Project 11: Semantic Analysis

Dennis Neumann
Oulu, Finland
dennis2.neumann@student.uni-siegen.de

Muhammad Sheraz Khan
City, Pakistan
Sherazkhan1234@gmail.com

***Abstract*—Semantic similarity has been used for multiple tasks regarding natural language processing for many years now. Various static and lexica-based methods for analyzing semantic similarity such as Wu-Palmer and Leacock-Chodorow with different approaches and results exist to determine the similarity between given phrases. This project aims to show how the dynamic and web-based similarity methods of the WebJaccard similarity work in comparison to the static methods with the help of the Python Natural Language Toolkit and google search.**

***Index Terms*—semantic similarity, natural language processing, language processing, semantic, analysis, google search**

I. Introduction

In contrast to static semantic similarity analysis methods which are based on lexical search using WordNet, Wikipedia and text corpora [1], the WebJaccard similarity (WebSim) introduces a more dynamic way to determine the relation of words, since those can change the meanings over time, [2] for example the word *Tesla* nowadays can not only be associated with Nikola Tesla but also with the electric car brand of the same name.
All methods have their own approaches to calculate the similarities which can be overwhelming for those not knowing how well they perform on the same set of words. For a good comparison between the static approaches with the dynamic methods, the WebSim will include the page-count-based similarity as well as the procentual overlap similarity.

Chapter II introduces to the tasks for this project, the used tools and libraries and how the results wil be saved and visualized. Chapter III shows the implementation details. In chapter IV the achieved results are shown and discussed. Chapter IV concludes this project documentation and sums up the content.

II. Methodology

A. Tasks

B. synsets

According to the requirements for this project, the similarity methods have to be applied on at least ten synonym set (”synset”) pairs *(P,Q)*. . The pairs should include the three cases, in which a) *P* is a synonym of *Q*, b) *P* is an antonym of *Q*, c) *P* has an entailment relation to *Q*. For the tasks of this project, following synset pairs were selected:

- pleased - excited
- run - rush
- displeased - upset
- sleep - nap
- laugh - weep
- rich - poor
- drunk - sober
- mobile phones - cell phones
- introvert - extrovert
- huawei - iphone
- run - sweat
- contested - won
- fell - broken

Many synsets are not applicable to use together in some of the similarity methods, since they are not sharing common hypernyms [3]. Therefore, the choice of synset pairs is very important for applying and testing them against all similarities. This issue is discussed further in chapter III-E.

C. Tools and Libraries

In this project, following tools and packages were used for corresponding purposes:

- NLTK : ...

III. Implementation

A. Preliminary Work

For the purposes of understanding and for avoiding recurring code snippets, this short chapter shows code snippets related to the implementations in the following chapters.

1) Visualization of Synsets and Results: To visualize the similarity results for the synsets defined in II, the synset pairs will be saved to the table *pair_words_df* via the *DataFrame* Python class. For every similarity,

```
sim_layer = [['pleased', 'Excited'], ['run
    ', 'rush'], ...]
...
pair_words_df = pd.DataFrame(sim_layer,
    columns=['P', 'Q'])
pair_words_df['Column␣name'] = ''

for i in range(len(synsets)):
   pair_words_df.iloc[i,k] = ...
```

2) Pre-Processing function: In case of synsets of word length¿1, pre-processing should be performed. It has to contain the conversion in lower case, removal of special characters and lemmatizer for removing words with only one character as well as stopwords. The code snippet 1 demonstrates the

important steps taken for the pre-processing with the help of the Regular expression operations package **re**.

```
def pre_processing(text):

    text = text.split('␣')
    text = [word.strip().lower() for word
        in text if word.lower() not in
        stopwords]
    rx = re.compile('([&#.:?!-()])*')
    text = [rx.sub('', word) for word in
        text]

# selecting only the alpha bets and words
    length greater than 1
    text = [word for word in text if len(
        word)>1 and word.isalpha()]
# if some word appear more than 1, remove
    others
    unique = set(text)
    unique = [lemmatizer.lemmatize(word)
        for word in unique]

# storing after processing in third column
    return '␣'.join(text)
```

Listing 1. Pre-processing for synsets of word length¿1

B. WebJaccard page count similarity

The first task consisted of implementing the WebJaccard similarity based on page count and calculate the similarity coefficient. Where *H(P)* in 1 are the results of the pair key P and *H(Q)* the results of pair key Q, *H(PQ)* are the results when searching for P and Q together. This similarity coefficient is calculated as followed in figure 1:

$$\text{Sim}(P,Q) = \begin{cases} 0 & \text{if } H(P \cap Q) \leq c \\ \frac{H(P \cap Q)}{H(P)+H(Q)-H(P \cap Q)} & \text{otherwise} \end{cases}$$

Fig. 1. Formula for page-count-based WebJaccard similarity [2].

For the implementation we used the Google Search API. To use the *search* function properly, some parameters have to be considered as seen in 2: whereas *num* is the number of results and the maximum number is 10, the maximum request number per day is 100. The *pause* between every requests also needs to be adjusted as a too high amount might slow down the process too much and a too low number might end up in an IP block by Google [4].

```
def google_search(query):
    return search(query, lang = 'en', num =
        10, start = 0, stop = None, pause
        = 2.0)
```

Listing 2. Google search function

Now the function can be used to get the results for each pair (P,Q). To get the page count, the *len()* function is used on the returned *list* of results (see 3). Once each *H(P)*, *H(Q)* and *H(PQ)* results are saved in according variables *count_p*, *count_q* and *count_pq*, they can be used to calculate the Jaccard similarity coefficient. The *threshold* is used to determine the minimum number of results. If the number of results is less than the threshold, the coefficient is 0 [2]. For the chosen synset pairs, there is no need for pre-processing the input since they only contain one word. Since the google API has a limitation of searching for one pair per hour, the values of the WebJaccard similarity had to be calculated one after another. To save the time, the calculated values were saved inthe array *web_results*.

```
def sim(P, Q, threshold, pre_process=False
    ):
    if pre_process:
        P = pre_processing(P)
        Q = pre_processing(Q)

    results_p = google_search(P)
    count_p = len(list(results_p))

    results_q = google_search(Q)
    count_q = len(list(results_q))

    results_pq = google_search('{}␣AND␣{}'.
        format(P,Q))
    count_pq = len(list(results_pq))

    if count_pq <= threshold:
        return 0

    else:
        return (count_pq) /(count_p +
            count_q - count_pq)
...
web_results = [0.700, 0.960, 0.564, ...]
```

Listing 3. Getting google results and calculation of WebJaccard similarity

C. Wu-Palmer-, Path- and Leacock-Chodorow-Similarity

Unlike the WebJaccard similarity which has to be implemented manually, the WordNet Python library has predefined functions for the WuPalmer-Similarity (WUP) as well as the Path-Similarity (PATH) and Leacock-Chodorow-Similarity (LCH). The code snippet 4 demonstrates the use of the *wup_similarity()*, *path_similarity()* and *lch_similarity()* functions on the self defined synsets from II. The only pre-processing which has to be taken care of is to convert all synset's characters to lower case.

```
def wordnet_sim(P, Q):
    P=wordnet.synsets(P.lower())
    Q=wordnet.synsets(Q.lower())
```

```
   for i in range(len(P)):
      for k in range(len(Q)):
...
         chod_sim = P[i].lch_similarity(Q[
            k])
...
         wup.append(P[i].wup_similarity(Q[
            k]))
         path.append(P[i].path_similarity(
            Q[k]))
```

Listing 4. Use of WUP, PATH and LCH

D. Snippet Similarity

Besides the pagecount based similarity, the WebSim also includes a websearch snippet based similarity (SnippetSim) which is calculated as followed:

(TODO)

Where x is the number of common words, y is the number of combined unique words.

To achieve it, Google's Custom Search JSON API was used to retrieve the search results [5]. First an API key has to be created [5] and a Custom Engine ID (*cx*) generated. These have to be inserted in the corresponding *build()* function to make the source code work. The implementation is shown in the following code snippet 6:

```
def google_snippets(query, num=10):

   query_service = build(serviceName="
      customsearch", version="v1",
      developerKey='...')

   query_results = query_service.cse().
      list(q=query,cx='...', num=num).
      execute()
    return query_results['items']
```

Listing 5. Connection to Google Search API

For this similarity, the preprocessing function 1 from chapter III-A2 has to be used. For the calculation of the SnippetSim in this project, the first 5 results were taken. Based on the formula shown in (TODO) the implementation of the SnippetSim looks roughly as followed in 6):

```
def sim_snippets1(p, q):
  for i in range(5):
      snippets_p += pre_processing(p)
...
      snippets_q += pre_processing(p)
...
   common_words = len(set(snippets_p.strip
      ().split('␣')) & set(snippets_q.
      strip().split('␣')))
   combined_unique_words = len(set(
      snippets_p + snippets_q))

  return common_words/
     combined_unique_words
```

Listing 6. Calculation of SnippetSim

E. Snippet Overlapping Similarity

Another similarity using snippets is based on overlapping percentage between two phrases. In this paper, it will be shortened as OLSim. This task needs a special pre-processing technique called the Levenshtein distance, also called editing distance. It calculates the number of operations needed to change one phrase to another by changing one character after another. The allowed operations are insertions, deletions and substitutions [6]. The formula of the Levenshtein distance is described as followed in picture 2:

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j)+1 \\ \mathrm{lev}_{a,b}(i,j-1)+1 \\ \mathrm{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Fig. 2. Formula for Levenshtein distance [7].

The first row corresponds to deletion, the second row to insertion and the third row to substitution [7]. The Levenshtein ratio is the corresponding similarity ratio derived from the Levenshtein distance [7] and is defined as followed in 3:

$$\frac{(|a|+|b|) - lev_{a,b}(i,j)}{|a|+|b|}$$

Fig. 3. Formula for Levenshtein similarity ratio [7].

Similar to SnippetSim, the implementation of the OLSim is roughly as followed in code snippet **??**:

```
def sim_snippets2(p, q):
   for i in range(10):
...
      snippets_p += pre_processing(p)
...
      snippets_q += pre_processing(q)
...
   return round(lev.ratio(snippets_p,
      snippets_q),4)
```

Listing 7. Calculation of OLSim

F. Human Judgement Score

IV. Results and discussions

WebSim and SnippetSim also work on buzzwords like "Huawei - iPhone" and "Trump - Biden", while WordNet are not able to do that.

As mentioned in chapter II, many pair words exist in which the combination of such do not work together, which causes some of the methods showing the similarity results as "None"

[3], or in the case of LCH the pairs cause a compilation error ”Computing the lch similarity requires synset1 and synset2 to have the same part of speech” [8]. So one of the challenges with the tasks in chapter III-E and III-B

V. Conclusion

In this project the

References

[1] K. Radinsky, E. Agichtein, E. Gabrilovich, and S. Markovitch, “A word at a time: Computing word relatedness using temporal semantic analysis,” January 2011, pp. 337–346.

[2] D. Bollegala, Y. Matsuo, and M. Ishizuka, “Websim: a web-based semantic similarity measure,” 01 2007.

[3] J. Perkins, *Python 3 Text Processing with NLTK 3 Cookbook*. CreateSpace Independent Publishing Platform, August 2014. [Online]. Available: https://books.google.fi/books?id=eCHarQEACAAJ

[4] M. Vilas, “Welcome to googlesearch’s documentation!” https://python-googlesearch.readthedocs.io/en/latest/, 2018, accessed on 01.11.2020.

[5] G. Developers, “Custom search json api,” https://developers.google.com/custom-search/v1/overview, 2020, accessed on 01.11.2020.

[6] G. Navarro, “A guided tour to approximate string matching,” vol. 33, no. 1, p. 31–88, March 2001.

[7] F. J. C. Arias, “Fuzzy string matching in python,” https://www.datacamp.com/community/tutorials/fuzzy-string-python, February 2019, accessed on 04.11.2020.

[8] N. Project, “Source code for nltk.corpus.reader.wordnet,” https://www.nltk.org/_modules/nltk/corpus/reader/wordnet.html, 2020, accessed on 04.11.2020.