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 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 WebJaccard similarity will include the page-count-based similarity as well as the procentual overlap similarity.
Chapter II introduces to the tasks for this project and the methodology in which they shall be approached. Chapter III shows the implementation details

II. Methodology

A. Used libraries

As the base for accomplishing the tasks, To illustrate the

B. Used libraries

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 word pairs 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-B.

III. Implementation

A. 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 III-A 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:

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

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 1: 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 1. 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 2). 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, otherwise the similarity will be determined as 0. For the chosen synset pairs, there is no need for pre-processing the input since the synsets only contain one word.

```
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)
```

Listing 2. Getting google results and calculation of WebJaccard similarity

For completeness it should be mentioned, that 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. The code snippet 3 demonstrates the important steps taken for the pre-processing.

```
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 3. Pre-processing for synsets of word length¿1

The use of this function is shown in III-A: To apply the function on the decided word pairs defined in II and visualize them with their results of the WebJaccard similarity, first the pairs are saved in an array which is here called *sim_layer*. Then the pairs are saved to a table *pair_words_df* via the *DataFrame* Python class. Since the google API has a limitation of searching for one pair per hour, the values of the calculated similarity had to be calculated and saved as pre-determined values in an array called *web_results*.

```
sim_layer = [['pleased', 'Excited'], ['run
    ', 'rush'], ...]
web_results = [0.700, 0.960, 0.564, ...]

pair_words_df = pd.DataFrame(sim_layer,
    columns=['P', 'Q'])
pair_words_df['Websim'] = ''

for i in range(len(web_results)):
    pair_words_df.iloc[i,2] = web_results[i
        ]
```

B. 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 pseudo code snippet 4 demonstrates the use of the *wup_similarity()*, *path_similarity()* and *lch_similarity()* functions on the self defined synsets from II. The methods here

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

Listing 4. Use of WUP, PATH and LCH

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" [5]. So one of the challenges with the tasks in chapter III-B and III-A

IV. Results and discussions

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

V. Conclusion

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*. Packt Publishing, August 2014. [Online]. Available: https://subscription.packtpub.com/book/application_development/9781782167853/1/ch01lvl1sec16/calculating-wordnet-synset-similarity

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

[5] 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.