CentraleSupélec

Projet Logiciel Report

Discovering inter-language links in Wikipedia

ARARIPE FURTADO CUNHA Rebeca

SHARMA Naman

N°de binome: 142

Promo 2019

June 5, 2018

Contents

1 Introduction

Wikipedia is a massive online database that serves as an digital encyclopedia that can be edited and read by anyone. The web-based encyclopedia was initially available in only English but currently Wikipedia has editions in 301 different languages. English remains the largest of these with 5,649,063 articles as of 14th May 2018. Each Wikipedia language edition exists in the form of a tree, branching our based on the categories of the articles. The articles themselves are similar to the nodes of the tree, and are connected to each other by links (or "internal links"). Wikipedia also allows users to switch from one language to another by providing inter-language links (called "cross-links") for every article. Hence a cross-link connects the same article between two different editions of Wikipedia in two distinct languages.

It is important to note that since Wikipedia is edited openly by the public, these cross-links are also generated by the public. As a result, we often have cross-links that are missing between languages or sometimes articles in two different languages that have been erroneously cross-linked.

In their paper, [Penta et al., 2012] propose the WikiCL algorithm that can be used to discover cross-links from an article in the source language Wikipedia edition to an article in the target language Wikipedia edition. The WikiCL algorithm takes as inputs: i) the title of the source article, ii) the language of the source article and iii) the language of the target article. It returns back as output the title of the article in the target language. What is unique about the WikiCL algorithm is that it is unsupervised and language independent, leading to robustness. Being unsupervised means that this algorithm does not need to be trained over a subset of the Wikipedia graph and does not involve a complex feature extraction process. Being language independent means that this algorithm does not depend upon any text-based features, and hence does not use any translating dictionaries between languages to find the target node.

In this programming project, we implement the WikiCL algorithm by using Neo4j as the database management system to store the information of the articles in Wikipedia. Neo4j is the rational choice for this application as it is a graph based database. The Wikipedia articles are considered to be the nodes of the graph, which are connected to each other by links or cross-links. To implement the WikiCL algorithm, we use Python along with the available Neo4j Bolt Driver. We choose Python because of the presence of extensive libraries for data manipulation and its concise expressions.

2 Notation

In this work, we follow the notation established by [Penta et al., 2012]. A directed Wikipedia graph in a language l is represented by W_l. Articles are labeled as lowercase Greek letters, while the node corresponding to a given article α is written in lowercase Latin letters as in v_α. $\mathrm{CL}_t(v_\alpha)$ stands for the node w_β in the target language t such that α and β are cross-linked. $N(v)$ is the set of neighbors of v in its own language, regardless of the direction of the link(incoming, outgoing or bidirectional) between them. $N_{in}(v)$ refers then to the nodes of v which links towards v, whereas $N_{out}(v)$ refers to the nodes to which v links to. $Label(v_{beta})$ stands for the title of the article $beta$.

3 Algorithm Description

In this section we present a brief explanation of the WikiCL algorithm to find language cross-links in Wikipedia as described by [Penta et al., 2012]. A version of Wikipedia in language s is modeled by a graph W_s in which every article corresponds to a node v connected by internal links to other nodes of W_l and by cross-links to nodes w belonging to a different language Wikipedia edition W_t.Here, we use a subset of the Wikipedia editions in English and French, with 126805 nodes and 203283 nodes respectively.

Each node in the graph has the following properties:

- *title*: The title corresponding to the node
- *lang*: The language of the article, written using the ISO 639-1 code of the language
- *latitude* (optional): The latitude value of the entity described in the article
- *longitude*: (optional): The longitudinal value of the entity described in the article

The WikiCL algorithm is structured in 2 majors stages: the generation of the candidates set amongst the target language nodes; and the selection of the best suitable node with respect to some criteria further described.

3.1 Candidates Generation

The goal at this stage is to choose a set of nodes C that are likely to contain the translation $v_\beta \in W_t$ of the node $v_\alpha \in W_s$. In order to maximize the chances of success,the search space is reduced by dividing the nodes into 3 categories: non-geographic named entity, geographic named entity and non-named entity. The first challenge is to correctly place them.
Nodes presenting latitude and longitude are classified as geographic named entities. In the absence of coordinates, v is non-geographic named if:

- The title of v is composed of a single word presenting more than one capital letter(case of an acronym).
- The title of v has multiple words, each one beginning by a capital letter with the exception of prepositions, determiners, conjunctions, relative pronouns and negations.

A good evidence of a named entity is at least 75% occurrence of the title in the text. Since here we don't have access to the text article itself, we will content ourselves with the indications cited above.
Accordingly to the category of the node v_α, the candidates for its translation in the target language will be chosen following 3 different approaches.

Case 1: v_α is non-geographic named entity

Since this category accounts for proper nouns(names of people, companies, books, events, etc) and thus usually remains the same regardless of the language, we first check if there exists a node $v_\gamma \in W_t$ where $Label(v_\gamma) = Label(v_\alpha)$. If we don't find such a node, then the algorithm skips to case 3. In the same way, if v_γ exists but it has geographic coordinates, the candidate is dropped and case 3 is also invoked. This prevents mistakes when a proper name can also designate a

name of a place. Finally, if v_γ exists and is named non-geographic, then it is automatically elected as the sole candidate and thus the target article. In the other hand, common nouns are almost always different in distinct languages and v_γ is likely to be not found. v_α is processed in case 3.

Case 2: v_α is a geographic named entity

Here, the first step is to check if there exists a node v_γ in the target language with the same title as α and presenting geographic coordinates. When v_γ cannot be found, we take as candidates the nodes with geographic locations residing in a predetermined radius R from the geographic location of v_α. For every node w in the current set of candidates C, we add also to the selection $N(w)$.
By adding to candidates the nodes that are geographically nearby or graph neighbors, this method allows us to account for errors or imprecisions due to the manual entry of geographic coordinates.

Case 3: v_α is not a named entity

In this case, we fairly assume that v_β and v_α are connected by a great amount of chain-links. In other words, a lot of neighbors of v_β are very likely present as cross-links to the neighbors of v_β. We add to candidate set C with the highest number of chain-links to v_α. In order to elect to candidates the nodes most related to v_α, we start by selecting from the group of mutual neighbors of v_α. If the final number of candidates is not enough according to a predefined limit, we can add to C the best choices amongst the $N_out(v_\alpha)$.

3.2 Target Selection

The next step after generating the candidates is to select an article from the candidates C as the translation of v_α. The WikiCL algorithm proposes to do this selection by the use of a relatedness score $S(v_\gamma)$. This relatedness score is calculated for every node $v_\gamma \in C$ and the node with the highest score is selected as the cross-link for v_α, as long as the score is above a certain threshold, experimentally set to 0.15 by the authors. Since the target node and the source node are in two different languages, the calculation of their similarity becomes difficult. WikiCL uses the structure of the graphs to calculate this similarity vector and to translate the similarity vectors between languages.

Wikipedia Link-based Measure (WLM)

The WikiCL algorithm proposes calculating a similarity vector $Sim^l(V)$ for every node $v \in W_l$. For each node $i \in N_{out}(v)$, the value of $Sim^l(v)[i]$ is a measure of semantic relatedness of i to v. To calculate this similarity vector, the Wikipedia Link-based Measure(WLM) is used. The WLM was presented by [Milne and Witten, 2007] and is based on the Normalized Google

Distance. It can be defined as:

$$WLM(v,i) = \frac{max\{\log f(v), \log f(i)\} - \log f(v,i)}{\log|W| - min\{\log f(v), \log f(i)\}} \tag{1}$$

where,
v, i: Wikipedia articles
$f(v)(f(i))$: number of articles that link to v(respectively, i)
$f(v,i)$: number of articles that link to both $v$and i
$|W|$: total number of articles in Wikipedia

According to [Penta et al., 2012] the calculation of this similarity vector is what dominates the running time of the algorithm.

For the algorithm, the similarity vectors $Sim^s(v_\alpha)$ (for the source node v_α) and $Sim^t(v_\gamma)$ (for all nodes $v_\gamma \in C$) are calculated. However, we need to keep in mind that $Sim^t(v_\gamma)$ corresponds to nodes in the target Wikipedia, while $Sim^s(v_\alpha)$ is calculated for the source Wikipedia. Hence a mapping between the two languages is needed. This can be achieved in the following manner: For each node $i \in N(v_\gamma)$ we get $CL_s(i)$ in W_s. If $CL_s(i)$ exists, then $Sim^s(v_\gamma)[CL_s(i)] = Sim^t(v_\gamma)[i]$, else $Sim^s(v_\gamma)[CL_s(i)] = 0$ The final similarity score is taken to be the cosine similarity between $Sim^s(v_\alpha)$ and $Sim^s(v_\gamma)$.

Jaccard Index

As mentioned by [Penta et al., 2012] in their paper, and also seen by us during the tests, the running time of the algorithm is dominated by the calculation of the WLM metric for a node. In an attempt to reduce this time complexity, we propose another metric which is easier to calculate and requires less computational power. Once implemented, we can test its efficiency and discuss its relative strengths and weaknesses over the WLM metric. The metric that we propose is the Jaccard index.
The Jaccard Index between two nodes v_α and i can be defined as:

$$J(v_\alpha, i) = \frac{N_{out}(v_\alpha) \cap N_{out}(i)}{N_{out}(v_\alpha) \cup N_{out}(i)} \tag{2}$$

Similar to the case of the WLM metric, we calculate the vector $Sim^l(v_\alpha)$ by calculating the Jaccard index between v_α and $i \forall i \in N_{out}(v_\alpha)$. We perform a similar mapping between $Sim^t(v_\gamma)$ and $Sim^s(v_\gamma)$.

4 Implementation of Algorithm

4.1 Candidate Generator

The Candidate Generator class is responsible for building and returning the candidate set C, which is here a 1D array of nodes. It takes as input the Neo4j session managed by the Bolt driver, a string corresponding to the title of the article in the source Wikipedia, a string for the source language and a string to the target language("en" or "fr").

Requests to the database returns a object of the type statementresult and can be turned into an iterable of record objects by applying its method records(). In the Candidate Generator

initialization function we categorize the source article by assigning boolean values to the variables **geographic** and **named** according the criteria explained in section 3.1 and commented in the following paragraphs.

Categorization of source node

The first step towards selecting candidates is to figure out whether the source node is non named, named geographic or named non-geographic. This is done using the code below: The **checkGeographic** function simply returns *True* if it finds a latitude attribute associated with the node. When trying to decide whether a node with more than one word in the title is named, we decided to categorize it as named if at least any two words started with a capital letter. This is done to ensure that we do not mislabel a node as non-named because the title contains pronouns, prepositions etc., which always start with a lowercase letter in Wikipedia In the method **generateCandidates()** we call the functions **case1**, **case2** or **case3** depending on the value of **geographic** and **named**.

Case 1: Named non geographic entity

The implementation of this case is fairly simple and exactly follows the steps explained previously in 3.1. The Cypher request for finding articles with the same title is also straightforward.

Case 2: Named non geographic entity

Here, we deploy exactly the same procedure described in 3.1. The query used is depicted below.

```
MATCH (a:Article)-[:link]-(b:Article)
WHERE a.lang = $t_lang AND
2 * $earthR * asin(sqrt(haversin(radians($lat - a.latitude))
+ cos(radians($lat))*cos(radians(a.latitude))*
haversin(radians($long - a.longitude)))) <= $limit
UNWIND [a,b] AS x
WITH x , COUNT(x) as count RETURN  x ORDER BY count
DESCENDING LIMIT $maxNum
```

The dollar sign $ allows us to access the values of variables defined in the code, which are passed as parameters of the run function of the cypher session. The use of this feature suggests better performance than simply converting each variable into strings and then concatenating the entire request into a single string, which is passed as argument to the run function. This is becuase the Bolt driver sees these queries as repeated, and in turn does various optimisations in the backend. However, if we were to simply concatenate the values into the string, this would be seen by the driver as performing different queries each time.

In this query, $t_lang refers to the language of the target Wikipedia, $earthR to radius of the Earth, $lat and $long to the geographic coordinates of the source article. We compute the great-circle distance between two locations by applying the Haversin function. The $limit distance chosen for the articles to be considered candidates was of 3 miles.

Since this request can return an enormous number of candidates (for example, in the case of Paris it returns 127876 candidates out of the possible 203283 articles in the French Wikipedia) we limit the size of resulting records to $maxNum = 1000. Before selecting the first 1000 records we make sure to sort list in descending order of number of times it appears as a neighbor of the articles inside the liming geodistance radius.

Case 3: Non-named entity

In the case of non-named entity, the following query is used to get the $N(CL_t(w_\gamma))$ where $w_\gamma \in M(v_\alpha)$:

```
MATCH (n:Article)-[:link]-(c:Article{lang:$t_lang})
    -[:crosslink]-(b:Article)-[:link]->
    (a:Article{title:$s_title , lang:$s_lang})-[:link]->(b)
WITH n , COUNT(n) as cnt
RETURN  n ORDER BY cnt DESCENDING LIMIT $max
```

As before, we limit the number of nodes returned by first ordering them on the basis of how many times they were encountered during the query, which means that nodes with higher number of chain links to the source node is more likely to be a part of the candidate set. In the case where we are not able to hit the limit with the above query (set at 1000 nodes) we perform the following query to obtain $N(CL_t(w_\gamma))$ where $w_\gamma \in N_{out}(v_\alpha)$:

```
MATCH (n:Article)-[:link]-(c:Article{lang:$t_lang})
    -[:crosslink]-(b:Article)<-[:link]-
    (a:Article{title:$s_title, lang:$s_lang})
WHERE NOT (a)<-[:link]-(b)
WITH n , COUNT(n) as count
RETURN  n ORDER BY count DESCENDING LIMIT $max
```

Here we simply add the **WHERE NOT** clause to ensure that we do not reconsider nodes that are mutually linked to the source node (which were recovered in the previous query). Also, here we ensure that the **$max** is equal to the overall limit minus the candidates that were returned by the first query.

4.2 Target Article Selector

This class takes as input to its **__init** function a reference to the cypher session, the source node and the list of candidates. The method selectTargetArticle() iterates throughout the list of candidates and builds a similarity vector for each element, which is made in either the Jaccard or the WLM method. We wrote two different implementations for each measure, described in the following paragraphs. Subsequently, we compute the cosine similarity and choose the best candidate as an answer.

An important thing to note is that to translate the vectors between the source language and the target language, we decided to translate the source node to the target language. This is in conflict to what was described in the article [Penta et al., 2012], but it should give us the same results, and it is less costly to translate a single vector than translating all the candidates.

Classical WLM

We translate the formula of the WLM measure 1 into the syntax of a cypher query. We show below the case when the similarity vector to be computed is for a node v in the target language (one of the candidates).

```
Match (v:Article{title:$title, lang:$lang})<-[:link]-(i_in:Article)
with v, count(i_in) as fv
Match (v)-[:link]->(i:Article)
with v, i, fv
Match (i)<-[:link]-(i_out:Article)
with  count(i_out) as fi,fv, v, i
where fi > 0
Match(v)<-[:link]-(b:Article)-[:link]->(i)
unwind [log10(fv),log(fi)] as for_max
unwind [log10(fv),log10(fi)] as for_min
with count(b) as fvi, i, fv,fi,for_max,for_min
where fvi>0
return i.title,
(max(for_max)-log10(fvi))/(log10(126805)-min(for_min)) as score
```

By using "with" clause, we are able to save outputs of a "match" clause throughout the cascade of clauses, or even use it as input to the following "match". We thus can get the number of inward neighbors of v, fv, and, for every outward neighbor i of v, the number of its inwards neighbors fi, the number of common inwards neighbors of i and v, and finally the value of the corresponding element of the similarity vector. We return a 2-dimension matrix of records, where the first column has the name of each i and the second the corresponding score. The names are necessary because they will allow us to compute the cosine similarity of two vectors in the same basis, multiplying the elements corresponding to the same neighbors i in the target Wikipedia. The python structured used to store these pairs (name, value) was a dictionary.

In the case where the vector v is actually the source node, we perform the same query as above except that instead of returning the names of each i in the first column, we return the names of the corresponding translations of i in the target language using the existing cross-links in the graph. If the cross-link of a specific outward neighbor i is not present, it is just not taken into account in the similarity vector.

WLM v2

We decided to also experiment the same approach of classical WLM but replacing fi and fvi for respectively the number of outwards neighbors of i and the number of common outwards neighbors of both i and v. We further compare the perfomaces of both metrics in the "Results and Discussion" session.

Jaccard Index

The Jaccard index 3.2 is another index of similarity that we have implemented. Owing to its simplicity, the Jaccard index should be faster to calculate than the WLM index. However, its relative effectiveness will be discussed later in the results. We can implement the Jaccard index in two different ways, as described below.

First Implementation

The first implementation involves performing the following query for v_β and $i \forall i \in N_{out}(v_\beta)$, where v_β is every candidate in the list of candidates:

```
MATCH (a:Article{title:$title, lang:$lang})-[:link]->(b:Article)
WITH b OPTIONAL MATCH (b)-[:link]->(c:Article)
RETURN b.title, c.title
```

Hence, we return the titles of all the articles that point outwards of v_β and all the articles that point outward of each i. We do a similar query for the source node, but in this case we simply make sure that in addition to returning the titles of the neighbors of the source node, we also return the titles of the crosslinks of the neighbors. This allows us to perform the cross product between the vectors of different languages. The query for the source node is as follows:

```
MATCH (a:Article{title:$title, lang:$lang})-[:link]->(b)
    -[:link]->(c)
OPTIONAL MATCH(b)-[:crosslink]->(k:Article{lang:$t_lang})
RETURN b.title, k.title, c.title
```

On the python end, we perform the actual calculation of the Jaccard index, by calculating the union and intersection of the outward neighbours between v_β and i.

```
i_Ni_dict = {}
i_set = set()

for record in response.records():
    i_Ni_dict.setdefault(record[0], set())
    if (record[1] is not None):
        i_Ni_dict.setdefault(record[0], set()).add(record[1])
    i_set.add(record[0])

for i, Ni in i_Ni_dict.items():
    union = len(i_set | Ni)
    if (union != 0):
        simV[i] = len(i_set & Ni)/union
```

Here, **i_Ni_dict** is a dictionary with the key as the title of the neighbors (i) and the values as the outward neighbors of the neighbors ($N_{out}(i)$) stored in the form of a set. We also save the titles of the neighbors (i) in the form of a set. Using python sets is convenient as sets do not allow duplicates and also allow us to perform the union and intersection functions of their elements directly. In the case when we are dealing with the source node, we need to perform

the translation of the $SimV$ vector also. Hence, we keep another dictionary whose key is the titles of i and the values are the titles of $CL_t(i)$. Then while building $SimV$, we only perform the computation for i which have a crosslink. The code snippet for the same can be found below:

```
i_Ni_dict = {}
i_set = set()
i_cl_dict = {}
for record in response.records():
    if(record[1] is not None):
        i_cl_dict[record[0]]=record[1]
    i_Ni_dict.setdefault(record[0], set()).add(record[2])
    i_set.add(record[0])

for i, crosslink in i_cl_dict.items():
    union = len(i_set | i_Ni_dict[i])
    if (union != 0):
      simV[crosslink] = len(i_set & i_Ni_dict[i])/union
```

Second Implementation

The second approach simplifies things by putting the burden of the calculation of the vector completely on the Cypher query. For every candidate, we perform the following query:

```
MATCH (v:Article{title: $title, lang: $lang})-[:link]->(i:Article)
WITH v, i, COUNT(i) as Nv
MATCH (i)-[:link]->(i_out:Article)
WITH  COUNT(i_out) as Ni,Nv, v, i
MATCH (v)-[:link]->(b:Article)<-[:link]-(i)
WITH COUNT(b) as common, (Ni + Nv - COUNT(b)) as uni, i
WHERE uni>0
RETURN i.title, common*1.0/uni
```

In the case of the source node, we simply return the titles of $CL_t(i)$ along with the respective Jaccard index.

Cosine similarity

The cosine similarity between the source between every candidate and the source is computed by calculating the normalized dot product of each pair of vectors. As discussed above, we make sure to multiply elements corresponding to the same node, which is trivial when using python dictionaries. The threshold for a resulting score to be considered in the selection is 0.15. The candidate with the best score is given as answer to the translation.

Saving $SimV$ vectors

Seeing that the most computationally exhaustive step of the algorithm is the calculation of the $SimV$ vectors for each candidate, there is an immediate interest in saving the $SimV$ vectors that are calculated. However, these can only be saved for the target nodes, since the $SimV$

vector for the source node is translated to the target language and would change with every target language. The pickle library allows us to easily store the vectors that we have calculated in the form of a dictionary. However, we had to decide at what time in the algorithm we save the vectors. At first thought, the pickle file can be extended every time we calculate the new $SimV$ vector. However, reading and writing to a file can be slow and might lead to inefficiencies. Hence, we only read the pickle file once at the start of the **targetArticleSelector** and store it as a local dictionary. During the entire program, we keep adding to this dictionary and at the end of the program we simply rewrite the pickle file once. Once the reading and writing is implemented, it is simply a matter of checking whether the $SimV$ vector already exists in the dictionary before trying to calculate it using the Cypher queries. If id does, then we simply take the version in the dictionary. If it doesn't we calculate it and then add it to the dictionary.

5 Tests and Results

5.1 Testing candidateGenerator.py

Before continuing with the implementation of targetArticleSelector.py, it was important for us to test if our implementation of candidateGenerator.py was working as expected. Hence, we decided to build a python test file using the TestCase library provided with Python. Using This allowed us to write down test cases that would go through each condition of the code to check if it was working. For example we wrote test cases for each of the three types of nodes. But at the same time, within each type of node we ensured that we had differing examples which would run though different conditions as put inside the implementation of the candidate generator. After writing the test file, we would just need to run it once and it would go through all the test cases and give us the number of test cases that were failed and the number of test cases that failed. To pass a test case, the list of candidates returned had to include the gold candidate: the true translation of the node. The various test cases that were gone through are summarized below:

- Named, non-geographic node where the target node has the same title.
- Named, non-geographic node where the target node has a different title.
- Geographic node where the target node has the same title
- Geographic node where the target has the same lat/long coordinates
- Geographic node where the target node does not have lat/long coordinates.
- Geographic node which has no other nodes close to it geographically in the target language.
- Non-named node where 1000 candidates found with only 1 query.
- Non-named node where 1000 candidates found with both queries combined.
- Non-named node where less than 1000 candidates found with both queries.

It is through this test file we discovered examples of cases where the candidate generator is unsuccessful. Some of these are presented here:

- No candidates are generated when the source node is "Brazil" (English to French). This is because there is no node in the database which has a Geo-location within 3 miles of the Geo-location provided for the node of Brazil. We suspect this is primarily because our dataset is a subset of the original Wikipedia and only contains the nodes surrounding Paris.

- The candidate "Stylo" does not exist in the list of candidates generated for "Pen". This is because none of the mutual links of "Pen" have cross-links that link to "Stylo". We have confirmed this using the Neo4j browser to look through the data.

5.2 Testing targetArticleSelector.py

With the intent of testing the performance of our target selector algorithm (both result accuracy and time efficiency), we created a test routine called "stats.py". We start by selecting a subset N of neighbors v of Paris in the source language and its corresponding $\mathrm{CL}_t(v)$ to be the gold set. Since only a part of the french and english wikipedia(mostly centered around Paris) is available for us to test our algorithm, this selection is likely to avoid the problem of taking a node in the edge of the graph, what would make a translation very difficult and interfere in the statistical analysis.

For every v belonging to M we go through the entire process of generating candidates and selecting the target, timing these two steps. We keep track of the case it was assigned to in Candidate Generator (geographic,named non geographic, non-named) and whether if the gold translation was actually present in the pool of candidates C. In the testing we have only tried tranlating nodes from the English language to the French language. This corresponds to the harder translation since the crosslinks from English are sparser than French. In the two sections below we provide the results in terms of accuracy and time performance respectively.

Accuracy Results

In order to test the accuracy of the algorithm, we decided to use similar measurements pf *Precision* P, *Recall* R and *(*F-Measure) F as used by [Penta et al., 2012] which are redefined below:

$$P = \frac{C_X}{D_X} \qquad R = \frac{C_X}{X} \qquad F = \frac{2 * P * R}{P + R}$$

	N°of nodes				**Present**				**Correct**			
	NG	G	NN	Total	NG	G	NN	Total	NG	G	NN	Total
WLM	37	45	13	95	0.97	0.82	1	0.8	0.75	0.68	0.31	0.65
WLM V2	34	37	12	83	0.97	0.81	1	0.90	0.73	0.73	0.17	0.64
Jaccard	112	106	42	260	0.96	0.85	1	0.92	0.74	0.88	0.38	0.73

Table 1: **Accuracy**.NG: non-geographic, G:geographic, NN:non-named. *Correct* are the ratios between the number of correct cross-links found and the number of nodes present at the specific categories

In addition to the statistics above, we would like to bring to attention a few examples of the cases where the crosslink found while using the Jaccard index is not the same as the existing crosslink:

- Source Node: Al-Qaeda
  Existing Crosslink: Al-Qada
  Ans found: Terrorisme islamiste

- Source Node: Airline
  Existing Crosslink: Compagnie arienne
  Ans found: Alliance de compagnies ariennes

- Source Node: Fire Brigade
  Existing Crosslink: Brigade de sapeurs-pompiers de Paris
  Ans found: Fdration nationale des sapeurs-pompiers de France

- Source Node: SNCF
  Existing Crosslink: Socit nationale des chemins de fer franais
  Ans found: SNCF

- Source Node: Champs-lyses
  Existing Crosslink: Avenue des Champs-lyses
  Ans found: Cartier des Champs-lyses

- Source Node: Brussels
  Existing Crosslink: Rgion de Bruxelles-Capitale
  Ans found: Bruxelles

We observe that some of the unsuccessful translations by WikiCL are equaly good or even better than the existing cross-links(the case of Brussels, for exemple). Similarly, here are some examples of the same when we used the WLM (unmodified) measure.

- Source Node: Art
  Existing Crosslink: Art
  Ans found: Mouvement artistique

- Source Node: RATP Group
  Existing Crosslink: Rgie autonome de transports parisiens
  Ans found: RATP Dveloppement

As we can see, a large part of the incorrectly found translations are in fact very close to the gold translation that already exists. This shows the strength of the algorithm. Another measure that can help us understand better the performance of the algorithm is the top-K measure, which counts the percentage of times the correct crosslink was among the top K results that were returned by the algorithm. The table 2 summarizes the values obtained for this statistical measure.

	Precision	**Recall**	**F-measure**	**Top-1**	**Top-2**	**Top-3**	**Top-4**	**Top-5**
WLM	0.59	0.59	0.59	0.59	0.62	0.63	0.66	0.67
WLM V2	0.61	0.58	0.59	0.59	0.62	0.64	0.69	0.71
Jaccard	0.69	0.67	0.68	0.67	0.73	0.77	0.80	0.81

Table 2: **Statistical inference on the results**

As can be seen from the results, the Jaccard index outperforms both WLM and WLM V2. This is unexpected since the WLM measure was assumed to be a more comprehensive measure of similarity than the Jaccard index. However, to be sure of its relative performance, we need to check similar statistics for translation between languages other than English and French.

Another curious result is the comparison between WLM and WLM V2. THeir overall performance is very similar. However, WLM V2 out performs WLM in the case of geographic nodes. On the other hand, WLM is much ahead of WLM V2 in the case of non-named nodes. Also, it is worth noting that WLM V2 has much better $top - K$ statistics than WLM classic.

Time Performance

		N°of nodes				**Time (s)**			**Time with data (s)**		
	Recorded data	NG	G	NN	Total	CG	TS	Total	CG	TS	Total
WLM	0.054	7	7	1	15	1.826	1568.873	1574.524	2.28	4.07	10.48
WLM V2	0.22	7	7	1	15	1.77	765.65	771.25	2.45	5.49	12.07
Jaccard	0.36	112	106	42	260	2.20	883.53	889.56	2.69	3.83	10.71

Table 3: **Time performance**. *Recorded data* is the percentage of the nodes in the source Wikipedia that have their similarity vectors recorded in the picle file. We present the time taken by the algorithm without and with access to these data. NG: non-geographic, G:geographic, NN:non-named; CG: candidate generator, TS: target selector

With the results of the timings data, we can immediately see the difference in the performance of WLM and Jaccard in the absence of any pre-saved $SimV$ data. The Jaccard index is much faster than the WLM measure. This is primarily due to the way in which the Jaccard index is calculated. In Neo4J, going through relationships between nodes is extremely costly, and WLM requires us to travel many more relationships than Jaccard. Interestingly, the WLM V2 (modified version) not only performs slightly better as seen in the accuracy results, but also is much faster.

The presence of pre-saved $SimV$ vectors greatly improves the time performance of the algorithm, and the crosslinks are found in a matter of seconds. In fact, the majority of the time spent in this case is loading the data file into python. This was done once for every translation in order to stay true to the assumption that we ask for crosslinks one at a time. If we were to try and find multiple crosslinks between the same pair of languages, we would need to only load the data file once and this would bring down the time spent even more considerably.

6 Conclusion

The purpose of this project was to implement the WikiCL algorithm described by [Penta et al., 2012]. Our implementation in Python shows the strength of this algorithm in finding the crosslinks of Wikipedia articles. The strength of this algorithm lies in its independence from the language of the source and target nodes. This property allows it to be much more usable in the real world. Our implementation was able to give a satisfactory accuracy. We believe that the fact that we only used a subset of the Wikipedia centered around "Paris" affects our accuracy results and these statistics should only improve if used on the full version of the Wikipedia.
In terms of the time taken, our implementation falls severely behind the performance of the

authors of the algorithm. We believe that is is because of two primary reasons. Firstly, we used Python for the implementation instead of Java. The Neo4J Bolt driver for Python is comparatively much slower that the corresponding drivers for other languages such as Java. Secondly, since Python is an interpreted language, it is much slower than other compiled languages such as Java.

References

[Milne and Witten, 2007] Milne, D. and Witten, I. H. (2007). An Effective , Low-Cost Measure of Semantic Relatedness Obtained from Wikipedia Links. *Artificial Intelligence*, pp:25–30.

[Penta et al., 2012] Penta, A., Quercini, G., Reynaud, C., and Shadbolt, N. (2012). Discovering cross-language links in Wikipedia through semantic relatedness. *Frontiers in Artificial Intelligence and Applications*, 242:642–647.

Appendices