NATURAL LANGUAGE PROCESSING
BITI 3413

FINAL REPORT

TITLE:
TEXT SIMILARITY

LECTURED BY:
TS DR HALIZAH BINTI BASIRON

GROUP BY:

NO	NAME	MATRIC NO
1	ELLE ALIZ BINTI AMINUDDIN	B031810295
2	GION MIN MING	B031810222
3	THITTHIMA A/P WAT	B031810361
4	ZAITI AKTA BINTI ZAHARUDDIN	B031810365

Table of Contents

Introduction

Natural Language Processing (NLP) is one of the main components of Artificial Intelligence (AI), which has the potential to allow computers to comprehend human language. Be it ratings, comments, tweets, blogs, etc., wherein a significant amount of data is in natural language, a lot of data is generated in unstructured format. NLP helps computers to grasp and derive patterns from text like that. Through applying different techniques such as text similarity, information retrieval, labeling of texts, object extraction, clustering, NLP enables computers to understand and extract patterns from such text data.

Text Similarity is one of NLP's basic methods used by its context or surface to find the closeness between two chunks of text. To perform any machine learning process, computers need data to be translated into a numeric format. Different word embedding techniques are used to perform certain functions, i.e., Bag of Terms, TF-IDF, word2vec, to encrypt text details.

Analysis

The motivation behind the use of NLP methods to pick informative terms in a text is that a word token's meaning depends on both its type and the particular linguistic context in which it appears. Syntactic analysis, under the presumption that there is a regular mapping between the meaning of a text and its syntactic structure, is a computer-efficient first step to classify which words contain content full details in the document. Due to the shortcomings of existing NLP technologies, we prefer to use certain pieces of a syntactic analysis that can be reliably done on a wide scale. Therefore, as the identification of phrases is specific, we tag the sentences, extract heads of phrases, and classify subjects and objects, a role that can take advantages of the very set English word order, particularly for subjects.

A term that is out of place does not include information in the text on its prominence. A method of selection based on salience with respect to the inventory of classes of words is the normal procedure of taking only material words into account: noun, verbs, adverbs, adjectives, and prepositions are salient classes, whereas other such as determiners, are not. The semantic and pragmatic prominence of a word, however, is primarily determined in a sentence in relation to its syntactic salience. In more or less salient positions, words from content groups can appear. For example, the computer is more prevalent than the pocket in the term pocket computer, as the pocket computer is a kind of device.

Linguistic heads occur, generally speaking, in semantically more prominent positions than non-head words. Therefore, we recommend using a text representation consisting of the linguistic heads of the words in the material. We thus define the head of the main material sentences in order to decrease the dimensionality of the text representation without missing too much content. Specifically, we consider the heads of NPs and VPs in the way the syntactic form expresses them. For example, although a new corporation is a form of company, New York is not a type of York; British Petrol is similarly an essential part of the correct noun as the head and is therefore used in the representation of the text.

Design

Flowchart

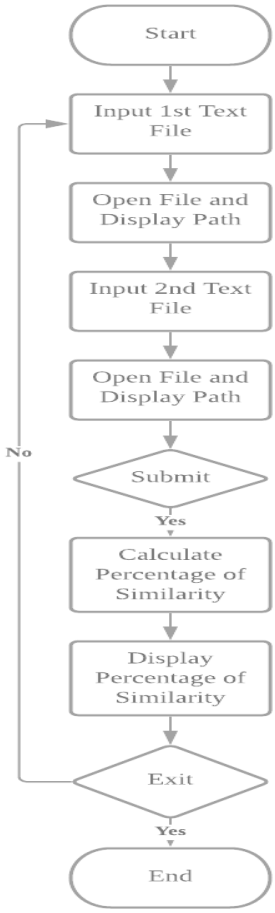

Figure 1: Flowchart

Pseudocode

```
Start

Input 1st Text File;

then File Open and Path Display;

Input 2nd Text File;

then File Open and Path Display;

If (Submit==yes);

    Calculate Percentage of Similarity;

    then Display Percentage of Similarity;

else

    exit;

End
```

Interface

Figure 1.2: Interface

In this system, the interface was made by using Tkinter. Tkinter is a framework provides Python users with a simple way to create GUI (Graphical User Interface) elements using the widgets found in the Tk toolkit. Tk widgets can be used to construct buttons, menus, data fields, etc. in a Python application. It is the standard Python interface to the Tk GUI toolkit.

Implementation

```
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
from tkinter import *

# import filedialog module
from tkinter import filedialog
# Create the root window
window = Tk()

# Function for opening the
# file explorer window
def browseFiles():
    global filename
    window.filename = filedialog.askopenfilename(initialdir = "/",
                                          title = "Select a File",
                                          filetypes = (("Text files",
                                                        "*.txt*"),
                                                       ("all files",
                                                        "*.*")))
    filename=window.filename

    # Change label contents
    label_file_explorer.configure(text="File Opened: "+filename)
    return (filename)

def browseFiles2():
    global filename2
    window.filename2 = filedialog.askopenfilename(initialdir = "/",
                                          title = "Select a File",
                                          filetypes = (("Text files",
                                                        "*.txt*"),
                                                       ("all files",
                                                        "*.*")))
    filename2=window.filename2


    # Change label contents
    label_file_explorer2.configure(text="File Opened: "+filename2)
    return (filename2)
```

Figure 1.3: Code Snippet 1

Firstly, we import all the libraries needed in this system. Then we started the code with define the function to input files. In this code, it only can accept text file (.txt).

```
def similar():
    global percentage_of_similarity
    #File 1
    file_docs = []
    # file1=input("Enter the file:")
    f = open(filename, "r").read()
    print(f)
    tokens = sent_tokenize(f)
    print(tokens)
    for line in tokens:
        file_docs.append(line)
    gen_docs = [[w.lower() for w in word_tokenize(text)]
                for text in file_docs]
    from gensim import corpora
    dictionary = corpora.Dictionary(gen_docs)
    corpus = [dictionary.doc2bow(gen_doc) for gen_doc in gen_docs]
    import numpy as np
    from gensim.models import TfidfModel
    tf_idf = TfidfModel(corpus)
    from gensim.similarities import Similarity
        # building the index
    sims = Similarity(r'C:/Users/User/OneDrive/Desktop/UTeM/Sem5/NLP/Project/Real code',tf_idf[corpus],
                                            num_features=len(dictionary))
```

Figure 1.4: Code Snippet 2

Next, we define the function for calculate the similarity between both files. We split all the sentences from first file using tokenization. After that we split each word from the sentences and calculate the term frequency-inverse document frequency. The number of times a word appears in a document divided by the total number of words in the document. Every document has its own term frequency and the log of the number of documents divided by the number of documents that contain the word.

```
    #File 2
    file2_docs = []
    # file2 =input("Enter the second file:")

    d = open(filename2, "r").read()
    print(d)
    tokens2 = sent_tokenize(d)
    for line in tokens2:
        file2_docs.append(line)
    avg_sims = [] # array of averages

    # for line in query documents
    for line in file2_docs:
        # tokenize words
        query_doc = [w.lower() for w in word_tokenize(line)]
        # create bag of words
        query_doc_bow = dictionary.doc2bow(query_doc)
        # find similarity for each document
        query_doc_tf_idf = tf_idf[query_doc_bow]
        # print (document_number, document_similarity)
        print('Comparing Result:', sims[query_doc_tf_idf])
        # calculate sum of similarities for each query doc
        sum_of_sims =(np.sum(sims[query_doc_tf_idf], dtype=np.float32))
        # calculate average of similarity for each query doc
        avg = sum_of_sims / len(file_docs)
        # print average of similarity for each query doc
        print(f'avg: {sum_of_sims / len(file_docs)}')
        # add average values into array
        avg_sims.append(avg)
    # calculate total average
    total_avg = np.sum(avg_sims, dtype=np.float)
    # round the value and multiply by 100 to format it as percentage
    percentage_of_similarity = round(float(total_avg) * 100)
    # if percentage is greater than 100
    # that means documents are almost same
    if percentage_of_similarity >= 100:
        percentage_of_similarity = 100
    # ========================================================================
    print("Percentage of similarity is ",percentage_of_similarity)
    # ========================================================================
    label_percentage.configure(text="Similarity Percentage: "+ str(percentage_of_similarity))


def printt():
    return(percentage_of_similarity)
```

Figure 1.5: Code Snippet 3

We did the same process for the second file and create the bag of words. After that we calculate the similarity between both files and display the percentage of similarity only on the user interface.

```
# Set window title
window.title('Text Similarity')
lable_0 = Label(window,text="Welcome to Text Similarity", width=20,font=("bold",20))
lable_0.pack()

# Set window size
window.geometry("700x500")

#Set window background color
window.config(background = "light blue")

# Create a File Explorer label
label_file_explorer = Label(window,
                            text = "First Text File",
                            width = 100, height = 4,
                            fg = "blue")


button_explore = Button(window,
                        text = "Browse File",
                        command = browseFiles)

label_file_explorer2 = Label(window,
                            text = "Second  Text File",
                            width = 100, height = 4,
                            fg = "blue")

button_explore2 = Button(window,
                        text = "Browse File",
                        command = browseFiles2)


button_submit = Button(window, text="Submit" , command=similar)

label_percentage = Label(window,
                            text = "Similarity Percentage ",
                            width = 100, height = 4,
                            fg = "blue")


# Grid method is chosen for placing
# the widgets at respective positions
# specifying rows and columns
label_file_explorer.pack()
button_explore.pack()
label_file_explorer2.pack()
button_explore2.pack()
button_submit.pack()
label_percentage.pack()

# Let the window wait for any events
window.mainloop()
```

Figure 1.6: Code Snippet 4

Figure 1.6 shown the code for build user interface where user will see when they use this system.

Strength and Weakness of the System

In every system, there are always strengths and weaknesses, which is the same for this system. We found out there are some strengths and weaknesses after completing the system as shown in table 1.1

Table 1.1: Strength and Weakness of the System

STRENGTH	WEAKNESS
Can check the percentage of similarities of document in a fleeting period.	Cannot compare with many text files
Friendly User Interfaces	Only allow text file
Available and free for all users	Cannot compare with documents in the google like Turnitin
	User must change the path in the coding by their own

Conclusion

In conclusion, this Text Similarity can be used in real life but there are some weaknesses. Thus, in the future we hope to improve the system by applying enhancements to the system by allowing the system to access the links on the website needed. Next, we hope to expand the system to be able to compare with multiple text files, other file formats and even online documents.

Based on our opinion, this system's development is straightforward and does not require a lengthy time to execute. Although there are other ways to create a Text Similarity system, we believe that the method of implementation that we chose is easier to understand and implement. Moreover, our system's design which includes the flow of the system and the GUI is easy for users to use.

Our system can contribute to the education sector especially for lecturers and students alike. Students can check the similarity of their work directly with the references that are involved before turning in their assignments. Lecturers are also able to detect plagiarism using this system. Moreover, this system is free thus it is very appealing for the students as they do not have a steady income to subscribe to sites that charge for their services.

References

Tkinter library, Accessed on 15/1/2021 at https://docs.python.org/3/library/tkinter.h

D. Amos, Python GUI Programming With Tkinter, Accessed on 5/1/21 at https://realpython.com/python-gui-tkinter/

Python - GUI Programming (Tkinter), Accessed on 15/1/21 at https://www.tutorialspoint.com/python/python_gui_programming.htm

ProgrammingKnowledge, Tkinter Python GUI Tutorial For Beginners 1 - Introduction to Tkinter, Accessed on 15/1/21 at https://www.youtube.com/watch?v=-GhzpvvIXlM

S. Gupta, 2018, Overview of Text Similarity Metrics in Python, Accessed on 1/1/21 at https://towardsdatascience.com/overview-of-text-similarity-metrics-3397c4601f50

A.Sieg, 2018, Text Similarities: Estimate the degree of similarity between two texts, Accessed on 1/1/21 at https://medium.com/@adriensieg/text-similarities-da019229c894