him', 'them']

  for word in not_stop_words:
    stop_words.remove(word)

  for word in pronouns:
      if word in stop_words:
          stop_words.remove(word)
  words_without_sw = [word for word in sentences if not word.lower() in stop_words and word.lower() not in string.punctuation and word != "-"]
  return words_without_sw

# Getting sentences without stopwords
sentences_without_sw = list()
i = 0
while i < len(tokenized_sentences):
  sentences_without_sw.append(remove_stopwords(tokenized_sentences[i]))
  i += 1

print(sentences_without_sw)
```

Figure 15: Stopwords removal code

3.5 POS Tagging

This step is performed using the *pos_tag* method of the *nltk* module and consists of tagging each word of the processed text with a POS tag, then removing all of those that are not tagged as nouns or verbs. In this step we noticed that some tags were labeled incorrectly, so to remedy this, we manually forced the label of the tokens representing the subjects (obtained through Named Entity Linking) to NNS.
Sometimes, however, it may happen that a verb can also be used as a noun or vice versa. Unfortunately, this leads to problems when creating triples. Therefore, this and other problems of this kind were unsolvable, since in order to solve them we need to enter in the specificity of the domain, losing the automation of the whole system. We accepted this trade-off as the triples obtained with these errors are not many and, often, they can be filtered in the triple extraction phase, as they led to very long triples that are discarded.

```python
from nltk import pos_tag

# Function to do POS tagging on sentence, removing words that are not
# verbs or nouns
def tagging(words):
  pos_words = pos_tag(words)
  nouns_dict = ['NN', 'NNS', 'NNP', 'NNPS']
  verbs_dict = ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']
  sent_clean = [(x,y) for (x,y) in pos_words if (y in nouns_dict or y in verbs_dict)]

  return sent_clean

# Getting POS tagged sentences
tagged_sentences = list()
i = 0
while i < len(sentences_without_sw):
  tagged_sentences.append(tagging(sentences_without_sw[i]))
  i += 1

print(tagged_sentences)
```

Figure 16: POS Tagging code

3.6 Lemmatization

This step is performed using the *WordNetLemmatizer* class of the *nltk.stem* module and consists of modifying the form of the verbs encountered in the processed sentences, bringing them to the infinite form using the *lemmatize* method. During this step there is also the tagging of common nouns as 'NN' and of subject nouns as 'NNS'.

```python
from nltk.stem import WordNetLemmatizer

# Function to lemmatize verbs, converting them in base form
def lemmatization(tagged_sentence):
  nouns_dict = ['NN', 'NNS', 'NNP', 'NNPS']
  verbs_dict = ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']
  lemmatizer = WordNetLemmatizer()
  lemmatized_words = []
  for (x,y) in tagged_sentence:
    if y in verbs_dict:
      y = 'V'
      x = lemmatizer.lemmatize(x, 'v')
    if x in subjects:
      y = 'NNS'
    lemmatized_words.append((x,y))

  return lemmatized_words

  # Getting lemmatized sentences
lemmatized_sentences = list()
i = 0
while i < len(tagged_sentences):
  lemmatized_sentences.append(lemmatization(tagged_sentences[i]))
  i += 1

print(lemmatized_sentences)
```

Figure 17: Lemmatization code

3.7 Taking synset from WordNet

In our project we also made use of a well-known large lexical database of English, *WordNet*. In WordNet the nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (*synsets*). Synsets are interlinked by means of conceptual-semantic and lexical relations.

In our project we thought of using synsets to be able to extrapolate as many triples as possible. To do it, we use the *WordNet* module of the *nltk.corpus* library, obtaining synonyms using *WordNet's synsets* method.

Specifically, we decided to create new triples starting from the original triples and replacing them with the verb with synonyms. In order to do this, the verb was extrapolated from the triple, the synset associated with it was searched and, subsequently, filtered to obtain only the synonyms that are verbs, i.e. the chosen synonyms are only the ones that are considered verbs (looking at what's the POS tag on *WordNet*) and they are stored in a dictionary of dictionaries. Having obtained the list of synonyms, let's create a triple for each of them. It is important to note that we only take distinct synonyms, so we avoid repetition.

```
from nltk.corpus import wordnet as wn

# Getting synonyms of each verb in sentences by WordNet (the synonyms are only verbs too)
verbs_synonyms = {}
for sentence in lemmatized_sentences:
  for (x,y) in sentence:
    if y == 'V':
      verbs_synonyms[x] =  {str(lemma.name()).split('.')[0] for lemma in wn.synsets(x, pos=wn.VERB) if str(lemma.name()).split('.')[0] != x }
print(verbs_synonyms)
```

Figure 18: Taking synset from WordNet code

3.8 Finding Triples

This step is performed by looking at each word of each sentence to create 'Subject-Predicate-Object' triples. The subjects are one or more nouns (present in *nouns_dict*) that follow each other, the subjects are one or more verbs (lemmatized as '*V*') that follow each other, the objects are single nouns that follow verbs. The triples and URIs are then formed and stored in lists.
By manually extrapolating triples after the various steps of the NLP, we noticed that in some cases there were errors of POS Tagging and Lemmatization, in which some verbs were labeled as nouns and vice versa. In such cases very long (and often wrong) triples were obtained, since an entire triple (the one containing the verb labeled as a noun) was labeled as the subject.
In order to overcome this, an analysis step has been introduced which involves filtering the triples, keeping only those that satisfy certain parameters, such as a maximum number of tokens in the subject and a maximum number of tokens in the predicate.

The pipeline developed in this way allow to handle also difficult statements in which the object of a triple is another triple itself and so on; in the final result the statement is decomposed in more simple triples and the entire statement sense can be retrieved by seeing these consecutive triples.

```python
# Function to extrapolate "Subject-Predicate-Object" triples and associated URIs from a sentence
def find_triples(lemmatized_sentence, verbs_synonyms, URI="http://group6.com/"):
  triples = []
  uris = []
  nouns_dict = ['NN', 'NNS', 'NNP', 'NNPS']
  verbs_dict = ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']
  i=0
  len_words = len(lemmatized_sentence)
  last_previous_token = ""

  while(i < len_words):
    triple = ""
    subject_flag = 0
    obj_flag = 0
    pred_flag = 0
    subject_treshold = 2
    pred_treshold = 1
    subject = ""
    pred = ""
    obj = ""

    # Building subject part
    while(i < len_words and lemmatized_sentence[i][1] in nouns_dict):
      subject += lemmatized_sentence[i][0] + " "
      i += 1
      subject_flag += 1
    # Removing possible duplicates
    subject = ' '.join([x for i,x in enumerate(subject.split()) if i==0 or x!=subject.split()[i-1]])

    # Adding subject to the triple
    triple += subject + " "

    # Building predicate part
    while(i < len_words and lemmatized_sentence[i][1] == 'V'):
      pred += lemmatized_sentence[i][0] + " "
      i += 1
      pred_flag += 1

    # Removing possible duplicates
    pred = ' '.join([x for i,x in enumerate(pred.split()) if i==0 or x!=pred.split()[i-1]])

    # Adding subject to the triple
    triple += pred + " "

    # Getting object
    if(i < len_words and lemmatized_sentence[i][1] in nouns_dict):
        obj = lemmatized_sentence[i][0]
        obj_flag += 1

    # Adding object to triple
    triple += obj

    # Checking if every part of the triple have been taken and checking maximum number of words in subject and predicate part
    # If there are more words then the allowed ones the triple is not valid
    if (subject_flag > 0 and subject_flag <= subject_treshold) and (pred_flag > 0 and pred_flag <= pred_treshold) and obj_flag >

        # If the check is passed the triple is saved
        triples.append(triple)

        # Building and saving URI
        uris.append(["<"+URI+"subject#"+subject.replace(" ","_")+">", "<"+URI+"pred#"+pred.replace(" ","_")+">", "<"+URI+"obj#"+obj.replace(" ","_")+">"])

        # Building new triples by using verb's synonyms obtained by WordNet
        for verb in pred.split(" "):
          if verb in verbs_synonyms:
            for synonym in verbs_synonyms[verb]:

              # Changing predicate part
              syn_pred = pred.replace(verb, synonym)

              # Building new triple
              new_triple = subject + " " + syn_pred + " " + obj

              # Saving new triple
              triples.append(new_triple)

              # Building and saving new URI
              uris.append(["<"+URI+"subject#"+subject.replace(" ","_")+">", "<"+URI+"pred#"+syn_pred.replace(" ","_")+">", "<"+URI+"obj#"+obj.replace(" ","_")+">"])

    # Returning sentence lists of "Subject-Predicate-Object" triples and associated URIs
    return triples, uris


triples_total = list()
uris_total = list()
i = 0

# Getting "Subject-Predicate-Object" triples and associated URIs by text sentences
while i < len(lemmatized_sentences):
  triple, uri = find_triples(lemmatized_sentences[i], verbs_synonyms)
  triples_total.append(triple)
  uris_total.append(uri)
  i += 1
triples = [ triple for listOfTriple in triples_total for triple in listOfTriple ]
uris = [ uri for listOfUri in uris_total for uri in listOfUri ]
```

Figure 19: Finding Triples code

4 Validation/testing

The pipeline previously detailed has been tested on some different texts. It's here reported an example of the pipeline in action on a small piece of one of this texts:

```
The central character in the series is Harry Potter, a boy who lives
in the fictional town of Little Whinging, Surrey with his aunt, uncle,
and cousin the Dursleys and discovers at the age of eleven that he
is a wizard, though he lives in the ordinary world of non-magical people
known as Muggles. The wizarding world exists parallel to the Muggle world,
albeit hidden and in secrecy. His magical ability is inborn, and children
with such abilities are invited to attend exclusive magic schools that teach
the necessary skills to succeed in the wizarding world.
```

4.1 Coeference Resolution

```
The central character in the series is Harry Potter, a boy who lives in the
fictional town of Little Whinging, Surrey with The central character in the
series aunt, uncle, and cousin the Dursleys and discovers at the age of
eleven that The central character in the series is a wizard, though The central
character in the series lives in the ordinary world of non-magical people known
as Muggles. The wizarding world exists parallel to the Muggle world, albeit
hidden and in secrecy. The central character in the series magical ability is
inborn, and children with such abilities are invited to attend exclusive magic
schools that teach the necessary skills to succeed in the wizarding world.
```

4.2 Named Entity Linking with Wikifier

It's shown only a part of the results for length reasons.


```
'title': 'Harry Potter', 'url': 'http://en.wikipedia.org/wiki/Harry_Potter',
'lang': 'en', 'pageRank': 0.02593534651626024, 'secLang': 'en',
'secTitle': 'Harry Potter', \\'secUrl': 'http://en.wikipedia.org/wiki/Harry_Potter',
'wikiDataItemId': 'Q8337', 'wikiDataClasses':
[{'itemId': 'Q1667921', 'enLabel': 'novel series'},
{'itemId': 'Q614101', 'enLabel': 'heptalogy'},
{'itemId': 'Q277759', 'enLabel': 'book series'},
{'itemId': 'Q15980953', 'enLabel': 'fiction series'},
{'itemId': 'Q20936777', 'enLabel': 'series of specified number of works'},
{'itemId': 'Q29441572', 'enLabel': 'heptad'},
{'itemId': 'Q7725310', 'enLabel': 'series of creative works'},
{'itemId': 'Q17538690', 'enLabel': 'group of manifestations'},
{'itemId': 'Q47461344', 'enLabel': 'written work'}
```

4.3 Extracting first type of Triples (and associated URIs)

Here we show some of the 'is a' triples (and associated URIs) extracted in this first phase (not all of them are reported for length reasons):

Harry Potter is a novel series
'http://group6.com/subject#Harry_Potter', 'http://group6.com/pred#is',
'http://group6.com/obj#novel_series'

Harry Potter is a heptalogy
'http://group6.com/subject#Harry_Potter', 'http://group6.com/pred#is',
'http://group6.com/obj#heptalogy'

Harry Potter is a book series
'http://group6.com/subject#Harry_Potter', 'http://group6.com/pred#is',
'http://group6.com/obj#book_series'

Harry Potter is a fiction series
'http://group6.com/subject#Harry_Potter', 'http://group6.com/pred#is',
'http://group6.com/obj#fiction_series'

Harry Potter is a series of specified number of works
'http://group6.com/subject#Harry_Potter', 'http://group6.com/pred#is',
'http://group6.com/obj#series_of_specified_number_of_works'

. . .

wizarding world is a media franchise
'http://group6.com/subject#wizarding_world', 'http://group6.com/pred#is',
'http://group6.com/obj#media_franchise'

wizarding world is a fictional universe
'http://group6.com/subject#wizarding_world', 'http://group6.com/pred#is',
'http://group6.com/obj#fictional_universe'

wizarding world is a intellectual work
'http://group6.com/subject#wizarding_world', 'http://group6.com/pred#is',
'http://group6.com/obj#intellectual_work'

wizarding world is a series of creative works
'http://group6.com/subject#wizarding_world', 'http://group6.com/pred#is',
'http://group6.com/obj#series_of_creative_works'

wizarding world is a fictional entity
'http://group6.com/subject#wizarding_world', 'http://group6.com/pred#is',
'http://group6.com/obj#fictional_entity'

wizarding world is a setting
'http://group6.com/subject#wizarding_world', 'http://group6.com/pred#is',
'http://group6.com/obj#setting'

. . .

Little Whinging is a Places in Harry Potter
'http://group6.com/subjectLittle_Whinging', 'http://group6.com/predis',
'http://group6.com/objPlaces_in_Harry_Potter'

Little Whinging is a class of fictional entities
'http://group6.com/subjectLittle_Whinging', 'http://group6.com/predis',
'http://group6.com/objclass_of_fictional_entities'

Little Whinging is a fictional entity
'http://group6.com/subjectLittle_Whinging', 'http://group6.com/predis',
'http://group6.com/objfictional_entity'

. . .

the Dursleys is a List of supporting Harry Potter characters
'http://group6.com/subjectthe_Dursleys', 'http://group6.com/predis',
'http://group6.com/objList_of_supporting_Harry_Potter_characters'

the Dursleys is a Wikimedia list of fictional characters
'http://group6.com/subjectthe_Dursleys', 'http://group6.com/predis',
'http://group6.com/objWikimedia_list_of_fictional_characters'

the Dursleys is a Wikimedia list article
'http://group6.com/subjectthe_Dursleys', 'http://group6.com/predis',
'http://group6.com/objWikimedia_list_article'

. . .

4.4 Wikifier subjects retrieval

Subjects:

```
wizarding world
the Dursleys
Harry Potter
Surrey
Muggle
Little Whinging
Muggles
List of supporting Harry Potter characters
Wizarding World
Places in Harry Potter
```

More Words Subjects:

```
wizarding world
the Dursleys
Harry Potter
Little Whinging
List of supporting Harry Potter characters
Wizarding World
Places in Harry Potter
```

4.5 Tokenization

```
'The', 'central', 'character', 'in', 'the', 'series', 'is', 'Harry', 'Potter',
',', 'a', 'boy', 'who', 'lives', 'in', 'the', 'fictional', 'town', 'of',
'Little', 'Whinging', ',', 'Surrey', 'with', 'The', 'central', 'character',
'in', 'the', 'series', 'aunt', ',', 'uncle', ',', 'and', 'cousin', '-', 'the',
'Dursleys','-', 'and', 'discovers', 'at', 'the', 'age', 'of', 'eleven', 'that',
'The', 'central', 'character', 'in', 'the', 'series', 'is', 'a', 'wizard', ',',
'though', 'The', 'central', 'character', 'in', 'the', 'series', 'lives', 'in',
'the', 'ordinary', 'world', 'of', 'non-magical', 'people', 'known', 'as',
'Muggles', '.','The', 'wizarding', 'world', 'exists', 'parallel', 'to', 'the',
'Muggle', 'world', ',', 'albeit', 'hidden', 'and', 'in', 'secrecy', '.',
'The', 'central', 'character', 'in', 'the', 'series', 'magical', 'ability', 'is',
'inborn', ',', 'and', 'children', 'with', 'such', 'abilities', 'are',
'invited', 'to', 'attend', 'exclusive', 'magic', 'schools', 'that', 'teach','the',
'necessary', 'skills', 'to', 'succeed', 'in', 'the', 'wizarding', 'world', '.'
```

4.6 Merging composed subjects

```
'The', 'central', 'character', 'in', 'the', 'series', 'is', 'Harry Potter',
',', 'a', 'boy', 'who', 'lives', 'in', 'the', 'fictional', 'town', 'of',
'Little Whinging', ',', 'Surrey', 'with', 'The', 'central', 'character', 'in',
'the', 'series', 'aunt', ',', 'uncle', ',', 'and', 'cousin', '  ', 'the Dursleys',
'  ', 'and', 'discovers', 'at', 'the', 'age', 'of', 'eleven', 'that', 'The',
'central', 'character', 'in', 'the', 'series', 'is', 'a', 'wizard', ',',
'though', 'The', 'central', 'character', 'in', 'the', 'series', 'lives', 'in',
'the', 'ordinary', 'world', 'of', 'non-magical', 'people', 'known', 'as',
'Muggles', '.','The', 'wizarding world', 'exists', 'parallel', 'to', 'the',
'Muggle', 'world', ',', 'albeit', 'hidden', 'and', 'in', 'secrecy', '.', 'The',
'central', 'character', 'in', 'the', 'series', 'magical', 'ability', 'is',
'inborn', ',', 'and', 'children', 'with', 'such', 'abilities', 'are', 'invited',
'to', 'attend', 'exclusive', 'magic', 'schools', 'that', 'teach', 'the',
'necessary', 'skills', 'to', 'succeed', 'in', 'the', 'wizarding world', '.'
```

4.7 Stopwords removal

```
'central', 'character', 'series', 'is', 'Harry Potter', 'boy', 'lives',
'fictional', 'town', 'Little Whinging', 'Surrey', 'central', 'character',
'series', 'aunt', 'uncle', 'cousin', 'the Dursleys', 'discovers', 'age',
'eleven', 'central', 'character', 'series', 'is', 'wizard', 'though', 'central',
'character', 'series', 'lives', 'ordinary', 'world', 'non-magical', 'people',
'known', 'Muggles', 'wizarding world', 'exists', 'parallel', 'Muggle',
'world', 'albeit', 'hidden', 'secrecy', 'central', 'character', 'series',
'magical', 'ability', 'is', 'inborn', 'children', 'abilities', 'invited',
'attend', 'exclusive', 'magic', 'schools', 'teach', 'necessary', 'skills',
'succeed', 'wizarding world'
```

4.8 POS Tagging

```
('character', 'NN'), ('series', 'NN'), ('is', 'VBZ'), ('Harry Potter', 'NNP'),
('boy', 'NN'), ('lives', 'VBZ'), ('town', 'NN'), ('Little Whinging', 'NNP'),
('Surrey', 'NNP'), ('character', 'NN'), ('series', 'NN'), ('aunt', 'NN'),
('cousin', 'NN'), ('the Dursleys', 'NNS'), ('discovers', 'NNS'), ('age', 'NN'),
('character', 'NN'), ('series', 'NN'), ('is', 'VBZ'), ('character', 'NN'),
('series', 'NN'), ('lives', 'VBZ'), ('world', 'NN'), ('people', 'NNS'),
('known', 'VBN'), ('Muggles', 'NNS'), ('wizarding world', 'NN'),
('exists', 'VBZ'), ('Muggle', 'NNP'), ('world', 'NN'), ('albeit', 'NN'),
('hidden', 'NN'), ('secrecy', 'NN'), ('character', 'NN'), ('series', 'NN'),
('ability', 'NN'), ('is', 'VBZ'), ('children', 'NNS'), ('abilities', 'NNS'),
('invited', 'VBD'), ('magic', 'NN'), ('schools', 'NNS'), ('teach', 'VBP'),
('skills', 'NNS'), ('succeed', 'VB'), ('wizarding world', 'NN')
```

4.9 Lemmatization

```
('character', 'NN'), ('series', 'NN'), ('be', 'V'), ('Harry Potter', 'NNS'),
('boy', 'NN'), ('live', 'V'), ('town', 'NN'), ('Little Whinging', 'NNS'),
('Surrey', 'NNS'), ('character', 'NN'), ('series', 'NN'), ('aunt', 'NN'),
('cousin', 'NN'), ('the Dursleys', 'NNS'), ('discovers', 'NNS'), ('age', 'NN'),
('character', 'NN'), ('series', 'NN'), ('be', 'V'), ('character', 'NN'),
('series', 'NN'), ('live', 'V'), ('world', 'NN'), ('people', 'NNS'),
('know', 'V'), ('Muggles', 'NNS'), ('wizarding world', 'NNS'),
('exist', 'V'), ('Muggle', 'NNS'), ('world', 'NN'), ('albeit', 'NN'),
('hidden', 'NN'), ('secrecy', 'NN'), ('character', 'NN'), ('series', 'NN'),
('ability', 'NN'), ('be', 'V'), ('children', 'NNS'), ('abilities', 'NNS'),
('invite', 'V'), ('magic', 'NN'), ('schools', 'NNS'), ('teach', 'V'),
('skills', 'NNS'), ('succeed', 'V'), ('wizarding world', 'NNS')
```

4.10 Taking verbs synonyms from WordNet

```
{'be': {'cost', 'embody', 'constitute', 'equal', 'exist'},
'live': {'exist', 'populate', 'survive', 'know', 'be'},
'know': {'sleep_together', 'acknowledge'},
'exist': set(),
'invite': {'tempt', 'receive'},
'teach': set(),
'succeed': set()}
```

4.11 Extracting second type of Triples (and associated URIs)

Here we show some of the Subject-Predicate-Object triples (and associated URIs) extracted from the text:

character series be Harry Potter
'http://group6.com/subject#character_series', 'http://group6.com/pred#be',
'http://group6.com/obj#Harry_Potter'

character series cost Harry Potter
'http://group6.com/subject#character_series', 'http://group6.com/pred#cost',
'http://group6.com/obj#Harry_Potter'

character series exist Harry Potter
'http://group6.com/subject#character_series', 'http://group6.com/pred#exist',
'http://group6.com/obj#Harry_Potter'

character series constitute Harry Potter
'http://group6.com/subject#character_series',
'http://group6.com/pred#constitute', 'http://group6.com/obj#Harry_Potter'

character series equal Harry Potter
'http://group6.com/subject#character_series', 'http://group6.com/pred#equal',
'http://group6.com/obj#Harry_Potter'

character series embody Harry Potter
'http://group6.com/subject#character_series', 'http://group6.com/pred#embody',
'http://group6.com/obj#Harry_Potter'

Harry Potter boy live town
'http://group6.com/subject#Harry_Potter_boy', 'http://group6.com/pred#live',
'http://group6.com/obj#town'

Harry Potter boy be town
'http://group6.com/subject#Harry_Potter_boy', 'http://group6.com/pred#be',
'http://group6.com/obj#town'

Harry Potter boy survive town
'http://group6.com/subject#Harry_Potter_boy',
'http://group6.com/pred#survive', 'http://group6.com/obj#town'

Harry Potter boy exist town
'http://group6.com/subject#Harry_Potter_boy', 'http://group6.com/pred#exist',
'http://group6.com/obj#town'

Harry Potter boy know town
'http://group6.com/subject#Harry_Potter_boy', 'http://group6.com/pred#know',
'http://group6.com/obj#town'

Harry Potter boy populate town
'http://group6.com/subject#Harry_Potter_boy',
'http://group6.com/pred#populate', 'http://group6.com/obj#town'

character series live world
'http://group6.com/subject#character_series', 'http://group6.com/pred#live',
'http://group6.com/obj#world'

character series be world
'http://group6.com/subject#character_series', 'http://group6.com/pred#be',
'http://group6.com/obj#world'

character series survive world
'http://group6.com/subject#character_series', 'http://group6.com/pred#survive',
'http://group6.com/obj#world'

character series exist world
'http://group6.com/subject#character_series', 'http://group6.com/pred#exist',
'http://group6.com/obj#world'

character series know world
'http://group6.com/subject#character_series', 'http://group6.com/pred#know',
'http://group6.com/obj#world'

character series populate world
'http://group6.com/subject#character_series',
'http://group6.com/pred#populate', 'http://group6.com/obj#world'

world people know Muggles
'http://group6.com/subject#world_people', 'http://group6.com/pred#know',
'http://group6.com/obj#Muggles'

world people sleep_together Muggles
'http://group6.com/subject#world_people',
'http://group6.com/pred#sleep_together', 'http://group6.com/obj#Muggles'

world people acknowledge Muggles
'http://group6.com/subject#world_people', 'http://group6.com/pred#acknowledge',
'http://group6.com/obj#Muggles'

wizarding world exist Muggle
'http://group6.com/subject#wizarding_world', 'http://group6.com/pred#exist',
'http://group6.com/obj#Muggle'

magic schools teach skills
'http://group6.com/subject#magic_schools', 'http://group6.com/pred#teach',
'http://group6.com/obj#skills'

skills succeed wizarding world
'http://group6.com/subject#skills', 'http://group6.com/pred#succeed',
'http://group6.com/obj#wizarding_world'

As it can be saw from this example, the final results is quite good. It's possible to extract a lot of information, even from a short text, expecially through the 'is a' triples; however we can also see that, some of these are a little bit forced, because of the distancing by the real class, but (as already reported in the Implementation paragraph) it has been decided to accept this. Also the triples extracted from the text are quite good and possible errors during the NLP phases (like a verb tagged as a noun in the POS tagging phase) are not related to our pipeline but on the library functions, so we don't have control of them.