Project 11: Semantic Analysis 1

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

Muhammad Sheraz Khan
Oulu, Finland
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 dynamic and web-based similarity methods work in comparison to the static methods with the help of the Python Natural Language Toolkit and Google Search API.**

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

I. Introduction

Over many decades, for many tasks regarding semantic similarity there have been methods introduced. Based on the proposal of Rada et. al. [1], similarity of synonym sets (synsets) describes the minimum distance from one node to another in a taxonomical tree. Since this approach assumes, that the links in the taxonomy represent uniform distances [2] between two words, some approaches derive their own measurement methods which try to solve this problem. Wu-Palmer similarity (WUP) [3], Path similarity (PATH) [2] and Leacock-Chodorow Similarity (LCH) [4] proposed their own approaches and use the English database WordNet, which groups English words into synonym sets (synsets) according to similar meanings and lexical relations [5]. WUP considers the depth in two synsets, as concepts of words become more similar the lower they lie in the ontological hypernim tree [6]. PATH calculates the shortest distance between the synsets, but other than the solution of Rada et. al, it considers the fact, that some sub-trees of a taxonomy are deeper than others [2]. LCH calculates the overall shortest path between two given concepts [7] to their common hypernyms in the taxonomy tree [8].
However, those are static methods and only work on formal words, not considering slang or buzzwords. Moreover, meanings of words or relations to such can change over time [9], 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. Therefore, this project aims to show some dynamic approaches like the WebJaccard similarity (WebSim). Those are based on web search engines and try to determine the correlation between the given words or phrases according to the web search results.
Chapter II introduces to the used tools and libraries. 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. Word pairs

According to the requirements for this project, the similarity methods have to be applied on at least ten synonym sets 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 word 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 [8]. Therefore, the choice of word pairs is very important for applying and testing them against all similarities. This issue is discussed further in chapter III-D.

B. Tools and Libraries

- **Natural Language Toolkit (NLTK)**: For overall processing human language words
- **Google Search API** and **Custom Search API**: For later discussed websearch-based similarity methods
- **Levenshtein**: Python package for calculation of Levenshtein distance
- **re**: Regular expression package to process regular expressions
- **Matplotlib**: Calculate data relations
- **Seaborn**: Visualization of calculated data relations
- **Pandas**: For implementation of tables
- **Tkinter**: Python Interface programming

III. Implementation

A. Preliminary Work

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

1) Gihub Repository: The source code and this report are available in the following Github repository: https://github.com/dneumann20/NLPproject.

2) Visualization of Synsets and Results: To visualize the similarity results for the word pairs defined in chapter II, the word pairs are 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] = ...
```

3) 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 python 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 consists 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:

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

Fig. 1. Formula for page-count-based WebJaccard similarity [9].

For the implementation the Google Search API was used. For proper use of the *search()* function, 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 [10].

```
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 [9]. For the chosen word 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 WUP as well as PATH and LCH. This way, *wup_similarity()*, *path_similarity()* and *lch_similarity()* can be easily used on the self defined synsets from II. The only preprocessing which has to be taken care of is to convert all synset's characters to lower case.
As seen in picture 2, WUP takes the depths of each synset as well as their least common subsumer (LCS) in account [11]:

$$\mathbf{Wu-Palmer} = 2 * \frac{\mathbf{depth}\,(\mathbf{lcs}(\mathbf{s1}, \mathbf{s2}))}{(\mathbf{depth}\,(\mathbf{s1}) + \mathbf{depth}\,(\mathbf{s2}))}$$

Fig. 2. Formula of WUP [11].

LCH as a derived form of PATH is calculated as a negative log of shortest path (spath) of two phrases [11]. The formula is described as followed in pciture 3:

$$\boldsymbol{LCH\ Similarity} = -log\frac{\boldsymbol{spath}\,(\boldsymbol{synset1},\ \boldsymbol{synset2})}{2 * \boldsymbol{Depth}}$$

Fig. 3. Formula of LCH [11].

Code snippet 4 shows the use of the predefined similarity functions mentioned above.

```
P.lch_similarity(Q)
P.wup_similarity(Q)
P.path_similarity(Q)
```

Listing 4. Use of WUP, PATH and LCH

D. Snippet Similarity

For this task, the similarity between multiple snippets (SnippetSim) from google search have to be calculated. To retrieve the results, Google's Custom Search JSON API was used. First an API key has to be created and a Custom Engine ID (*cx*) generated (see code snippet 5). These have to be inserted in the corresponding function *build()* and class *Customsearch.Cse.List* to make the source code work.

```
def google_snippets1(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-A3 has to be used. For the calculation of the SnippetSim in this project, the first 5 results were taken. 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 report, it will be shortened as OLSim. This task needs a special pre-processing technique called the Levenshtein distance [12], 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 [13]. The formula of the Levenshtein distance is described as followed in picture 4:

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

Fig. 4. Formula for Levenshtein distance [14].

The first row corresponds to deletion, the second row to insertion and the third row to substitution [14]. The Levenshtein ratio is the corresponding similarity ratio derived from the Levenshtein distance [14] and is defined in picture 5:

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

Fig. 5. Formula for Levenshtein similarity ratio [14].

Similar to SnippetSim, the implementation of the OLSim is roughly as followed in code snippet 7:

```
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

In this task, the human judgement score (HJS) according to Miller & Charles (MC-28) [15] was taken to calculate the correlation to the similarity methods from chapter III-D by using the Pearson coefficient. Exceptionally for this case, the following five pairs from the openly available datasets MC-28 [15] were used:

- automobile - car
- gem - jewel
- journey - voyage
- boy - lad
- coast - shore

Similar to III-A2, a table is created in code snippet 8 and the mentioned dataset is embedded.

```
df_mc = pd.read_csv('mc.csv', sep=';',
    names=['Word␣1', 'Word␣2', 'Human␣
    Judgement␣Score'])
df_rg = pd.read_csv('rg.csv', sep=';',
    names=['Word␣1', 'Word␣2', 'Human␣
    Judgement␣Score'])
df_wordsim = pd.read_csv('wordsim.csv',
    sep=';', names=['Word␣1', 'Word␣2', '
    Human␣Judgement␣Score'])
...
data = pd.concat([df_mc, df_rg, df_wordsim
    ])
data.head()
```

Listing 8. HJS correlation with WUP, PATH, LCH

Although the final output shows the mentioned pairs earlier in this chapter, it also shows the way how other datasets like RG (Rubenstein and Goodenougth, 1965) or WordSim353 (Finkelstein et al., 2001) can be embedded as well.
To calculate and plot each correlation from the table created in the code above, the function in code snippet 9 was implemented. For the visualization, the *scatter* plot was used and the *despine* function for the removal of unnecessary axis. After that the correlation can be calculated with the *corr()* function.

```
import matplotlib.pyplot as plt
import seaborn as sns

def plotting_correlation(data, col1, col2)
    :
    ...
   plt.scatter(data[col1], data[col2])
   sns.despine()
   corr = round(data[[col1, col2]].corr().
      iloc[0,1], 3)
  ...
   plt.show()
```

Listing 9. Correlation calculation function

G. Similarity GUI

This project also includes a GUI to run all previously implemented similarity methods by clicking corresponding buttons. The source code for the GUI implementation included in the appendix chapter A.

IV. Results and discussions

A. Results of web-based and lexica-based Solutions

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” [8], 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” [16]. So one of the challenges with the tasks in chapter III-D and III-B was to find the right word pairs.
As shown in picture 6, WebSim outperforms WUP and PATH most of the time, LCH being an exception of which the results compared to WebSim. Moreover, WebSim works also on buzzwords like ”Huawei - iPhone” while WUP, PATH and LCH are not able to use them. SnippetSim overall performs poorly and is not resulting in any similarity on most of the pairs, as the ones with 0 as resulted values seem to have too many unique or too less common words after pre-processing. Other than that, OLSim is results in values which are not too far or in between the values of the static similarity methods, except from ”contected - won” which seems to need too many steps of the Levenshtein distance to process.

	P	Q	Websim	Wu & Palmer	Path_length	Leacock Chodorow	Sim_snippet1	Sim_snippet2
0	pleased	Excited	0.70000	0.4000	0.2500	1.4663	0.000000	0.3797
1	run	rush	0.96000	0.9565	0.5000	1.4404	0.000000	0.4527
2	displeased	upset	0.56400	0.5000	0.3333	1.6487	0.040000	0.4275
3	sleep	nap	1.01530	1.0000	1.0000	1.3350	0.000000	0.4369
4	laugh	weep	1.47200	0.3333	0.3333	2.1595	0.000000	0.4136
5	rich	poor	0.64100	0.8000	0.3333	2.5390	0.000000	0.4327
6	drunk	sober	1.15200	0.4000	0.2500	1.6487	0.074074	0.4233
7	run	sweat	1.28272	0.7059	0.2000	0.8044	0.000000	0.4313
8	contested	won	1.04663	0.1818	0.1000	0.9555	0.041667	NaN
9	fell	broken	1.66197	0.8571	0.5000	1.1787	0.000000	0.4495
10	huawei	iphone	1.80282	0.0000	0.0000	0.0000	0.034483	0.3091
11	mobile phones	cell phones	1.13021	0.0000	0.0000	0.0000	0.041667	0.5173
12	introvert	extrovert	1.18848	0.7500	0.3333	2.5390	0.076923	0.4708

Fig. 6. Results for self defined word pairs on similarity methods from chapters III-B - III-E.

B. Human Judgement Score and Correlations

As seen in picture 7, while the results of WUP and PATH differ from the HJS values, LCH comes more closer to latter.

	Word 1	Word 2	Human Judgement Score	Wu & Palmer	Path_length	Leacock Chodorow
0	automobile	car	3.92	1.0000	1.0	3.6376
1	gem	jewel	3.84	1.0000	1.0	1.5581
2	journey	voyage	3.84	0.9524	0.5	2.2513
3	boy	lad	3.76	0.9474	0.5	2.5390
4	coast	shore	3.70	0.9091	0.5	2.9444

Fig. 7. Result table of word pairs from MC-28 including WUP, PATH and LCH.

The correlation data, as resulted according to picture 8 between WUP and HJS is very distributed without any noticeable outliers and the correlation between the measurements is very varying. The resulted correlation coefficient of 0.225 shows a lesser correlation rate between both methods.

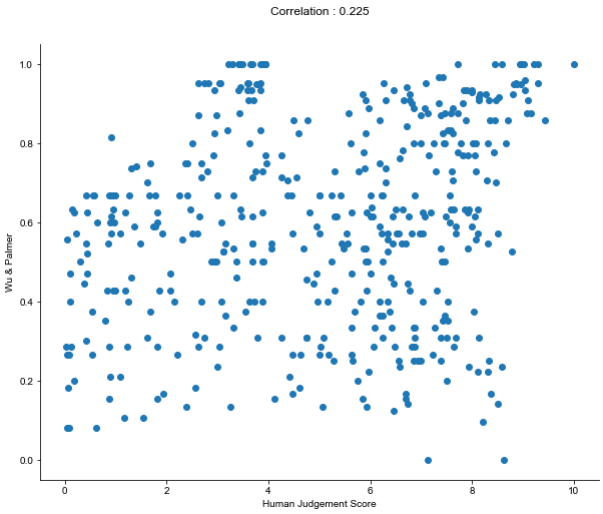

Fig. 8. Correlation between HJS and WUP.

The correlation results beween PATH and HJS, as shown in 9, are noticeably sparsely and linearly distributed in the range between 0 - 0.5 with some positive outliers, and with a correlation coefficient of 0.187 the correlation between those measures are less likely.

Fig. 9. Correlation between HJS and PATH.

The correlation results beween PATH and HJS, as shown in 10, are again more distributed and with a few outliers in higher values and 0. With a correlation rate of 0.152 however, the correlation coefficient proves a very low correlation rate.

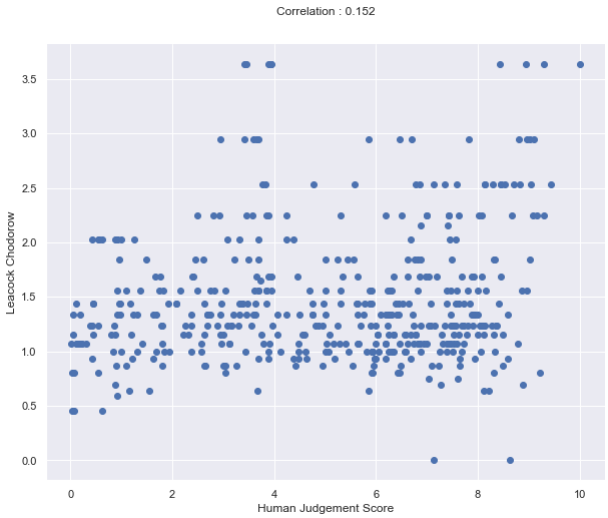

Fig. 10. Correlation between HJS and LCH.

V. Conclusion

In this project websearch-engine-based dynamic similarity methods were implemented and discussed in comparison to static lexica-based methods. HJS shows a low correlation rate to the static methods, proving that they are less reliable

similarity measurements.. The page-count-based WebSim outperforms the static methods WUP and PATH in most cases and can also measure buzzwords. SnippetSim is not a recommendable measurement method for similarity, as it fails on most word pairs with resulting values of 0. OLSim is more comparable to WUP and PATH and can also use buzzwords. It should be also assumed, that even a high similarity between two pairs can end up in having a bad result if the snippets contain the same contexts but different wordings, as the websearch results are always changing over time. Furthermore, getting the needed websearch results can take time due to the Google Search API'S limit of 100 requests per day, which is the biggest disadvantage of the dynamic methods.

REFERENCES

[1] R. Rada, H. Mili, E. Bicknell, and M. Blettner, "Development and application of a metric on semantic nets," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 19, pp. 17 – 30, 02 1989.

[2] P. Resnik, "Using information content to evaluate semantic similarity in a taxonomy," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, p. 448–453.

[3] Z. Wu and M. Palmer, "Verbs semantics and lexical selection," 01 1994, pp. 133–138.

[4] C. Leacock and M. Chodorow, *Combining Local Context and WordNet Similarity for Word Sense Identification*, 01 1998, vol. 49, pp. 265–.

[5] C. Fellbaum, *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[6] D. Guessoum, M. Miraoui, and C. Tadj, "A modification of wu and palmer semantic similarity measure," 10 2016.

[7] T. Slimani, "Description and evaluation of semantic similarity measures approaches," *International Journal of Computer Applications*, vol. 80, no. 10, p. 25–33, Oct 2013. [Online]. Available: http://dx.doi.org/10.5120/13897-1851

[8] 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

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

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

[11] M. Gupta, "Nlp — wupalmer – wordnet similarity," https://www.geeksforgeeks.org/nlp-wupalmer-wordnet-similarity/, January 2019, accessed on 06.11.2020.

[12] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet physics. Doklady*, vol. 10, pp. 707–710, 1965.

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

[14] 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.

[15] G. A. Miller and W. G. Charles, "Contextual correlates of semantic similarity," *Language and Cognitive Processes*, vol. 6, no. 1, pp. 1–28, 1991.

[16] 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.

APPENDIX A
SOURCE CODE OF SIMILARITY GUI

```
import tkinter as tk
from tkinter import *
import pandas as pd
#from tkinter.ttk import Progressbar

from implementation import sim,
    wordnet_sim, sim_snippets1
from implementation import google_snippets
    , sim_snippets2
#import sys

from pandastable import Table

def get_values():
    word1, word2 = entry1.get(), entry2.get
        ()

    if len(word1) ==0:
        tk.messagebox.showinfo('Error', '
            Please␣Enter␣First␣Word␣before␣
            finding␣similarity')
        return None, word2

    if len(word2) ==0:
        tk.messagebox.showinfo('Error','
            Please␣Enter␣Second␣Word␣before␣
            finding␣similarity')
        word2=None

    return word1, word2


def wordsim(event=None):

    word1, word2 = get_values()
    if word1 and word2:
        value = sim(word1, word2, 5)
        tk.messagebox.showinfo('Similarity',
            'Word␣Sim␣Score␣:␣', round(
            value, 3))


def net_sim(event=None):

    word1, word2 = get_values()
    if word1 and word2:
        a,b,c = wordnet_sim(word1, word2)
        tk.messagebox.showinfo('Wordnet␣
            Similarity','WUp:␣{}␣,␣Path:␣{}␣
            ,␣Chod:␣{}'.
                        format(a,b,c))

def snippet1(event=None):
```

```
    word1, word2 = get_values()
    if word1 and word2:
        snip_p = google_snippets(word1)
        snip_q = google_snippets(word2)

        value = sim_snippets1(snip_p, snip_q
            )
        tk.messagebox.showinfo('Snippet␣
            Similarity','Sim␣SNippet␣1␣Score
            :␣{}'.
                        format(round(value,
                            2)))



def snippet2(event=None):

    word1, word2 = get_values()
    if word1 and word2:
        snip_p = google_snippets(word1)
        snip_q = google_snippets(word2)
        value = sim_snippets2(snip_p, snip_q
            )
        tk.messagebox.showinfo('Snippet␣
            Similarity','Sim␣SNippet␣2␣Score
            :␣{}'.
                        format(round(value,
                            2)))



def exit():
    root.destroy()



def gui():
    global root, entry1, entry2

    root = tk.Tk()

    lbl=Label(root, text="Simantic␣
        Similarity", fg='#37251F', font=("
        Helvetica", 20))
    lbl.place(x=130, y=30)

    lb_e1=Label(root, text="First␣Word:␣",
        fg='#37251F', font=("Helvetica",
        10))
    lb_e1.place(x=90,y=90)

    entry1 = tk.Entry(root)
    entry1.place(x=165, y=90, height=30,
        width=200)

    lb_e2=Label(root, text="Second␣Word:␣",
         fg='#37251F', font=("Helvetica",
        10))
    lb_e2.place(x=70,y=140)
    entry2 = tk.Entry(root)
    entry2.place(x=165, y=140, height=30,
        width=200)

    x=172
    button1 = tk.Button(root, text='Web␣Sim
        ', command=wordsim, fg='white',bg='
        black', activebackground='#4F2619',
         height=3,
                width = 25)
    button1.place(x=x, y=190)

    button2 = tk.Button(root, text='Wordnet
        ␣Sim', command=net_sim, fg='white',
        bg='black', activebackground='#4
        F2619', height=3,
                width = 25)
    button2.place(x=x, y=260)

    button3 = tk.Button(root, text='Sim␣
        Snippet␣1', command=snippet1, fg='
        white',bg='black', activebackground
        ='#4F2619', height=3,
                width = 25)
    button3.place(x=x, y=330)

    button4 = tk.Button(root, text='Sim␣
        Snippet␣2', command=snippet2, fg='
        white',bg='black', activebackground
        ='#4F2619', height=3,
                width = 25)
    button4.place(x=x, y=400)

    button5 = tk.Button(root, text='Exit',
        command=exit, fg='white',bg='black'
        , activebackground='#4F2619',
        height=3,
                width = 25)
    button5.place(x=x, y=470)

    root.title('Application')
    root.geometry("500x650+300+20")
    root.mainloop()


if __name__ == '__main__':

    gui()
```