香港中文大學（深圳）

The Chinese University of Hong Kong, Shenzhen

School of Data Science

ERG3020: Web Analytics and Intelligence

Project Report

Abstract Extraction Based on TextRank, TextRank with GloVe, and MMR

Authors:

Cheng Yang 120090195

Dai Hui 119010051

Wang Muxi 119020052

May 4, 2022

Abstract

With the explosive growth of data in the digital space, most of them are unstructured text data, so it is necessary to develop automatic text summarization tools, so that people can easily get the general idea of a text. Nowadays, we have fast access to a lot of information. But most messages are long, irrelevant and may not convey their intended meaning. So, it is important to use automatic text summarization generators that can extract useful information and weed out irrelevant and useless data. The implementation of text summarization can enhance the readability of documents, reduce the time to search for information, and obtain more information suitable for a specific field. In this report, we use three methods: TextRank, MMR and GloVe based on TextRank to extract abstracts of different articles themed on COVID-19 and make a comparison. The result shows that MMR has a better performance compared to TextRank method, and GloVe model is more effective in extracting abstract from a group of texts.

1. Introduction

The aim of text summarization is to compress, conclude and summarize long texts so as to form short texts with general meanings. According to the different number of documents, text summarization tasks can be divided into single document summarization and multi-document summarization. According to the different summarization methods, text summarization task can be divided into extractive method and abstractive method. Extractive method is widely used in the industry due to its early development and relatively mature technology, which is based on a certain weight, from the original text to find the closest to the central idea of one or several sentences. In this paper, three methods of extractive summary (TextRank, MMR, and GloVe based TextRank) are put into practice and a comparative conclusion is obtained.

2. Data

2.2 Data Description

In order to get better summary results and be able to compare them between different methods, we need a data set with sufficient data, which belongs to the same theme. In addition, each set of data should contain a professional article and an original abstract. Here we have selected a dataset of a collection of articles from Kaggle[1], which contains 13202 papers about the research of Covid-19 including paper id, source, title, abstract, and text body.

[1] Data is available at https://www.kaggle.com/code/d4v1d3/cord-19-lda-topic-modeling-reccomendation-system/data?select=kaggle_covid-19_open_csv_format.csvt

We choose the top 100 text bodies for text summarization, and then compare them with the original abstracts to test the effectiveness of summaries.

	doc_id	source	title	abstract	text_body
0	dd6c1a719bd75ba7fb7c62291a676b39c99bb0bc	BIORXIV	Veterinary Science Molecular characterization ...	The cDNA nucleotide sequence of genome segment...	Members of the family Birnaviridae have 2-segm...
1	d46d2f01a570bfb6e3b2e65ec50972d1649b6a3d	BIORXIV	Severe Acute Respiratory Syndrome: Lessons fro...	An outbreak of severe acute respiratory syndro...	A n outbreak of severe acute respiratory syndr...
2	99c0958f9fa04d32dcdd13fcc01ffcb531f48950	BIORXIV	Plagues, Public Health, and Politics 1	lord mayor and aldermen of the city of London ...	Science influences public health decisions and...
3	c1a03657b225afb8628f4806530db7f57645ad54	BIORXIV	CPT-cGMP Is A New Ligand of Epithelial Sodium ...	Epithelial sodium channels (ENaC) are localize...	Cyclic guanosine monophosphates (cGMP) are imp...
4	8c593d726094e8a047376050bd28806e371a2797	BIORXIV	Access to	Proteomics is the large-scale study of the str...	The term proteomics describes the study and ch...
...	...	...	...	...	...

2.3 Data preprocessing

We design three different methods to handle the words in a sentence: all words, stemmed, stemmed-stopwords.

2.3.1 All Words Processing

All words processing means we maintain all the words in a sentence no matter whether they are necessary. We split the sentence by a space and then we get a list storing all the words.

2.3.2 Stemmed Processing

Stemmed processing means we transform the words in different tenses into the original form or the simplified form. For example, ‘includes’ and ‘included’ will be recognized as the same and transformed into ‘include’ for simplicity. We import PorterStemmer from nltk.stem to implement this procedure.

2.3.3 Stemmed and Stopwords

Based on the stemmed processing, we download the English stopwords to delete some unnecessary words that have little influence on sentence meaning, such as ‘this’ and ‘a’. Such processing can better extract the meaning of a sentence and only store meaningful words. To implement this, we only need to traverse all the words in the sentence and store which is not in stopwords, then use PorterStemmer to get stemmed words.

3. The TextRank Model

3.1 Basic Principle

3.1.1 PageRank: Underlying Algorithm for TextRank

TextRank (Mihalcea & Tarau, 2004) is a graph-based ranking algorithm for natural language texts based on Google's PageRank algorithm (Brin & Page, 1998). In the PageRank algorithm, each web page is a vertex in the graph, and a link (an edge) is added between two vertices if one page "recommends" the other page. The basic idea behind the PageRank algorithm is that a web page is important if it is linked by many important web pages.

Let $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ be an unweighted directed graph with vertices V and edges E. The score of a vertex V_i can be calculated by (Brin & Page, 1998):

$$S(V_i) = (1-d) + d * \sum_{j \in In(V_i)} \frac{1}{|Out(V_j)|} S(V_j)$$

$\boldsymbol{S(V_i)}$**:** the score of the vertex V_i.
$\boldsymbol{d}$**:** damping factor, which is usually set to be 0.85.
$1-d$ denotes the probability that a user gets bored and jumps to a new page.
$\boldsymbol{In(V_i)}$**:** the set of V_i's back-links
$\boldsymbol{Out(V_j)}$**:** the set of V_j's forward-links

The initial score of each vertex can be set as any value. After running the algorithm iteratively, the scores will converge. Ranking the scores is equivalent to ranking the importance of the web pages. Since PageRank is unsupervised and easy to implement, it is well applied in search engines to rank the web pages and give users the most relevant search results.

3.1.2 TextRank: Based on Weighted Graphs

Borrowing the idea from PageRank, TextRank modified the algorithm by adding weights to the edges. In the original PageRank algorithm, there is no weight on each edge since the strength of each link is the same. However, when ranking the texts, the weight for each edge could be different. Higher w_{ij} means that V_i and V_j are more similar to each other.

Therefore, the score of a vertex V_i of a weighted graph can be calculated by (Mihalcea & Tarau, 2004):

$$WS(V_i) = (1-d) + d * \sum_{j \in In(V_i)} \frac{w_{ji}}{\sum_{V_k \in Out(j)} w_{jk}} WS(V_j)$$

3.2 Application in Abstract Extraction

For abstract extraction, the goal is to pick the most important sentences to generate the abstract. Here, "most important sentences" are the most relevant sentences to the text. To build the graph using the text, each sentence is added as a vertex, and the weight of the link between two sentences is the similarity of these two sentences. Therefore, a weight matrix W is needed to represent the similarity between every two sentences, with

$$w_{ij} = Similarity(S_i, S_j) = \frac{|\{w_k | w_k \in S_i \ \& \ w_k \in S_j \}|}{\log(|S_i|) + \log(|S_i|)}$$

$\boldsymbol{S_i}$**:** i-th sentence in the text, $S_i = w_1^i, w_2^i, \dots, w_{N_i}^i$

$\boldsymbol{w_{ij}}$**:** similarity/weight of sentence i and sentence j

$\{\boldsymbol{w_k} | \boldsymbol{w_k} \in \boldsymbol{S_i}$ & $\boldsymbol{w_k} \in \boldsymbol{S_j}\}$**:** words that are both in sentence i and sentence j

Using these weights, we can iteratively compute the scores of vertices using the formula in Section 3.1.2, and get the converged score for each vertex. Then, we can extract the top-ranked sentences as our automatically generated abstract.

The main steps of TextRank can be concluded as follows:

1) Add sentences as vertices V in the graph.
2) Compute the weight matrix W to represent the weights of the edges, where the weight w_{ij} is computed by $Similarity(S_i, S_j)$ in Section 3.2.
3) Iteratively compute the scores in Section 3.1.2 until convergence.
4) Rank the sentences by their scores, and select the top *n* sentences as the abstract generated by the algorithm.

An example of an application will be shown in section 6.

3.3 Code Implementation

In this section, several critical code implementations will be shown to further explain the algorithm, and the complete code can be reached in Appendix.

The code is shown following the four main steps introduced in Section 3.2:

1) To add sentences as vertices V in the graph, we decompose the text into sentences using **the sent_tokenize()** function from the **nltk** package, a Python toolkit used for Natural Language Processing.

```
# get the list of sentences of the text
sentences = np.array(nltk.sent_tokenize(text), dtype=object)
```

2) To compute the weights, we first decompose each sentence into words using **word_tokenize()** function from the **nltk** package. Furthermore, we can decide whether to do stemming or remove stopping words in this step.

   Secondly, **calculate_similarity(s1,s2)** function is defined to get the weight of sentence 1 and sentence 2. In the case that after stemming and removing the stopping words both two sentences only contain one word, the denominator becomes zero. Therefore, we need to define that:

$$\text{If } |S_i| = |S_j| = 1, w_{ij} = \begin{cases} 0, & \text{if } \left|\{w_k \ / w_k \in S_i \ \& \ w_k \in S_j\}\right| = 0 \\ 1, & \text{if } \left|\{w_k \ / w_k \in S_i \ \& \ w_k \in S_j\}\right| = 1 \end{cases}$$

```
# calculate the weight/similarity of Sentence 1 & Sentence 2
def calculate_similarity(s1,s2):
    num_of_same_word = 0
    for word in s1:
        if word in s2:
            num_of_same_word += 1
    if num_of_same_word == 0:
        return 0
    # to avoid zero denominator
    elif len(s1)==1 and len(s2)==1 and num_of_same_word == 1:
        return 1
    else:
        return num_of_same_word / (math.log(len(s1)) + math.log(len(s2)))
```

   Thirdly, we need a **generate_weights(text)** function to get the weight matrix W.

```
def generate_weights(text):
    n = len(text)
    # original weighted graph: nxn 0 matrix
    weights = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            # avoid empty sentences
            if i!= j and len(text[i])>0 and len(text[j])>0:
                weights[i][j] = calculate_similarity(text[i] , text[j])
    return weights
```

3) The first step is to define a **calculate_score(weights, scores, i)** function to compute the score of vertex i given the weight matrix (**weights**) and the scores from the last iteration (**scores**).

Then the threshold of convergence needs to be set. Here we define that the algorithm is converged if

$$|WS_{new}(V_i) - WS_{old}(V_i)| < 10^{-4} \qquad \forall V_i \ .$$

converge(scores, old_scores) function will return a bool value to tell us whether the algorithm is converged.

Finally, we can iteratively compute the scores until convergence. Note that the initial values do not matter, so we set the initial score of each vertex to be one.

```
def textrank(weights):
    n=len(weights)
    # initial value does not matter
    # scores = [1 for _ in range(n)]
    scores = np.ones(n)
    old_scores = np.zeros(n)

    # start iteration
    while not converge(scores, old_scores):
        old_scores = scores.copy()
        scores = [calculate_score(weights, scores, i) for i in range(n)]

    return scores
```

4) After having the final scores of the sentences, we use **nlargest()** function from **heapq** package to get the indices of the most relevant *n* sentences in the text. By combining these sentences, the abstract is generated.

```
sent_index.sort()
str = ' '
summary =  str.join([sentences[i] for i in sent_index])
```

4. The TextRank Model with GloVe

4.1 Basic Principle

The automatic text summarization algorithm based on TextRank and GloVe word vectors constructs a probability transfer matrix representing semantic similarity of sentences by embedding word vectors into sentences, and gradually adjusts the weight factor of sentences through several iterations.

4.1.1 TextRank

TextRank algorithm is the same as the algorithm in TextRank model.

4.1.2 Obtained Word Vectors from GloVe Model

GloVe (Xu & Chen, 2019) is a word representation tool based on global word frequency statistics. It reveals the semantic information of words by modeling the context relation of words, and expresses the semantic information of each word as a vector composed of real values. The idea is that words with similar meanings often appear in similar contexts. Expressing words as vectors has several benefits. First of all, the cosine similarity of two word vectors can effectively measure the semantic similarity of corresponding words. For example, Table 1 shows the word vector and vector cosine similarity closest to the keyword 'news'.

Word	'news'	'media'	'report'	'channel'	'journalist'	'newspaper'	'radio'	'program'	'interview'
Similarity	1.00	0.78	0.70	0.67	0.66	0.65	0.62	0.62	0.62

Table 1 The word closest to 'news' and its similarity

For example, the word vector equation: airport - plane + train = train station reveals an analogy between two sets of words. Therefore, after the word vector is trained, we can linearly add the vectors of words between sentences as the semantic vector of sentences, and form a more accurate inter-sentence similarity index by calculating the similarity between semantic vectors.

Obtaining word vectors from a large number of texts using the GloVe model generally follows two steps.

4.1.2.1 Constructing Co-occurrence Matrix Based on Words

Glove model first needs to construct a co-occurrence matrix X between words according to a large amount of document information in the corpus, and each element X_{ij} in the matrix represents the number of common occurrences of words i and j in a sliding window. In order to further reduce the weight of the total number occupied by the two distant words, the elements in the co-occurrence matrix can be adjusted by using the attenuation weight function. For example, the attenuation coefficient between two words with a distance of d in the window can be set to 1/d.

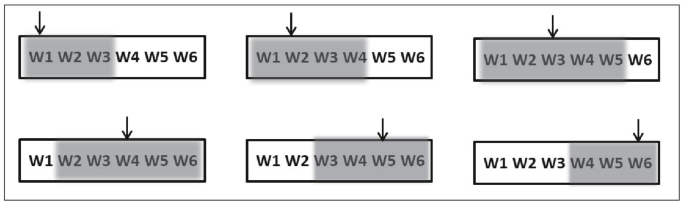


Figure 1 Process of a sliding window

Figure 1 shows the process of counting six-word sentences with a sliding window of width 5. The words indicated by the arrow are the center words of the window. Each window will count the words with a length of less than 2 around the center words and build a co-occurrence matrix according to the statistical information. For example, if the center word is w_2 and the context word is w_1, w_3, w_4, execute:

$$X_{w_2,w_1} += 1$$

$$X_{w_2,w_3} += 1$$

$$X_{w_2,w_4} += 0.5$$

When the window completes the traversal of the whole corpus, the co-occurrence matrix can be obtained.

4.1.2.2 Training Word Vector

GloVe model trains distributed vector representation of each word in the text by gradient descent, and its loss function is as follows:

$$J = \sum_{i,j}^{N} f(X_{i,j})(v_i^T v_j + b_i + b_j - \log(X_{i,j}))^2$$

Where v_i and v_j are the word vectors to be solved through training, b_i and b_i are the deviation terms of the two word vectors, and N is the size of the entire vocabulary. $f(X_{i,j})$ is a stage function, which is used to reduce the interference of high-frequency words to the model. It is generally set as:

$$f(x) = \begin{cases} (\frac{x}{x_{max}})^{0.75}, & x < x_{max} \\ 1, & x \geq x_{max} \end{cases}$$

4.2 Application in Abstract Extraction

4.2.1 Iterative Calculation

The iterative calculation process involves two aspects.

On the one hand, the semantic vector of each sentence in the text is calculated by the words contained in the sentence, the weight of the words, and the word vector trained by the GloVe model. At the same time, cosine similarity between sentence vectors is used to redefine the similarity between sentences, that is, the strength of the connection between different nodes on the graph model, so the TextRank algorithm can be used to recalculate the weight of each sentence.

On the other hand, the co-occurrence relationship between sentences and words is used to re-adjust the weights of sentences and words in each iteration. The idea is that when a word is frequently used in important sentences, the weight of the word itself increases, and conversely, if a sentence contains many important words, the weight of the sentence increases accordingly.

Mathematical symbols are defined as follows: Assuming that the text contains m sentences and n words, we use $S = [S_1, S_2, ..., S_m]$ and $T = [T_1, T_2, ..., T_n]$ to represent the weight of sentences and words respectively. $U = [u_1, u_2, ..., u_m]$ and $V = [v_1, v_2, ..., v_n]$ respectively represent the semantic vector of the synthesized sentence and the word vector learned through GloVe algorithm, C_{ij} represents the number of occurrences of word j in sentence i.

The model is initialized first. We set that each sentence has an average initialization weight, and the initialization weight of each word can be calculated by TF-IDF. The semantic vector of the sentence is 0 vector initially (same dimension and word vector).

In the following iteration process, the algorithm will update the semantic vector u_i of each sentence in the text, the weight S_i of each sentence, and the weight T_j of each word. The four steps in iteration are: calculate the semantic vector of sentences, calculate the weight of sentences according to the graph model, adjust the weight of words, and adjust the weight of sentences.

4.2.2 Calculate the Semantic Vector of Sentences

The first step is to recalculate the semantic vector of each sentence. In this step, all the word vectors contained in the sentence are weighted and summed up to form a vector representation that is highly generalized for the overall semantics of the sentence. The calculation formula is as follows:

$$u_i = \sum_{j \in D_i} T_j v_j$$

Where D_i represents the set of words contained in the i-th sentence, and the semantic vector u_i of the sentence is equal to the weighted sum of each word vector v_j contained in the sentence.

4.2.3 Update the Weight of Sentence According to Graph Model

The second step is to set each sentence as a vertex and the similarity relation between sentences as the edge that connects vertices, then input into graph model. The TextRank

algorithm is used to recalculate the weight of each vertex. The calculation formula is as follows:

$$S_i = (1-d) + d \times \sum_{V_j \in In(V_i)} S_j \times \frac{w_{ij}}{\sum_{V_k \in Out(V_j)} w_{jk}}$$

$$w_{ij} = \frac{u_i \cdot u_j}{\|u_i\| \cdot \|u_j\|}$$

The main difference with TextRank algorithm is that it redefines the similarity between two sentences.

Assume that the semantic vector u_i of the i-th sentence and the semantic vector u_j of the j-th sentence in the text have been obtained, then the similarity of the two sentences is the cosine similarity of the two vectors.

This value ranges from 0 to 1, with a larger value indicating that the content of the two sentences is more similar. After the similarity of each pair of sentences is calculated, the probability transfer matrix of the sentence graph can be obtained:

$$W = \begin{pmatrix} w_{11} & \cdots & w_{1m} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mm} \end{pmatrix},$$

where the corresponding value of row i and column j is the weight of the connecting edge in the word graph, and also represents the probability value of sentence i transferring the weight to sentence j.

4.2.4 Adjust Word Weights

The third step is to adjust the weight of each word according to the weight of sentences in the text and the co-occurrence relationship between sentences and words. If a word is frequently associated with important sentences, the weight of that word will increase in this iteration. The calculation formula is as follows:

$$\Delta T_j = \frac{\sum_{i=1}^{m} (C_{ij} \times S_i)}{\sum_{i=1}^{m} C_{ij}}$$

$$T_j = \alpha T_j + (1-\alpha)\Delta T_j$$

In the formula, α is an adjustment parameter with a value between 0 and 1. The smaller α is, the greater the influence of the calculation results of this round on word weight is.

4.2.5 Adjust Sentence Weights

Contrary to the third step, the fourth step uses the weight of words in the text and the co-occurrence relationship of sentence and words to adjust the weight of each sentence. The adjustment parameter β in the formula is used to control the degree of change in sentence weight in each iteration.

$$\Delta S_i = \frac{\sum_{j=1}^{n}(C_{ij} \times T_j)}{\sum_{j=1}^{n} C_{ij}}$$

$$S_i = \beta S_i + (1-\beta)\Delta S_i$$

After several iterations, we can get the vector u_i of each sentence, the weight S_i of each sentence and the weight T_j of each word. Through further analysis of these data, we can generate the final text summary.

4.3 Code Implementation

We first obtain the vectors of all the constituent words of each sentence (obtained from the GloVe word vector file, with each vector size of 100 elements), then take the average value of these vectors and obtain the combination vector of this sentence as the feature vector of this sentence.

```
# Eigenvectors of sentences
sentence_vectors = []
for i in clean_sentences:
    if len(i) != 0:
        v = sum([word_embeddings.get(w, np.zeros((100,))) for w in i.split()])/(len(i.split())+0.001)
    else:
        v = np.zeros((100,))
    sentence_vectors.append(v)
```

Define an n-by-n zero matrix and fill it with cosine similarity between sentences, where n is the total number of sentences.

```
# Similarity matrix
sim_mat = np.zeros([len(sentences), len(sentences)])
# Use cosine similarity to initialize the similarity matrix.
from sklearn.metrics.pairwise import cosine_similarity
for i in range(len(sentences)):
    for j in range(len(sentences)):
        if i != j:
            sim_mat[i][j] = cosine_similarity(sentence_vectors[i].reshape(1,100),
            sentence_vectors[j].reshape(1,100))[0,0]
```

The similarity matrix sim_mat is transformed into a graph structure. The nodes of this graph are sentences, and edges are represented by similarity scores between sentences. On this graph, we will apply the PageRank algorithm to get the sentence rankings.

```
# Apply PageRank Algorithm
import networkx as nx
nx_graph = nx.from_numpy_array(sim_mat)
scores = nx.pagerank(nx_graph)
```

5. The MMR (Maximal Marginal Relevance) Model

5.1 Basic Principle

MMR (Carbonell & Goldstein, 1998) is an algorithm which is used to rank the searched documents according to their similarity with the query statement. This ranking algorithm keeps the diversity of search output while chasing for high relevance.

$$MMR(Q, C, R) = \text{argmax}\,^{k}_{d_i\ in\ C}[\lambda \text{sim}(Q, d_i) - (1-\lambda)\max_{d_j\ in\ R}(\text{sim}(d_i, d_j))]$$

Q: Query Statement
C: The set of all searched documents.
R: A initial set based on calculated relevance.
d: Documents in the collection C.
λ: A parameter to adjust the preference between similarity and diversity.
k: The number of documents the algorithm returns

5.2 Application in Abstract Extraction

When we implement text summarization by using MMR algorithm, we should re-interpret the above symbols:

1) Both Q and C stand for the whole text.
2) d represents a single sentence in the text.
3) $\text{sim}(Q, d_i)$ means the similarity of a sentence with the whole text.
4) $\text{sim}(d_i, d_j)$ means the similarity of a sentence with the sentence which has been extracted as a part of the abstract.
5) We use cosine similarity as the similarity between sentences. We need to first derive the sparse matrix of each sentence based on its word frequency, then calculate the cosine similarity of the two compared sparse matrices.

6) We set $\lambda = 0.7$ to balance the relevance and diversity

As a result, we will combine the k sentences as the abstract of the input text. Such a rank algorithm considers both relevance and diversity.

5.3 Code Implementation

We use a DataFrame to store the papers and relevant data. There are three parts mainly to implement: text processing, similarity calculation, and evaluation. The detailed code is shown in Appendix.

In the text processing, firstly we create a list [‘!’,’?’,’.’] to cut the sentences in the text. Then for each sentence, we have three different methods (all words, stemmed, stemmed+stop-words) to store the words for later similarity calculation. We split the sentence by a space for first method. And we use PorterStemmer() to recognize the word in different tenses as the same for the second method. Based on the second method, we import English stop-words to delete some unnecessary words before recognition as our third method. To return the origin sentences to the output abstract, we also conduct a dictionary to store the modified sentences (which are used to calculate the similarity) as keys and the origin sentences as values. Moreover, we define a function to count the number of sentences in the text, which influences k by multiplying a percentage variable we study.

As for the similarity calculation, we first derive the sparse matrix of each sentence based on its word frequency, then calculate the cosine similarity of the two compared sparse matrices. We use CountVectorizer() from sklearn package to help us generalize the sparse matrices of sentences, and then use cosine_similarity() to calculate the cosine similarity of the two matrices.

```
def calculateSimilarity(sentence, doc):
    if doc == []:
        return 0
    vocab = {}
    for word in sentence.split():
        vocab[word] = 0
    docInOneSentence = ''
    for t in doc:
        docInOneSentence += (t + ' ')
        for word in t.split():
            vocab[word] = 0
    cv = CountVectorizer(vocabulary=vocab.keys())
    docVector = cv.fit_transform([docInOneSentence])
    sentenceVector = cv.fit_transform([sentence])
    return cosine_similarity(docVector, sentenceVector)[0][0]
```

We use Rouge-1 f score as our evaluation standard. The f score ranges from 0 to 1. For each row of the DataFrame, we calculate the f score between the output abstract and the artificial abstract. Then we get the mean of all pairs as the final result.

6. Sample Result

We show a sample result of extracting abstracts of a text using 3 models (TextRank, TextRank with GloVe, and MMR). The text is named *Veterinary Science Molecular characterization and genogrouping of VP1 of aquatic birnavirus GC1 isolated from rockfish Sebastes schlegeli in Korea*. For each model, we do the stemming and remove the stopping words, and the number of sentences in the abstract is set to be 7% of the total number of sentences in the text.

Manually assigned abstract:
"The cDNA nucleotide sequence of genome segment B encoding the VP1 protein was determined for the aquatic birnavirus GC1 isolated from the rockfish Sebastes schlegeli in Korea. The VP1 protein of GC1 contains a 2,538 bp open reading frame, which encodes a protein comprising 846 amino acid residues that has a predicted MW of 94 kDa. The sequence contains 6 potential Asn-X-Ser/Thr motifs. Eight potential Ser phosphorylation sites and 1 potential Tyr phophorylation site were also identified. GC1 contains the Leu-Lys-Asn (LKN) motif instead of the typical Gly-Asp-Asp (GDD) motif found in other aquatic birnaviruses. We also identified the GLPYIGKT motif, the putative GTPbinding site at amino acid position 248. In total, the VP1 regions of 22 birnavirus strains were compared for analyzing the genetic relationship among the family Birnaviridae. Based on the deduced amino acid sequences, GC1 was observed to be more closely related to the infectious pancreatic necrosis virus (IPNV) from the USA, Japan, and Korea than the IPNV from Europe. Further, aquatic birnaviruses containing GC1 and IPNV have genogroups that are distinct from those in the genus Avibirnaviruses and Entomo-birnaviruses. The birnavirusstrains were clustered into 5 genogroups based on their amino acid sequences. The marine aquatic birnaviruses (MABVs) containing GC1 were included in the MABV genogroup; the IPNV strains isolated from Korea, Japan, and the USA were included in genogroup 1 and the IPNV strains isolated primarily from Europe were included in genogroup 2. Avibirnaviruses and entomobirnaviruses were included in genogroup 3 and 4, respectively."

TextRank output abstract:
"However, they also discovered that the typical Gly-Asp-Asp (GDD) motif that is found in all RNA viruses was absent in the VP1 region of some IPNV [4] IBDV, and DXV [2] strains. Further, we confirmed the 'GLPYIGKT'motif at amino acid position 248; this motif is the putative GTP-binding site that is commonly found in other aquatic birnaviruses. On comparing the nucleotide sequences of VP1 in 22 birnavirus strains, it was found that GC1 shares 97-98% homology with MABVs; 86% homology with the IPNV strains of aquabirnaviruses isolated mainly from the USA, Japan, and Korea; 80-82% homology with the IPNV strains of aquabirnaviruses from Spain; 54-56% homology with the avibirnaviruses; and 46% homology with entomobirnaviruses (Table 4). As reported previously [1, 5, 17] , the GDD sequence is a highly conserved motif that is present in almost all putative RdRps."

TextRank selected sentences & scores

Index	Selected Sentences	TextRank Scores
34	On comparing the nucleotide sequences of VP1 in 22 birnavirus strains, …	1.705879735288319
48	As reported previously [1, 5, 17] , …	1.6868919909418922
33	Further, we confirmed the 'GLPYIGKT'motif at amino acid position 248; …	1.6567037105313585
9	However, they also discovered that …	1.641616821874964

TextRank with GloVe output abstract:

"However, they also discovered that the typical Gly-Asp-Asp (GDD) motif that is found in all RNA viruses was absent in the VP1 region of some IPNV [4] IBDV, and DXV [2] strains. The amino acid sequence of VP1 did not contain the GDD motif, which exists commonly in the RdRps of RNA viruses; however, we could identifythe Leu-Lys-Asn (LKN) motif at position 521 (Table 3). In the phylogenetic cladograms that were based on both nucleotide and amino acid sequences, the genetic relationships among the 22 birnaviruses were established and the viruses, including GC1, were clustered into 5 genogroups that generally correlated with the geographic origin of the viruses and the water environment of the host. Researchers have found that the Asp-Asp (DD) sequence lacking Gly, is conserved in IBDV, and also that IPNV does not contain the typical GDD motif in the corresponding region of its VP1 [4, 21]."

TextRank with GloVe selected sentences & scores

Index	Selected Sentences	TextRank+GloVe Scores
36	In the phylogenetic cladograms that were based on both nucleotide and amino acid sequences, …	0.022086582483392153
49	Researchers have found that the Asp-Asp (DD) sequence lacking Gly, …	0.02167297598202699
9	However, they also discovered that …	0.02161157071159617
32	The amino acid sequence of VP1 did not contain the GDD motif, …	0.02146134563184409

MMR output abstract:

"The VP1 amino acid sequence was scanned for several functional motifs, and the results are summarized in Table 3. The amino acid sequence of VP1 did not contain the GDD motif, which exists commonly in the RdRps of RNA viruses; however, we could identify the Leu-Lys-Asn (LKN) motif at position 521 (Table 3). On comparing the nucleotide sequences of VP1 in 22 birnavirus strains, it was found that GC1 shares 97-98% homology with MABVs; 86% homology with the IPNV strains of aquabirnaviruses isolated mainly from the USA, Japan, and Korea; 80-82% homology with the IPNV strains of aquabirnaviruses from Spain; 54-56% homology with the avibirnaviruses; and 46% homology with entomobirnaviruses (Table 4). We believe that this motif represents a potential GTP-binding site in the VP1 protein, and has been conserved in GC1, including aquatic birnaviruses."

MMR selected sentences & scores

Index	Selected Sentences	MMR Scores
28	The VP1 amino acid sequence was scanned for several functional motifs, …	0.31447897413281023
34	On comparing the nucleotide sequences of VP1 in 22 birnavirus strains, …	0.23406283455127913
32	The amino acid sequence of VP1 did not contain the GDD motif, …	0.2310437885614062
47	We believe that this motif represents a potential GTP-binding site in the VP1 protein, …	0.20013056323279976

7. Evaluation

7.1 The ROUGE Evaluation Toolkit

We evaluate our three models using the ROUGE[2] evaluation toolkit, which can be used to automatically evaluate the quality of a generated summary compared to the manual abstract (Lin, 2004). ROUGE-N is based on n-gram co-occurrence statistics. It works by recall between our generated summary and the manual abstract. Formally,

$$\text{ROUGE-N} = \frac{\sum_{S\in\{ReferenceSummaries\}}\sum_{gram_n\in S} Count_{match}(gram_n)}{\sum_{S\in\{ReferenceSummaries\}}\sum_{gram_n\in S} Count(gram_n)}.$$

n: length of $gram_n$

$\boldsymbol{Count_{match}(gram_n)}$**:** maximum number of n-grams co-occurring in the summary and the abstract

In this research, we use the ROUGE-1 score, which is a real number value between 0 and 1. A higher score represents a higher quality of the summary.

7.2 Results Using ROUGE-1 Score

We also run our three models with different data preprocessing methods[3] and different percentages, which are used to control the number of sentences in the summary. Given a percentage p, n is determined by

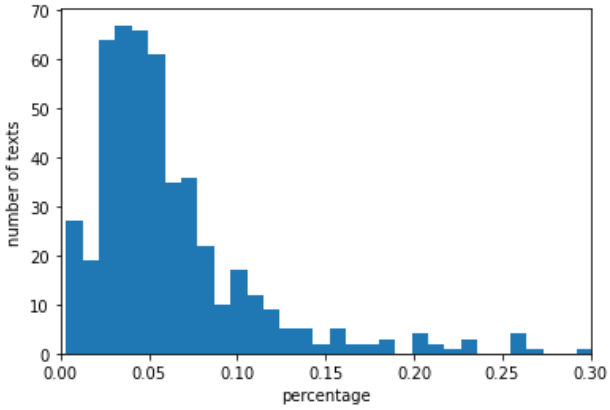
$$n = ceiling(p|T_i|),$$

[2] We use the rouge package from https://github.com/pltrdy/rouge

[3] Data preprocessing methods are introduced in Section 2.3.

where T_i is the i-th text ($T_i = S_1^i, S_2^i, ...$). For example, if the whole text contains 100 sentences and the percentage is 5%, it means that the abstract is expected to have 5 sentences.

To explore the best percentage, we use the original data to find what are the percentages of manual abstracts. The statistics show that the median percentage is 5.08% and the average percentage is 8.60%.



Therefore, we set the percentage to be 5%, 6%, 7%, 8% and 9%.

Notice that in TextRank with Glove model, we can only perform the stemmed+stopwords data preprocessing method due to the characteristics of GloVe. "n/Group" stands for running TextRank with Glove using n number of texts per time. TextRank and MMR only consider running the model one text at a time[4] so far.

ROUGE-1 Score									
	All words		Stemmed		Stemmed + Stopwords				
Percentage	TextRank	MMR	TextRank	MMR	TextRank with GloVe			TextRank	MMR
					1/Group	5/Group	10/Group		
5%	0.2666	0.2606	0.2663	0.2615	0.2616	0.2826	0.2975	0.2803	0.2937
6%	0.2651	0.2634	0.2698	0.2688	0.2615	0.2813	0.3023	0.2804	0.2974
7%	0.2639	0.2608	0.2664	0.2647	0.2602	0.2819	0.2999	0.2801	0.2967
8%	0.2614	0.2561	0.2638	0.2614	0.2599	0.2762	0.2976	0.2771	0.2953
9%	0.2580	0.2546	0.2618	0.2603	0.2574	0.2744	0.2970	0.2743	0.2932

[4] Running the model one text at a time is the same as "1/Group" processing.

7.3 Comments on the Results

1) The best model is TextRank with Glove model with a 6% percentage. Its ROUGE-1 score reaches 0.3023.

2) For each model, the best percentage is either 5% or 6%.

3) From the table, we conclude that Stemmed + Stopwords processing has the best performance. And Stemmed processing is slightly better than All words processing. Stemmed + Stopwords processing transforms the words in different tenses into their original form or simplified form and deletes some unnecessary words such as 'this' and 'can' before similarity calculation. This helps with the accuracy of similarity calculation and contributes to the ranking of sentences. It's reasonable for this processing method to dominate.

4) When we use Stemmed + Stopwords processing method, MMR performs better than TextRank. We think this is because of the consideration of abstract diversity in MMR algorithm. The algorithm avoids combining sentences with similar meanings into the abstract. So that the output sample reflects more information about the body text. Such an abstract corresponds to people's writing habits and therefore has a higher rouge score. Instead, TextRank may maintain the high relevance between the sentences in the abstract, which may cause an overlap in the meaning of the abstract.

5) When making text summaries with GloVe model, we try to divide the data into groups, both by groups of five and groups of ten, that is to say, putting articles together of different numbers to make summaries and generate separated text summaries of these articles. In terms of results, groups of ten performed best. We believe that this is due to the fact that multiple articles in a group can generate more connections between sentences, thus improving the accuracy of sentence ranking. If we apply this method to short texts, the effect will be more obvious. Through the combination of short articles, the richness of articles can be increased and the accuracy of abstracts can be improved, but only if all the articles are on the same subject.

6) In comparison with TextRank and MMR methods, we find that the accuracy of GloVe model is lower than the other two methods in processing a single article. We believe that this result is due to the lack of neologisms in the word vector set used by GloVe model. The word vector set we use is published by Stanford University in 2014 and has not been updated in the following years. Since the COVID-19 outbreak began in 2019, not only new words are in the articles, but also words that were once underused but now in frequent use. This will cause bias in the word vector allocation, resulting in inaccurate results.

8. Discussion

1) In practice, we only use GloVe method to group and process articles, but in principle, TextRank and MMR can also group and process articles with the same theme. In theory, group processing should be better than single-article processing because it can combine more relevant information and make more connections between words and sentences.

2) The three extractive methods are all extractive methods in Natural language processing (NLP). The advantages of extractive methods are that their implementation is convenient and simple, and the outputs have little bias in the topic since the sentences were from the text body. However, the output abstract may fail to achieve narrative continuity and is hard to control its number of words. The output abstract quality highly depends on the text body quality. Therefore, we can choose abstractive methods to avoid these problems. Those methods have high flexibility to generate new words or sentences. The classic abstractive methods are LSTM and Seq2seq+Attenrion.

3) TextRank, TextRank with GloVe and MMR can also be applied in keyword extraction, which is similar to our abstract extraction by setting vertices to be words and adding edges whenever two words co-occurrence frequently.

9. Conclusion

In this project, we compare three different extractive algorithms to implement text summarization for 100 papers about Covid-19: TextRank, Max Marginal Relevance (MMR), and TextRank with GloVe. We test them in three distinguish word processing (All words, Stemmed, Stemmed + Stopwords) and explore the best percentage to generate an abstract from the text body. The result reflects that MMR performs better than TextRank due to its consideration of abstract diversity. Text with GloVe algorithm performs better when selecting more samples as a group. The most appropriate percentage to generate an abstract from the text body ranges from 5% to 6%. After reviewing all results, more texts can be tried to run at a time when applying MMR or TextRank algorithm to improve the performance in the future.

References

Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7), 107-117.

Carbonell, J., & Goldstein, J. (1998, August). The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 335-336).

Lin, C. Y. (2004, July). Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out* (pp. 74-81).

Mihalcea, R., & Tarau, P. (2004, July). Textrank: Bringing order into text. In *Proceedings of the 2004 conference on empirical methods in natural language processing* (pp. 404-411).

Xu, C. & Chen, L.R. (2019). Automatic text summarization algorithm based on TextRank and GloVe. *China New Communications* (09), 90-92.

Appendix

Complete Code for TextRank

```
import math
import numpy as np
import pandas as pd
import nltk
from nltk.stem import PorterStemmer
from heapq import nlargest
from rouge import Rouge

# for the first time running nltk
# nltk.download('punkt')

# calculate the weight/similarity of Sentence 1 & Sentence 2
def calculate_similarity(s1,s2):
    num_of_same_word = 0
    for word in s1:
        if word in s2:
            num_of_same_word += 1
    if num_of_same_word == 0:
        return 0
    # to avoid zero denominator
    elif len(s1)==1 and len(s2)==1 and num_of_same_word == 1:
        return 1
    else:
        return num_of_same_word / (math.log(len(s1)) + math.log(len(s2)))

"""
generate the weights of the weighted graph
the input should be a list of sentences, and each sentence is decomposed into
 a list of words
i.e. text = [s1,s2,...,sn], where si =[w1_i,w2_i,...,wN_i]
"""
def generate_weights(text):
    n = len(text)
    # original weighted graph: nxn 0 matrix
    weights = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            # avoid empty sentences
```

```
            if i!= j and len(text[i])>0 and len(text[j])>0:
                weights[i][j] = calculate_similarity(text[i] , text[j])
    return weights

# for vextex i
def calculate_score(weights, scores, i):
    n = len(scores)
    # damping factor d, which is usually set to 0.85
    d = 0.85
    summation = 0
    for j in range(n):
        if sum(weights[j]) != 0:
            summation += weights[j][i]*scores[j]/sum(weights[j])

    score_i = (1 - d) + d * summation
    # return the score of vertex i
    return score_i

def converge(scores, old_scores):
    cvg = True
    threshold = 0.0001
    for i in range(len(scores)):
        if math.fabs(scores[i] - old_scores[i]) >= threshold:
            cvg = False
            break
    return cvg

def textrank(weights):
    n=len(weights)
    # initial value does not matter
    # scores = [1 for _ in range(n)]
    scores = np.ones(n)
    old_scores = np.zeros(n)

    # start iteration
    while not converge(scores, old_scores):
        old_scores = scores.copy()
        scores = [calculate_score(weights, scores, i) for i in range(n)]

    return scores


def Summarize(text, prec, stem =True, stop=True):
    # get the list of sentences of the text
```

```
    sentences = np.array(nltk.sent_tokenize(text), dtype=object)

    # decompose each sentence into a list of words
    # i.e. decomposed_sentences[i]: i-th sentence
    #      decomposed_sentences[i][j]: j-th word in the i-th sentence
    decomposed_sentences = []
    for s in sentences:
        words = nltk.word_tokenize(s)
        words = [word.lower() for word in words if word.isalpha()]
        decomposed_sentences.append(words)

    if stop == True:
        # remove the stopping words
        for i in range(len(decomposed_sentences)):
            for word in decomposed_sentences[i]:
                if word in stopwords:
                    decomposed_sentences[i].remove(word)

      if stem == True:
          # stemming the word
          stemmer = PorterStemmer()
          for i in range(len(decomposed_sentences)):
              for j in range(len(decomposed_sentences[i])):
                  decomposed_sentences[i][j] = stemmer.stem(decomposed_sente
nces[i][j])

      similarity_graph = generate_weights(decomposed_sentences)
      scores = textrank(similarity_graph)

      n = math.ceil(len(sentences)*prec)
      sent_index = list(map(scores.index, nlargest(n, scores)))

      print(f'top {n}: ', nlargest(n, scores))
      print(f'top {n}: ', sent_index)

      sent_index.sort()
      str = ' '
      summary =  str.join([sentences[i] for i in sent_index])
      return summary


  # evaluation
  def rouge(a,b):
      rouge = Rouge()
```

```
        rouge_score = rouge.get_scores(a,b, avg=True)
        r1 = rouge_score["rouge-1"]["f"]
        return r1

    global stopwords
    stopwords = np.array([line.strip() for line in open("stop_words_english.txt", 'r').readlines()])

    df=pd.read_csv('New_covid-19.csv')
    abstracts = df.abstract.drop(index=66)
    texts= df.text_body.drop(index=66)
    n = 101
    r1 = []
    for i in range(n):
        try:
            print(f'\n==========Text {i} Summary===========\n')
            summary = Summarize(texts[i],0.05)
            abstract = abstracts[i]
            r = rouge(summary, abstract)
            r1.append(r)
            print(r)
        except:
            pass

    print("=======final results======")
    print("mean(r1): ",np.mean(r1))
```

Complete Code for TextRank with GloVe

One by one:

```
import numpy as np
import pandas as pd
import nltk
import re
import warnings
from rouge import Rouge
import math
warnings.filterwarnings("ignore")
# nltk.download('punkt')  # download punkt
# nltk.download('stopwords')  # download stopwords

# input dataset
df = pd.read_csv("New_covid-19.csv", index_col=False)
```

```
df=df[~(df['abstract'].isnull())]  # delete empty rows
df=df[~(df['text_body'].isnull())]  # delete empty rows
df.drop([66], inplace=True)
df.index = range(len(df))



# train word vectors
word_embeddings = {}
f = open('glove.6B.100d.txt', encoding='utf-8')
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    word_embeddings[word] = coefs
f.close()




def rank(sentences, index):
    # clean the texts (including removing punctuation, numbers, Special characters and unifying into lowercase letters)
    clean_sentences = pd.Series(sentences).str.replace("[^a-zA-Z]", " ")
    clean_sentences = [s.lower() for s in clean_sentences]

    # remove the stopwors in sentences
    from nltk.corpus import stopwords
    stop_words = stopwords.words('english')
    def remove_stopwords(sen):
        sen_new = " ".join([i for i in sen if i not in stop_words])
        return sen_new
    clean_sentences = [remove_stopwords(r.split()) for r in clean_sentences]


    # Eigenvectors of sentences
    # We first obtain the vectors of all the constituent words of each sentence (obtained from the GloVe word vector file, with each vector size of 100 elements),
    # then take the average value of these vectors and obtain the combination vector of this sentence as the feature vector of this sentence.
    sentence_vectors = []
    for i in clean_sentences:
        if len(i) != 0:
```

```
            v = sum([word_embeddings.get(w, np.zeros((100,))) for w in i.split()])/(len(i.split())+0.001)
        else:
            v = np.zeros((100,))
        sentence_vectors.append(v)

    # Similarity matrix
    # Define an n-by-n zero matrix and fill it with cosine similarity between sentences, where n is the total number of sentences.
    sim_mat = np.zeros([len(sentences), len(sentences)])
    # Use cosine similarity to initialize the similarity matrix.
    from sklearn.metrics.pairwise import cosine_similarity
    for i in range(len(sentences)):
        for j in range(len(sentences)):
            if i != j:
                sim_mat[i][j] = cosine_similarity(sentence_vectors[i].reshape(1,100),
                sentence_vectors[j].reshape(1,100))[0,0]

    # Apply PageRank Algorithm
    # The similarity matrix Sim_mat is transformed into a graph structure. The nodes of this graph are sentences,
    # and edges are represented by similarity scores between sentences. On this graph, we will apply the PageRank algorithm to get the sentence rankings.

    import networkx as nx
    nx_graph = nx.from_numpy_array(sim_mat)
    scores = nx.pagerank(nx_graph)

    # extract abstracts
    # generate abstract according to the top N
    ranked_sentences = sorted(((scores[i],s) for i,s in enumerate(sentences)), reverse=True)
    top_sentences = ''
    num = math.ceil(len(scores) * 0.09)
    for i in range(num):
        top_sentences += " " + ranked_sentences[i][1]

    def rouge(a,b):
        rouge = Rouge()
        rouge_score = rouge.get_scores(a,b, avg=True)
        r1 = rouge_score["rouge-1"]["f"]
        return r1
```

```
89.
90.     a = df.loc[index, 'abstract']
91.     # print('original abstract: ', a)
92.     print('Predict abstract:', top_sentences)
93.     print(rouge(a,top_sentences))
94.     return(rouge(a,top_sentences))
95.
96.
97. # split text into seperated sentences
98. from nltk.tokenize import sent_tokenize
99. score = 0
100.   index = 0
101.   for s in df[:1]['text_body']:
102.       num = index
103.       index += 1
104.       print((index), ":")
105.       sentences = sent_tokenize(s)
106.       score += rank(sentences, num)
107.   score = score/index
108.   print("total score = ", score)
```

n/Group:

```
1.  import numpy as np
2.  import pandas as pd
3.  import nltk
4.  import re
5.  import warnings
6.  from rouge import Rouge
7.  warnings.filterwarnings("ignore")
8.  # nltk.download('punkt')  # download punkt
9.  # nltk.download('stopwords')  # download stopwords
10.
11. # input dataset
12. df = pd.read_csv("New_covid-19.csv", index_col=False)
13. df=df[~(df['abstract'].isnull())]  # delete empty rows
14. df=df[~(df['text_body'].isnull())]  # delete empty rows
15. df.drop([66], inplace=True)
16. df.index = range(len(df))
17.
18. # train word vectors
19. word_embeddings = {}
20. f = open('glove.6B.100d.txt', encoding='utf-8')
21. for line in f:
22.     values = line.split()
```

```
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    word_embeddings[word] = coefs
f.close()

def rank(sentences, index):
    # clean the texts (including removing punctuation, numbers, Special characters and unifying into lowercase letters)
    clean_sentences = pd.Series(sentences).str.replace("[^a-zA-Z]", " ")
    clean_sentences = [s.lower() for s in clean_sentences]

    # remove the stopwors in sentences
    from nltk.corpus import stopwords
    stop_words = stopwords.words('english')
    def remove_stopwords(sen):
        sen_new = " ".join([i for i in sen if i not in stop_words])
        return sen_new
    clean_sentences = [remove_stopwords(r.split()) for r in clean_sentences]


    # Eigenvectors of sentences
    sentence_vectors = []
    for i in clean_sentences:
        if len(i) != 0:
            v = sum([word_embeddings.get(w, np.zeros((100,))) for w in i.split()])/(len(i.split())+0.001)
        else:
            v = np.zeros((100,))
        sentence_vectors.append(v)

    # Similarity matrix
    sim_mat = np.zeros([len(sentences), len(sentences)])
    # Use cosine similarity to initialize the similarity matrix.
    from sklearn.metrics.pairwise import cosine_similarity
    for i in range(len(sentences)):
        for j in range(len(sentences)):
            if i != j:
                sim_mat[i][j] = cosine_similarity(sentence_vectors[i].reshape(1,100), sentence_vectors[j].reshape(1,100))[0,0]

    # Apply PageRank Algorithm
    import networkx as nx
    nx_graph = nx.from_numpy_array(sim_mat)
    scores = nx.pagerank(nx_graph)
```

```
63.
64.     # extract abstracts
65.     # generate abstract according to the top N
66.     ranked_sentences = sorted(((scores[i],s) for i,s in enumerate(sentences))
    , reverse=True)
67.     top_sentences = ''
68.     num = round(len(scores) * 0.09)
69.     for i in range(num):
70.         top_sentences += " " + ranked_sentences[i][1]
71.
72.     def rouge(a,b):
73.         rouge = Rouge()
74.         rouge_score = rouge.get_scores(a,b, avg=True)
75.         r1 = rouge_score["rouge-1"]["f"]
76.         return r1
77.
78.     a = ''
79.     for s in df[index:(index + 10)]['abstract']:
80.         a += " " + s
81.     print(rouge(a,top_sentences))
82.     return(rouge(a,top_sentences))
83.
84.
85. # split text into seperated sentences
86. from nltk.tokenize import sent_tokenize
87. # sentences = []
88. score = 0
89. index = 0
90.
91. for i in range(10):
92.     index += 1
93.     print(index, ":")
94.     sentences = []
95.     for s in df[i:(i+10)]['text_body']:
96.         sentences.append(sent_tokenize(s))  # sentences:[[,...,],[,...,],[,..
    .,]]
97.     sentences = [y for x in sentences for y in x]  # sentences:[,...,]
98.     score += rank(sentences, index-1)
99. score = score/index
100.   print("total score = ", score)
```

Complete Code for MMR

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import nltk
import re
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import operator
from nltk.stem import PorterStemmer
from rouge import Rouge
import pandas as pd
import numpy as np


# In[ ]:


stopwords = [line.strip() for line in open("C:/Users/Windyyam/Desktop/ERG3020
/Project 1/MMR/stop_words_english.txt", 'r',encoding='utf-8').readlines()]
stemmer = PorterStemmer()
path = 'C:/Users/Windyyam/Desktop/ERG3020/Project 1/MMR/New_covid-19.csv'
data = pd.read_csv(path,encoding='utf-8')
alpha = 0.7


# In[ ]:


del data['doc_id'],data['source'],data['title']
data = data.drop(66,axis=0)
data1 = data[0:55]
data


# In[ ]:


def cleanData(name):
    cut = re.sub('[^\w ]','',name)
```

```
    setlast = nltk.word_tokenize(cut)
    seg_list = [i.lower() for i in setlast if i not in stopwords]
    # seg_list = [i.lower() for i in setlast]
    for i in range(0,len(seg_list)):
        seg_list[i] = stemmer.stem(seg_list[i])
    return " ".join(seg_list)


# In[ ]:


def dic (group,sentence):
    puns = ['.','!','?'] #Seperate the article by ".","!","?"
    tmp = []
    parts = []
    for ch in group[sentence]:
        tmp.append(ch) #Travse the article by character
        if puns.__contains__(ch): #if encounter end punctuation
            parts.append(''.join(tmp))
            tmp = []
    clean = []
    orginSent = {}
    for part in parts:
        cl = cleanData(part)  # cut the sentences and delete stopwords
        clean.append(cl)  # sentences may have overlap
        orginSent[cl] = part
    group['clean'] = clean
    group['setClean'] = set(clean)  # sentences no overlap
    group['orginSent'] = orginSent
    return group


# In[ ]:


def calculateSimilarity(sentence, doc):  # calculate the cosine similarity
    if doc == []:
        return 0
    vocab = {}
    for word in sentence.split():
        vocab[word] = 0  # Add the words in the sentence into the vocabulary,
    value = 0.
    docInOneSentence = ''
    for t in doc:
```

```
85.         docInOneSentence += (t + ' ')  # combine all the remaining sentences
86.         for word in t.split():
87.             vocab[word] = 0  # add the words in the remaining sentences into vacabulary, value = 0.
88.     cv = CountVectorizer(vocabulary=vocab.keys())
89.     docVector = cv.fit_transform([docInOneSentence])
90.     sentenceVector = cv.fit_transform([sentence])
91.     return cosine_similarity(docVector, sentenceVector)[0][0]
92.
93.
94. # In[ ]:
95.
96.
97. def summarize(group,clean,setClean,count):
98.     print(group['Unnamed: 0.1'])
99.     sent_index = []
100.        summrayList = []
101.        scores = {}
102.        for data in group[clean]:
103.            temp_doc = group[setClean] - set([data])  # all sentences except for the current sentence.
104.            score = calculateSimilarity(data, list(temp_doc))  # cosine similarity of the current sentence with the remaining sentences
105.            scores[data] = score
106.        n = group[count]
107.        while n > 0:  #the number of output sentences
108.            mmr = {}
109.            for sentence in scores.keys():
110.                if not sentence in summrayList:
111.                    mmr[sentence] = alpha * scores[sentence] - (1 - alpha) * calculateSimilarity(sentence, summrayList)  # formula
112.            try:
113.                selected = max(mmr.items(), key=operator.itemgetter(1))[0]
114.                summrayList.append(selected)
115.                n -= 1
116.            except:
117.                n -= 1
118.        i = 1
119.        summrayList.sort(key=group[clean].index)  #make sure the order is the same
120.        for sentence in summrayList:
121.          #  print(i,end='')
122.           # print('、',end='')
```

```
            #print(orginSent[sentence])
            i += 1
            sent_index.append(group['orginSent'][sentence])
        group['output'] = ''.join(sent_index)
        return group


# In[ ]:


def count(group,text):
    total_sentences = nltk.sent_tokenize(group[text])
    num_of_total_sent = len(total_sentences)
    group['count'] = np.floor(num_of_total_sent*0.09) # number of sentence
s is 9% of the text body.
    return group


# In[ ]:


data1 = data1.apply(count,axis=1,text='text_body')


# In[ ]:


data1 = data1.apply(dic,axis=1,sentence='text_body')


# In[ ]:


data1 = data1.apply(summarize,axis=1,clean='clean',setClean='setClean',cou
nt='count')
data1


# In[ ]:


def evaluate(group,output,abstract):
    rouge = Rouge()
    try:
```

```
            rouge_socre = rouge.get_scores(group[output],group[abstract],avg=True)
            f1 = rouge_socre["rouge-1"]["f"]
            group['score'] = f1
        except:
            group['score'] = np.nan
        return group


# In[ ]:


final1 = data1.apply(evaluate,axis=1,output='output',abstract='abstract')


final1


# In[ ]:


data2 = data[55:110]
data2 = data2.apply(count,axis=1,text='text_body')


# In[ ]:


data2 = data2.apply(dic,axis=1,sentence='text_body')


# In[ ]:


data2 = data2.apply(summarize,axis=1,clean='clean',setClean='setClean',count='count')
data2


# In[ ]:


final2 = data2.apply(evaluate,axis=1,output='output',abstract='abstract')


final2
```

```
205.
206.
207.  # In[ ]:
208.
209.
210.  table = pd.concat([final1,final2])
211.
212.
213.  # In[ ]:
214.
215.
216.  table = table.dropna()
217.
218.
219.  # In[ ]:
220.
221.
222.  table = table.head(100)
223.  table
224.
225.
226.  # In[ ]:
227.
228.
229.  print(table['score'].mean())
230.
231.
232.  # In[ ]:
```