VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

VIT Chennai

Vandalur - Kelambakkam Road, Chennai - 600 127

Final Review Report

Programme: B.Tech. (Computer Science and Engineering)

Course: CSE4022 (Natural Language Processing)

Slot: E2 + TE2

Faculty: Dr. Premalatha M

Component: Embedded Project

Title: Mind Map Generator

Team Member(s)

Manigandan R – 20BCE1223

Siddharth Thakur – 20BCE1144

Mohd Saifee – 20BCE1980

Abstract:

We intend to create a Mind Map Generator which summarizes given text to make it simpler and quicker to interpret. A Mind Map is something which can be used to represent an idea in a visual way or in a structured manner where the main idea is at centre acting as root and the relevant idea emerges as nodes from the root node. To develop this model, we will be using the required text mapping libraries that can summarize texts and create branches with relevant subtopics using the given text as the source material. This strategy would assist in saving time while reading extensive paperwork or study materials by just providing a glimpse of the most crucial and valuable information through the usage of a mind map. This would allow us to focus on the information that is most relevant to the topic and needs.
Mind map generators have been mentioned in most research articles published up until this point, however, there is very little actual output that can be accessed. Because of this, our model is distinguished from others in that it enables us to customise both the level of detail represented by the tree and the degree to which its branches branch out. So, our main objective is to create a Mind Map Generator using NLP techniques which includes the usage of LDA topic modelling and RAKE algorithm.

Keywords: Mind Map, Node, Text Summarizer, NLP, LDA Topic modelling, RAKE

Introduction:

Everyone has been in a situation at least once in their lives where they have a really short time span to go through and remember large amounts of information. Most of the time it's the students who would want to revise their study material before an exam or maybe an office-going individual who has his presentation in some time for which he needs to refer to certain research papers or documents and capture a glimpse of them just before delivering his views on it to others.

Our main objective was to help people overcome this situation in an efficient way. We usually go through mind maps and other visual representations to get a list of whatever topics it is made for. Getting inspiration from the same thought, we decided to create something which would rather generate a mind map for whatever text you feed it with. This would be very useful to those people as mentioned above especially when it comes to going through notes while in a time crunch. The only step which needs to be taken is to feed the system with a paragraph or text, and within a few seconds, the text will be converted into a mind map. This Mind Map Generator falls in the level of Compositional Semantics because we focus on the meaning of each sentence given by the user.

Methodology and Implementation:

We have basically made a convenient way for people to convert a big text into a summarized form along with a mind map representation of it for easier review and a better understanding of it. The mind map also includes interconnection between nodes (Keywords/topics) which shows relationships between topics and how they are related to. So, in developing this model we are not directly using any dataset, it is just any kind of data that we can use from entering a text directly, from a file, or by web scrapping.

Since we are getting data directly from a file or through web scrapping, it is very important to pre-process the data. To do that, we first cleaned the data using regular expression as follows:

```python
def clean_text(text):
    space_pattern = r"(  |\r|\n|\t)"
    citation_pattern = r'\[[0-9a-zA-Z]*\]'
    text = re.sub(space_pattern, " ", text)
    text = re.sub(citation_pattern, "", text)

    return text
```

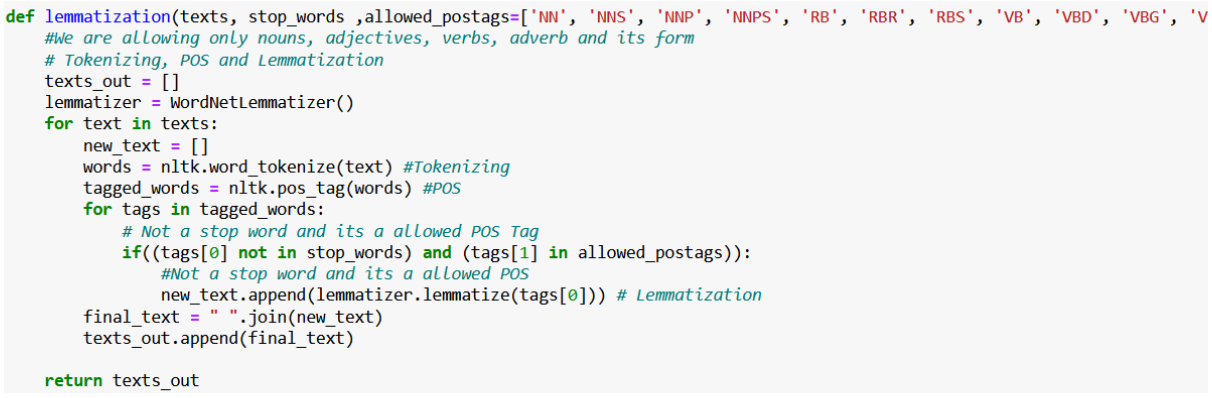
Code 1: Code for cleaning text

We removed all unnecessary spaces and then remove citations. Once the data is cleaned, we will then perform sentence segmentation. Once we segment sentences, we will group sentence into a group of 3 or 4 (which depends on user).

```python
def group_sentences(sentences, group_len = 3):
    #Grouping sentences so that it makes sense - Here we randomly chose 3 as threshold in one group.
    new_sentences = []
    for idx in range(0, len(sentences), group_len):
        new_sent = ''
        i = idx
        while i<len(sentences) and i<idx+3:
            new_sent += sentences[i]
            i += 1
        new_sentences.append(new_sent)
    return new_sentences
```

Code 2: Code for grouping sentences

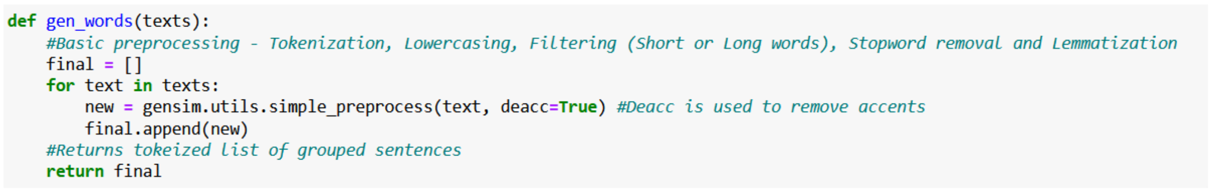
Once grouping is done, we will call a function named lemmatization() which first tokenizes the group of sentences then it tags with POS. After that, we have removed stop words and then we have finally lemmatized using WordNetLemmatizer() to convert the tokens into meaningful root form.



```
def lemmatization(texts, stop_words ,allowed_postags=['NN', 'NNS', 'NNP', 'NNPS', 'RB', 'RBR', 'RBS', 'VB', 'VBD', 'VBG', 'V
    #We are allowing only nouns, adjectives, verbs, adverb and its form
    # Tokenizing, POS and Lemmatization
    texts_out = []
    lemmatizer = WordNetLemmatizer()
    for text in texts:
        new_text = []
        words = nltk.word_tokenize(text) #Tokenizing
        tagged_words = nltk.pos_tag(words) #POS
        for tags in tagged_words:
            # Not a stop word and its a allowed POS Tag
            if((tags[0] not in stop_words) and (tags[1] in allowed_postags)):
                #Not a stop word and its a allowed POS
                new_text.append(lemmatizer.lemmatize(tags[0])) # Lemmatization
        final_text = " ".join(new_text)
        texts_out.append(final_text)

    return texts_out
```

Code 3: Code for Tokenization, POS and Lemmatization

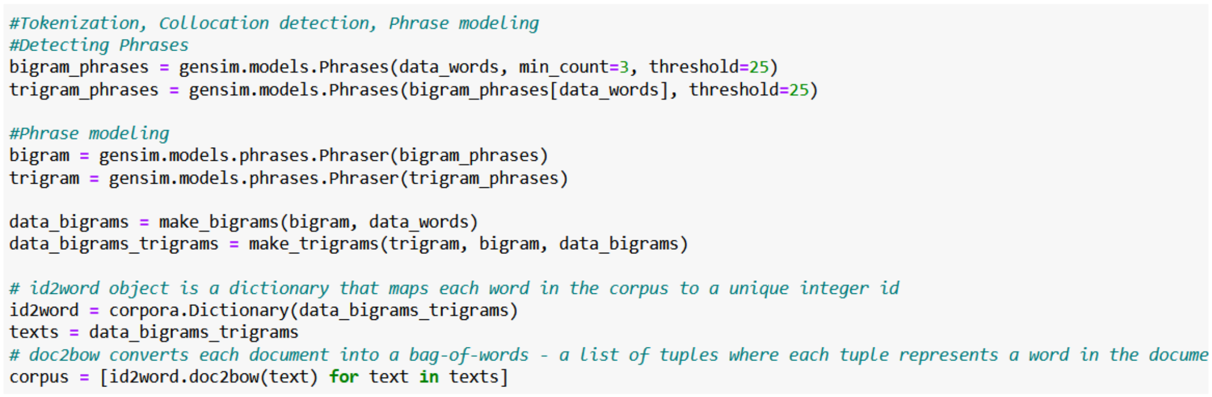
Then we call another function names gen_words() which is used to filter the words based on its length, lower case all the words and remove accents. This function will return tokenized list of grouped sentences.



```
def gen_words(texts):
    #Basic preprocessing - Tokenization, Lowercasing, Filtering (Short or Long words), Stopword removal and Lemmatization
    final = []
    for text in texts:
        new = gensim.utils.simple_preprocess(text, deacc=True) #Deacc is used to remove accents
        final.append(new)
    #Returns tokeized list of grouped sentences
    return final
```

Code 4: Code for Filtering, Lowercasing and Accent removal

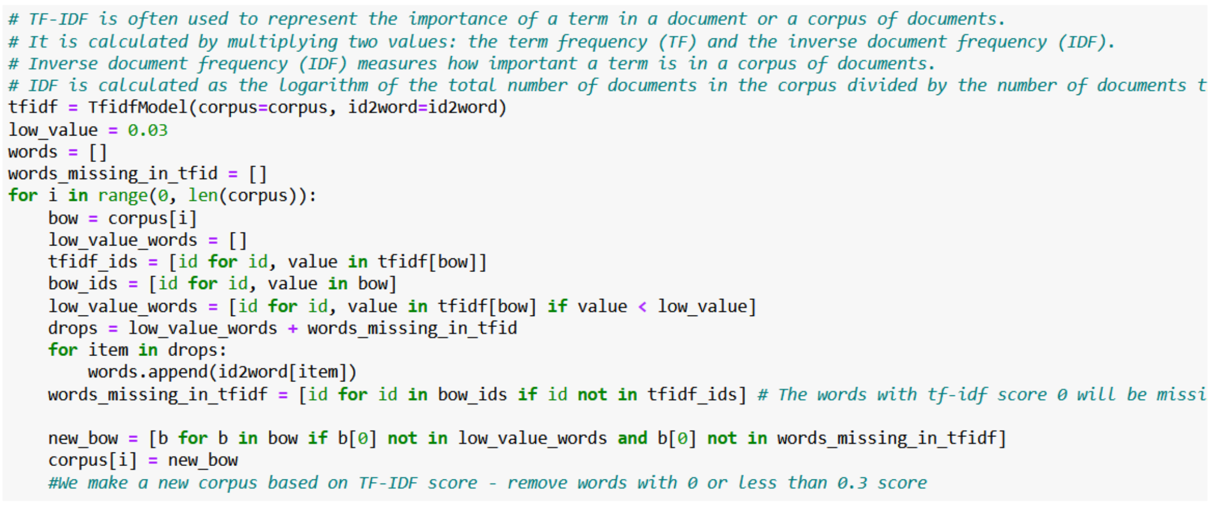
After this step, we find bigrams and trigrams by first detecting phrases. After this, phrase modelling will be done which will return bigrams and trigrams present in the given text. Then we perform id2word which maps each word in the corpus to a unique integer ID which will be later used to create bag of words.



```
#Tokenization, Collocation detection, Phrase modeling
#Detecting Phrases
bigram_phrases = gensim.models.Phrases(data_words, min_count=3, threshold=25)
trigram_phrases = gensim.models.Phrases(bigram_phrases[data_words], threshold=25)

#Phrase modeling
bigram = gensim.models.phrases.Phraser(bigram_phrases)
trigram = gensim.models.phrases.Phraser(trigram_phrases)

data_bigrams = make_bigrams(bigram, data_words)
data_bigrams_trigrams = make_trigrams(trigram, bigram, data_bigrams)

# id2word object is a dictionary that maps each word in the corpus to a unique integer id
id2word = corpora.Dictionary(data_bigrams_trigrams)
texts = data_bigrams_trigrams
# doc2bow converts each document into a bag-of-words - a list of tuples where each tuple represents a word in the docume
corpus = [id2word.doc2bow(text) for text in texts]
```

Code 5: Code for Bigram & Trigram Modelling, ID2Word and Bag of Words

Using the output of id2word, we get TF-IDF which represents the important of a term in a document or a corpus of document. It is calculated by multiplying two values: the term frequency (TF) and the inverse document frequency (IDF). Inverse document frequency (IDF) measures how important a term is in a corpus of documents. IDF is calculated as the logarithm of the total number of documents in the corpus divided by the number of documents that contain the term. This will give us the words which have high significance and low significance. So, we will be removing the words which have significance less than 0.03 so that the words present in the mind map are precise.



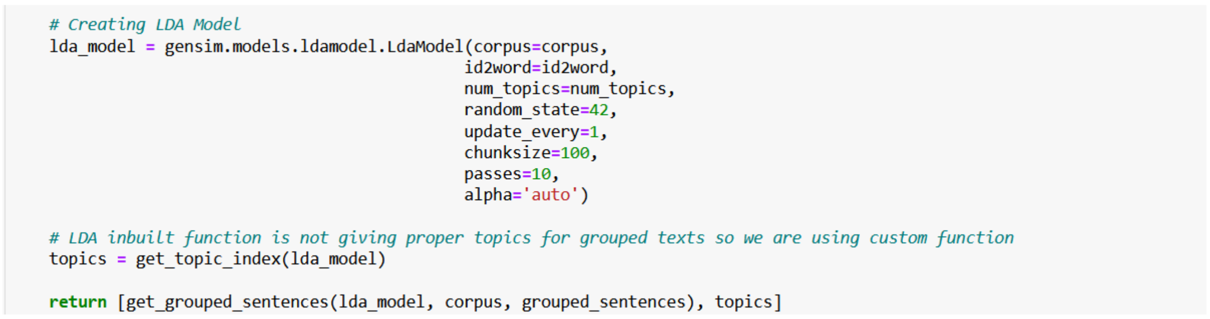
```python
# TF-IDF is often used to represent the importance of a term in a document or a corpus of documents.
# It is calculated by multiplying two values: the term frequency (TF) and the inverse document frequency (IDF).
# Inverse document frequency (IDF) measures how important a term is in a corpus of documents.
# IDF is calculated as the logarithm of the total number of documents in the corpus divided by the number of documents t
tfidf = TfidfModel(corpus=corpus, id2word=id2word)
low_value = 0.03
words = []
words_missing_in_tfid = []
for i in range(0, len(corpus)):
    bow = corpus[i]
    low_value_words = []
    tfidf_ids = [id for id, value in tfidf[bow]]
    bow_ids = [id for id, value in bow]
    low_value_words = [id for id, value in tfidf[bow] if value < low_value]
    drops = low_value_words + words_missing_in_tfid
    for item in drops:
        words.append(id2word[item])
    words_missing_in_tfidf = [id for id in bow_ids if id not in tfidf_ids] # The words with tf-idf score 0 will be missi

    new_bow = [b for b in bow if b[0] not in low_value_words and b[0] not in words_missing_in_tfidf]
    corpus[i] = new_bow
    #We make a new corpus based on TF-IDF score - remove words with 0 or less than 0.3 score
```

Code 6: Code for TF-IDF

The output of pre-processing will be a new corpus which is derived from the original text based on TF-IDF score with the removal of words with 0 or less than 0.3 score. Based on these words, we apply LDA and RAKE algorithm to extract keywords from this corpus.

Our Mind Map Generator falls in the level of Compositional Semantics because we focus on the meaning of each sentence given by the user and for this to work, we have used two major algorithms:

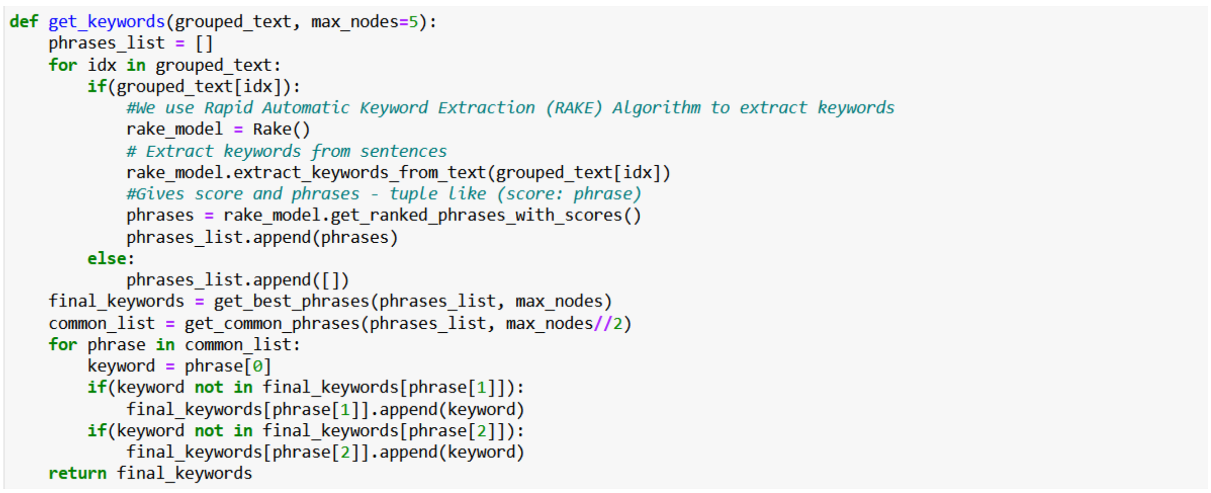
LDA Topic modelling: Latent Dirichlet Allocation (LDA) is an unsupervised modelling algorithm which is widely used to extract topics and from a text. It also clusters the relevant sentences belonging to the topic. So, we will be using the clusters it gave us as output and map each cluster to the most probable word of that cluster as the topic. This is how we will be deriving topics/the first level of the mind map from the text.



```python
    # Creating LDA Model
    lda_model = gensim.models.ldamodel.LdaModel(corpus=corpus,
                                                id2word=id2word,
                                                num_topics=num_topics,
                                                random_state=42,
                                                update_every=1,
                                                chunksize=100,
                                                passes=10,
                                                alpha='auto')

    # LDA inbuilt function is not giving proper topics for grouped texts so we are using custom function
    topics = get_topic_index(lda_model)

    return [get_grouped_sentences(lda_model, corpus, grouped_sentences), topics]
```

Code 7: Code for LDA

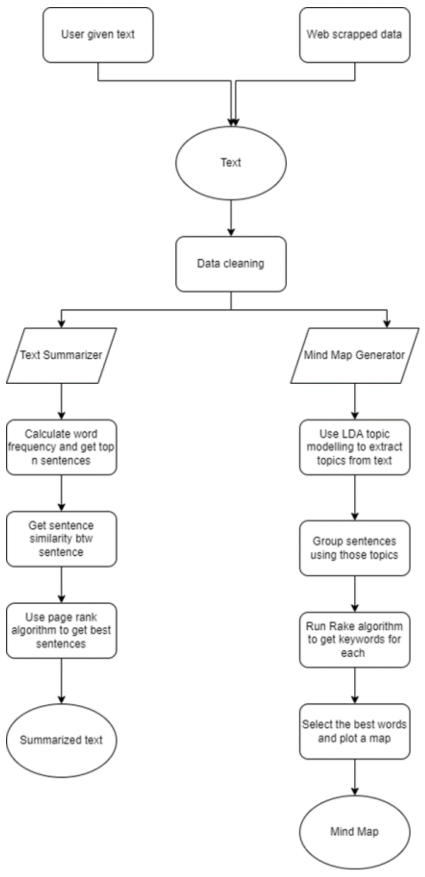
RAKE: Rapid Automatic Keyword Extraction (RAKE) is one of the widely used keyword extraction algorithms. Now to extract keywords what we have done is to first group all the sentences which belong to one topic and hence form grouped sentences. Now we will pass these grouped sentences to the rake algorithms to get keywords for each sentence. These will be the second level of the mind map.



```python
def get_keywords(grouped_text, max_nodes=5):
    phrases_list = []
    for idx in grouped_text:
        if(grouped_text[idx]):
            #We use Rapid Automatic Keyword Extraction (RAKE) Algorithm to extract keywords
            rake_model = Rake()
            # Extract keywords from sentences
            rake_model.extract_keywords_from_text(grouped_text[idx])
            #Gives score and phrases - tuple like (score: phrase)
            phrases = rake_model.get_ranked_phrases_with_scores()
            phrases_list.append(phrases)
        else:
            phrases_list.append([])
    final_keywords = get_best_phrases(phrases_list, max_nodes)
    common_list = get_common_phrases(phrases_list, max_nodes//2)
    for phrase in common_list:
        keyword = phrase[0]
        if(keyword not in final_keywords[phrase[1]]):
            final_keywords[phrase[1]].append(keyword)
        if(keyword not in final_keywords[phrase[2]]):
            final_keywords[phrase[2]].append(keyword)
    return final_keywords
```

Code 8: Code for RAKE

The methodology is summarized in the following flow chart:



User given text
Web scrapped data
Text
Data cleaning
Text Summarizer
Mind Map Generator
Calculate word frequency and get top n sentences
Use LDA topic modelling to extract topics from text
Get sentence similarity btw sentence
Group sentences using those topics
Use page rank algorithm to get best sentences
Run Rake algorithm to get keywords for each
Summarized text
Select the best words and plot a map
Mind Map

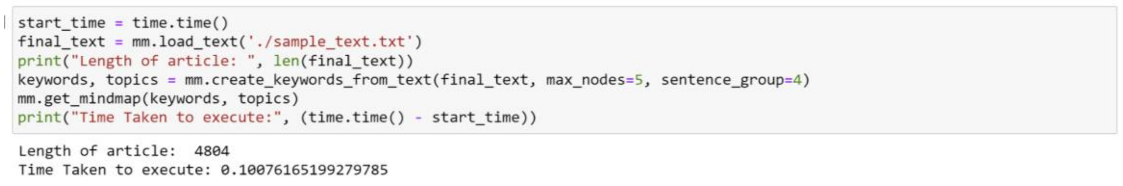
Figure 4: Flow chart of Mind Map Generation

Initially, the user will give the text as input. The model will take this text and it will be cleaning the input provided by the user. We will be then using Text Rank Algorithm to summarize the given text. Once we summarize the text, the model will be running LDA topic modelling algorithm to extract the topic and relevant sentences to the topic from the summarized text. Based on the extracted topics, we will be using RAKE algorithm to get keywords relevant keywords to each topic from the sentences. Now after getting the required words from these, what we will be doing is to take only the top 75th percentile of the words based on their probability score of the rake algorithm which gave us full keywords. After this we added a similarity check between the keywords using cosine similarity so that similarly keywords are removed from the list. Finally, based on these keywords, we will be generating the mind map. This is how the project works.

Results and Discussion:

The final output will be a mind map of the text which can be inputted through a file or by giving a URL. We evaluated the Mind Map based on the time of execution. We evaluated mind map generated using text from file and text from web scrapping separately because network latency comes into picture when we are getting text through web scraping.

Performance Analysis for text files:



```
start_time = time.time()
final_text = mm.load_text('./sample_text.txt')
print("Length of article: ", len(final_text))
keywords, topics = mm.create_keywords_from_text(final_text, max_nodes=5, sentence_group=4)
mm.get_mindmap(keywords, topics)
print("Time Taken to execute:", (time.time() - start_time))
```

```
Length of article:  4804
Time Taken to execute: 0.10076165199279785
```

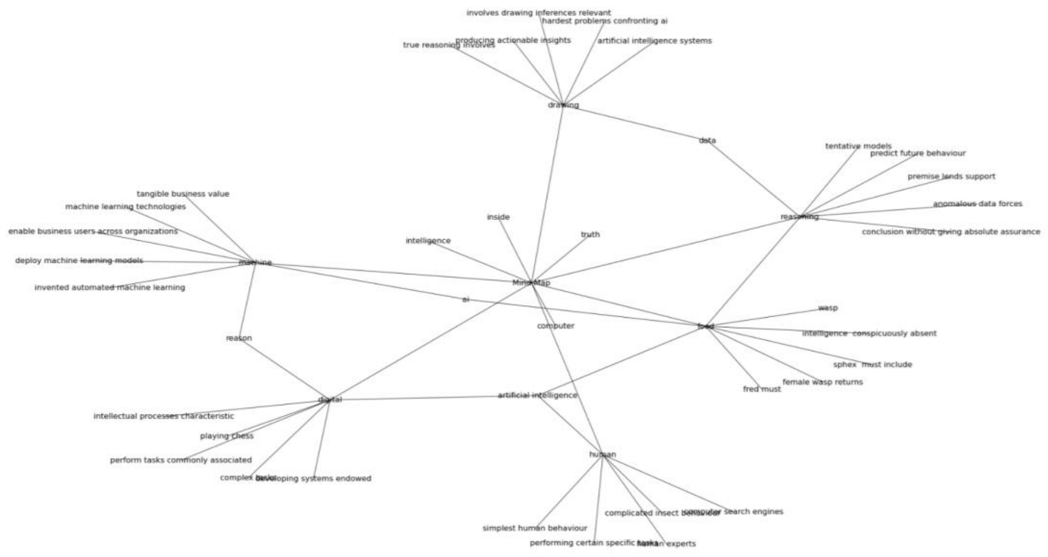

involves drawing inferences relevant
hardest problems confronting ai
true reasoning involves
producing actionable insights
artificial intelligence systems
drawing
data
tentative models
predict future behaviour
premise lends support
anomalous data forces
conclusion without giving absolute assurance
reasoning
tangible business value
machine learning technologies
enable business users across organizations
deploy machine learning models
invented automated machine learning
machine
inside
intelligence
truth
MindMap
ai
computer
food
wasp
intelligence conspicuously absent
sphex must include
female wasp returns
fred must
reason
digital
artificial intelligence
intellectual processes characteristic
playing chess
perform tasks commonly associated
complex tasks
developing systems endowed
human
complicated insect behaviour
computer search engines
simplest human behaviour
performing certain specific tasks
human experts

Figure 5: Mind Map Generated using text from a file

S No.	Article Length	Time of Execution
1	4808	0.1007
2	5063	0.1147
3	17157	0.3101

Table 1: Article Length and Time of Execution for Generated Mind Map using text from a file

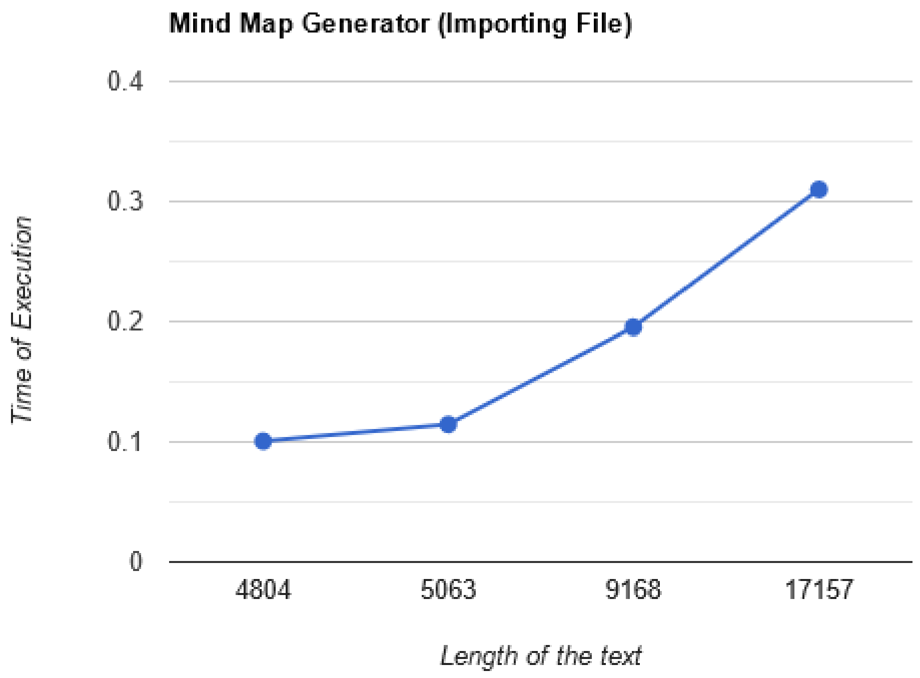
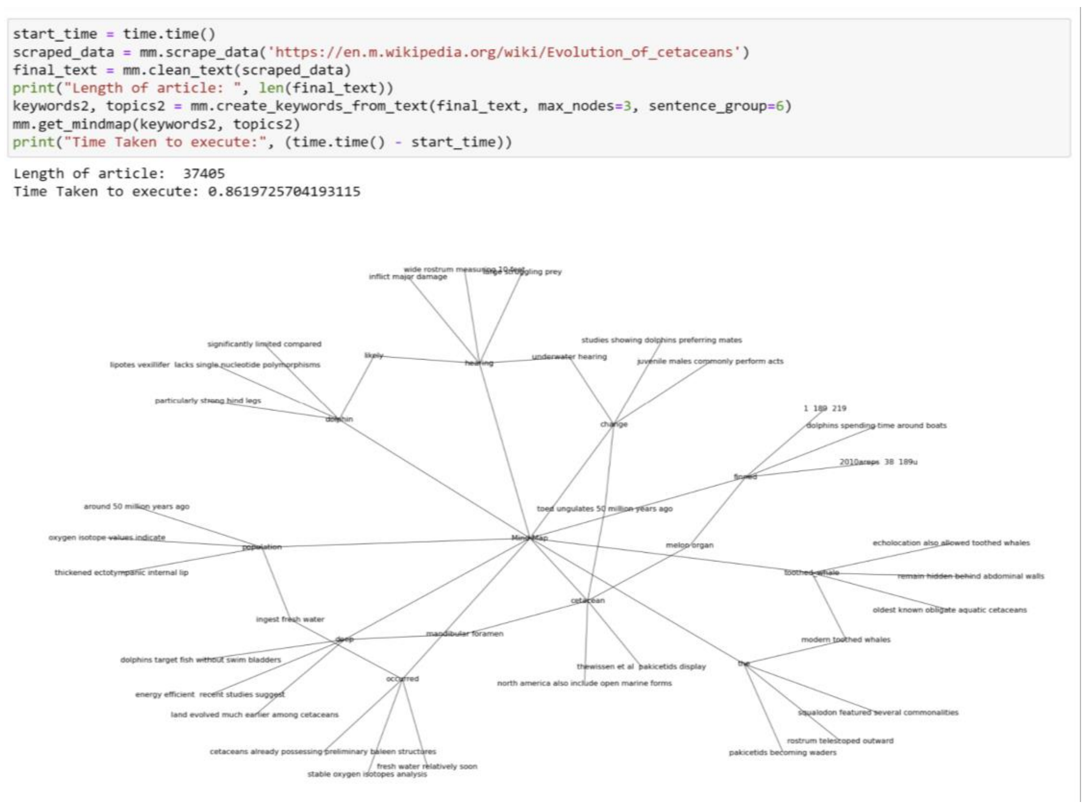


Figure 6: Graph for Article Length Vs Time of Execution for Generated Mind Map using text from a file

From the tabulation and the graph, we can see that the time taken for generating the mind map using the text from a file depends on the length of the text. This depends can be expressed in terms of direct proportionality. This means that if the length of the text increase, the time taken to generate the mind map also increases linearly.

Performance analysis for web scrapped text:



```
start_time = time.time()
scraped_data = mm.scrape_data('https://en.m.wikipedia.org/wiki/Evolution_of_cetaceans')
final_text = mm.clean_text(scraped_data)
print("Length of article: ", len(final_text))
keywords2, topics2 = mm.create_keywords_from_text(final_text, max_nodes=3, sentence_group=6)
mm.get_mindmap(keywords2, topics2)
print("Time Taken to execute:", (time.time() - start_time))
```

```
Length of article:  37405
Time Taken to execute: 0.8619725704193115
```

wide rostrum measuring 10 feet
inflict major damage
large struggling prey
significantly limited compared
likely
hearing
underwater hearing
studies showing dolphins preferring mates
juvenile males commonly perform acts
lipotes vexillifer lacks single nucleotide polymorphisms
particularly strong hind legs
dolphin
change
1 189 219
dolphins spending time around boats
2010areps 38 189u
flowed
around 50 million years ago
toed ungulates 50 million years ago
oxygen isotope values indicate
population
MindMap
melon organ
echolocation also allowed toothed whales
thickened ectotympanic internal lip
toothed whale
remain hidden behind abdominal walls
cetacean
oldest known obligate aquatic cetaceans
ingest fresh water
deep
mandibular foramen
modern toothed whales
dolphins target fish without swim bladders
thewissen et al pakicetids display
the
occurred
north america also include open marine forms
energy efficient recent studies suggest
squalodon featured several commonalities
land evolved much earlier among cetaceans
rostrum telescoped outward
cetaceans already possessing preliminary baleen structures
pakicetids becoming waders
fresh water relatively soon
stable oxygen isotopes analysis

Figure 7: Graph for Article Length Vs Time of Execution for Generated Mind Map using text from a file.

S No.	Article Length	Time of Execution
1	13940	0.4991
2	19971	0.7326
3	37405	0.8619
4	50469	1.4081
5	70912	2.3042

Table 2: Article Length and Time of Execution for Generated Mind Map using text through web scrapping

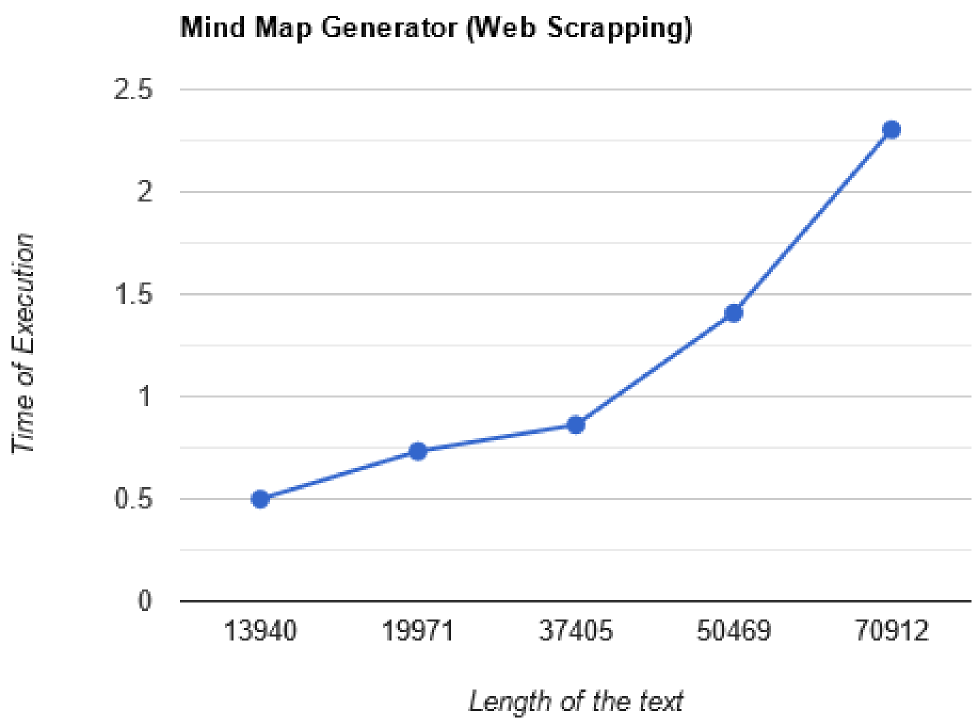


Figure 8: Graph for Article Length Vs Time of Execution for Generated Mind Map using text through web scrapping.

From the tabulation and the graph, we can see that the time taken for generating the mind map using the text through scrapping web and again this depends on can be expressed in terms of direct proportionality. This means that if the length of the text increase, the time taken to generate the mind map also increases linearly. The time of execution is more when compared to the previous one where mind map is generated through the text obtained through importing file is because of network latency.

Conclusion:

We have successfully created a summarized mind map by using our mapping libraries that can summarize texts and create branches connecting relevant subtopics and keywords. The output obtained is a summarized mind map where we can also successfully set the depth of the map and a few other parameters like max number of nodes to get a more customized and accurate mind map based on our need. The generation of mind map linearly depends on the length of the text given as input. The development such a tool for creating mind maps will greatly benefit individuals and teams in organizing their ideas and information in a visually appealing and intuitive way. The tool utilizes NLP algorithms and pre-processing techniques to generate mind maps that are customizable, easy to visualize and understand.

Future Work:

We have used RAKE algorithm to extract keywords. One area of future work would be usage of Key Bert to extract keywords instead of RAKE algorithm. Key Bert is a language model developed by Google, which is designed to improve the quality of natural language processing tasks such as text classification, named entity recognition, and question answering can be used to extract keywords more efficiently.