TEXT SUMMARIZATION USING LSA
BATCH 21

Syed Mohamed Asif - 2018103612
T.Athiban - 2018103013
Shree Harish - 2018103591

ABSTRACT

Due to an exponential growth in the generation of textual data, the need for tools and mechanisms for automatic summarization of documents has become very critical. Text documents are vital to any organization's day-to-day working and as such, long documents often hamper trivial work. Therefore, an automatic summarizer is vital towards reducing human effort. Text summarization is an important activity in the analysis of high volume text documents and is currently a major research topic in Natural Language Processing. It is the process of generation of the summary of input text by extracting the representative sentences from it. In this project, we present a novel technique for generating the summarization of domain specific text by using Semantic Analysis for text summarization, which is a subset of Natural Language Processing.

INTRODUCTION

Text summarization (or automatic summarization) is the creation of a shortened version of a text by a computer program. The product of this procedure still contains the most important points of the original text and is generally referred to as an abstract or a summary. Broadly, one distinguishes two approaches to text summarization:

Extraction
Abstraction

Extraction techniques merely copy information deemed to be most important by the system to the summary, while abstraction involves paraphrasing sections of the source document.

Abstraction can produce summaries that are more condensed than extraction, but these programs are considered much harder to develop.

Both techniques exploit the use of natural language processing and/or statistical methods for generating summaries. And, the classical approaches to text summarization proposed by Luhn et al have established the basis for the discipline of text summarization techniques.

The applicability of text summarization is increasingly being exploited in the commercial sector, in areas of telecommunications, data mining, information retrieval, and in word processing with high probability rates of success.

The Ultimate goal is to classify terms into different clusters of concepts. It is hypothesized that the number of clusters vary depending on the types of text documents

LITERATURE SURVEY / RELATED WORK

There are various text summarization approaches in literature. Most of them are based on extraction of important sentences from the input text. The first study on summarization, which was conducted in 1958 [11], was based on frequency of the words in a document. After this study other approaches arose, based on simple features like terms from keywords/key phrases, terms from user queries, frequency of words, and position of words/sentences. The algorithms belonging to Baxendale [12] and Edmundson [13] are examples of the approaches based on simple features. Statistical methods are another approach for summarization.

The SUMMARIST project [8] is a well-known text summarization project that uses a statistical approach. In this project, concept relevance information extracted from dictionaries and WordNet is used together with natural language-processing methods. Another summarization application based on statistics belongs to Kupiec et al. [14] where a Bayesian classifier is used for sentence extraction.

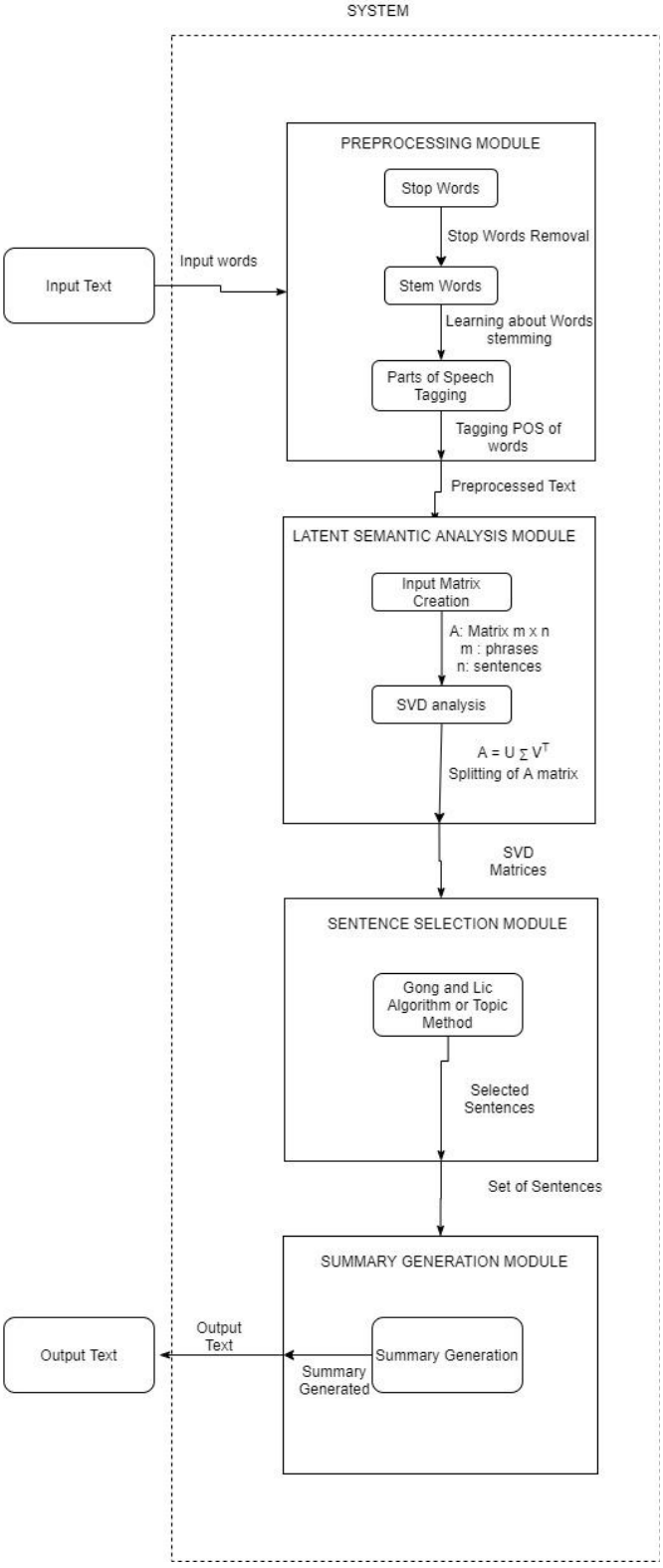
Text connectivity is another approach for dealing with problems of referencing to the already mentioned parts of a document. Lexical chains method [15, 16] is a well-known algorithm that uses text connectivity. In this approach, semantic relations of words are extracted using dictionaries and WordNet. Lexical chains are constructed and used for extracting important sentences in a document, using semantic relations.

There are graph-based summarization approaches for text summarization. As stated in Jezek and Steinberger [10], the well-known graph-based algorithms HITS [17] and Google's PageRank [18] were developed to understand the structure of the Web. These methods are then used in text summarization, where nodes represent the sentences, and the edges represent the similarity among the sentences. TextRank [19] and Cluster LexRank [20] are two methods that use a graph-based approach for document summarization.

There are also text summarization algorithms based on machine learning. These algorithms use techniques like Naı¨ve-Bayes, Decision Trees, Hidden Markov Model, Log–linear Models, and Neural Networks. More detailed information related to machine learning based text summarization approaches can be found in Das and Martins [2].

In recent years, algebraic methods such as LSA, Non-negative Matrix Factorization (NMF) and Semi-discrete Matrix Decomposition [21–23] have been used for document summarization. Among these algorithms the best known is LSA, which is based on singular value decomposition (SVD). Similarity among sentences and similarity among words are extracted in this algorithm. Other than summarization, the LSA algorithm is also used for document clustering and information filtering.

ARCHITECTURE DIAGRAM



MODULES

1. **Preprocessing Module**

   Input : The input text file name from the user
   Output : Words,Sentences without stop words and punctuations

   In the preprocessing stage, the input text would be broken down to sentences, and then to words. The main processes in preprocessing includes Stop Word removal, Part-of-Speech tagging and Word Stemming.

   Stop words removes the unnecessary words.
   Parts-of-Speech tagging tags each word with its parts of speech.
   Word Stemming gives the root word of the original word.

   Functions:-

   __init__(filename)

   1. The file is read and is stored in a variable.
   2. The sentences of the file are separated as a list and punctuation is removed from the sentence.
   3. The words of the sentences are separated and punctuations are removed from them.
   4. The words in each sentence are noted

   __punctuation_removal(tokens)

   1. This function takes the tokens and punctuations, hyphens and s word is removed from it.
   2. The remaining set of tokens are returned.

   stop_word_removal()

   1. This function removes stop words from the words list.
   2. The set of stop words is available from the nltk library in python.

3. Additional words can be added to this set.
4. They are removed because they do not contribute to the meaning of the text.

pos_tagging()

1. This function tags each word in a sentence with its corresponding Parts-of-Speech tag.
2. It is useful for the next step of stemming of words where word is stemmed based on its Parts-of-Speech.

word_stemming()

1. Each word is passed into a wordnet lemmatizer along with its corresponding Parts-of-Speech and the root word is found
2. The purpose of having Word Stemming is to have only one root word which could be written more than once in different formats.
3. Word Stemming deals with the prefix and suffix of each word.
4. This improves the accuracy of summary.
   Eg: given -> give

2. Latent Semantic Analysis Module

Input : The words and sentences from the preprocessing module
Output : The SVD analysis resulting matrices

Latent Semantic Analysis is an algebraic-statistical method that extracts hidden semantic structures of terms and sentences in a text. It is an unsupervised approach that does not need any training or external knowledge. A high number of common words indicates that the sentences are semantically related. The latent semantic analysis includes Input Matrix Creation and Singular Value Decomposition(SVD) Analysis.

Input Matrix Creation - The preprocessed text needs to be represented in a way understandable by a computer. Hence a matrix representation is used. There are different approaches for creating this matrix. The rows of

the matrix consists of the phrases that we get from the preprocessed module. The columns of the matrix represent the sentences in the document. The two approaches that we like to use based on performance,

Binary Representation: The cell is filled with 0/1 depending on the existence of a word in a sentence.
Tf-idf(Term Frequency-Inverse Document Frequency) : The cell is filled with the tf-idf value of the word. A higher tf-idf value means that the word is more frequent in the sentence but less frequent in the whole document.

Singular Value Decomposition(SVD) - In this method, the given input matrix A is decomposed into three new matrices as follows:
$A = U \sum V T$
Where A is the input matrix(m x n); U is words x extracted concepts (m x n); $\sum$ represents scaling values, diagonal descending matrix(n x n); and V is sentences x extracted concepts (n x n).

Functions:-

input_matrix_creation(method)

1. This function takes a method and directs the input matrix to be filled according to that method.

__init__()

1. The count of the documents is calculated.
2. The count of sentences are calculated.
3. The count of preprocessed words are calculated.

__binary_representation()

1. This method checks for each word, if the word is present in that sentence.
2. Then the entry is changed to 1 otherwise 0
3. The input matrix is returned

__tfidf_representation()

1. This method checks for each word, if the word is present in that sentence, then increment the count by 1.
2. Hence for now, we got the frequency matrix
3. Divide each element by the total number of words, we got the term frequency value for each term.
4. For each sentence in the word, the number of sentences with a word w is calculated.
5. For each word, the inverse document frequency is calculated.
6. inverse document frequency(t) = log(total no of documents / no of documents with term t)
7. Each entry in the matrix is multiplied with idf value.
8. We got the tf-idf value

__validate_num_of_concepts(num_of_concepts):

1. The rank of the input matrix is calculated.
2. The num_of_concepts should be less than the rank of the matrix
3. Otherwise the new concepts value is set and returned

__singular_value_decomposition(num_of_concepts):

1. The SVD of the input matrix is calculated.
2. Input_Matrix = U S Vt
3. The num_of_concepts is validated
4. The First num_of_concepts rows are taken from the U , S and Vt matrices.

3. Sentence Selection Module

Input : The SVD resulting matrices
Output : The selected sentences for summary

The sentences are selected using the SVD results and different algorithms. The two methods that we like to try are Gong and Liu Algorithm or Topic Method

- **Gong and Liu Algorithm**

  After representing the input document in the matrix and calculating SVD values, V^T matrix, the matrix of extracted concepts × sentences is used for selecting the important sentences. In V^T matrix, row order indicates the importance of the concepts, such that the first row represents the most important concept extracted. The cell values of this matrix show the relation between the sentence and the concept. A higher cell value indicates that the sentence is more related to the concept. Then one sentence is chosen from the most important concept, and then a second sentence is chosen from the second most important concept until a predefined number of sentences are collected. The number of sentences to be collected is given as a parameter.

- **Topic Method**

  In this approach, after deciding the main topics that may be a group of subtopics, the sentences are collected from the main topics as a part of the summary. First the average sentence score for each of the concepts in the V^T matrix. Then the cell values less than this average score is set to zero. This removes the sentences that are not highly related to the concept. Next, a concept x concept matrix is created by finding out the concepts that have common sentences. Then the new cell values of the concept × concept matrix are set to the total of common sentence scores. After the creation of the concept × concept matrix, the strength of each concept is calculated. For

each concept, the strength value is computed by getting the cumulative cell values for each row of the concept × concept matrix. The concept with the highest strength value is chosen as the main topic of the input document. After these steps, the sentences are collected from the pre-processed V^T matrix. Then a single sentence is collected from each concept until predefined numbers of sentences are collected.

4. Output Text Module

Input : The selected sentences
Output : The output summary text

The sentences selected from the Sentence selection module are put together in the form of a paragraph. And the resulting paragraph will be sent to the user as output text.

RESULTS DISCUSSION

1. **PREPROCESSING MODULE**

```
from numpy import array
from nltk import sent_tokenize, word_tokenize, pos_tag
from nltk.corpus import wordnet, stopwords
from nltk.stem import WordNetLemmatizer
from collections import defaultdict
import re


class PreProcessingModule:
    def __init__(self, filename):
        # Open the file and read the input as string
        file = open(filename)
        self.Paragraph = file.read()
        file.close()
```

```
        # Creating the sentences array from the given paragraph
        self.Sentences = array(sent_tokenize(self.Paragraph))

            # Removing punctuation from tokenized sentences and
lowercase them
                                        self.Lower_Sentences    =
self.__punctuation_removal(self.Sentences)

        # Creating the words array from the given paragraph
        self.Words = array(word_tokenize(self.Paragraph))

          # Removing punctuation from tokenized words and lowercase
them
        lower_words = []
        for sentence in self.Lower_Sentences:
            words = word_tokenize(sentence)
            words = self.__punctuation_removal(words)
            lower_words.extend(words)
        self.Lower_Words = lower_words

          # Creating the words in sentences array because words may
change
        self.Words_in_Sentences = {}

        # Creating the words - stem words mapping
        self.Stemmed_Words = {}

        # Creating the postag() to parts-of-speech tagging
        self.__POSDict = defaultdict(lambda: wordnet.NOUN)
        self.__POSDict['J'] = wordnet.ADJ
        self.__POSDict['V'] = wordnet.VERB
        self.__POSDict['R'] = wordnet.ADV
```

```
        # Creating the word to postag mapping
        self.__POS = defaultdict(lambda: wordnet.NOUN)

        # Finding words in final processed sentences
        for sentence in self.Lower_Sentences:
            words = word_tokenize(sentence)
            words = self.__punctuation_removal(words)
            self.Words_in_Sentences[sentence] = words

    @staticmethod
    def __punctuation_removal(tokens):
        # RE for removing punctuation
        punctuation = re.compile(r'[.?`\'$"!%,:;()|0-9]')
        hyphens = re.compile(r'^-')
        s = re.compile(r'^s$')

        # Removing punctuation from tokens and lowercase them
        post_punctuation = []
        for token in tokens:
            token = punctuation.sub("", token.lower())
            token = hyphens.sub("", token)
            token = s.sub("", token)
            if len(token) > 0:
                post_punctuation.append(token)
        return post_punctuation

    def stop_words_removal(self):

        # get stopwords from nltk library
        stop_words = set(stopwords.words('english'))
        stop_words.add('bn')
        stop_words.add('mr')

        # Removal of words from __Lower_Words
```

```
         filtered_tokens = [w for w in self.Lower_Words if w not in
stop_words]

        # making filtered_tokens as default tokens
        self.Lower_Words = filtered_tokens

    def pos_tagging(self):
        # Finding and storing POS of tokens
        for sentence in self.Lower_Sentences:
            words = self.Words_in_Sentences[sentence]
            pos_tags = list(pos_tag(words))
            for item in pos_tags:
                self.__POS[item[0]] = self.__POSDict[item[1][0]]

    def word_stemming(self):
        # Initializing word stemmer
        word_lemmatizer = WordNetLemmatizer()

        # Initializing stem removed lowercase words
        lower_words = []

        # Finding the stem of words and storing it
        for sentence in self.Lower_Sentences:
            sentence_words = self.Words_in_Sentences[sentence]
            for token in sentence_words:
                            stem  =  word_lemmatizer.lemmatize(token,
self.__POS[token])
                self.Stemmed_Words[token] = stem

        for token in self.Lower_Words:
                           stem   =   word_lemmatizer.lemmatize(token,
self.__POS[token])
            if stem not in lower_words:
                lower_words.append(stem)
```

```
        # Reinitializing lower words
        self.Lower_Words = lower_words
```

2. LATENT SEMANTIC ANALYSIS MODULE

```
from nltk.probability import FreqDist
from numpy import divide, zeros, array, diag
from numpy.linalg import svd
import math


class LatentSemanticAnalysisModule:
    def __init__(self, lower_sentences, lower_words, stemmed_words,
words_in_sentences):
        # Initialization with outputs of preprocessing module
        self.__Lower_Sentences = lower_sentences
        self.__Lower_Words = lower_words
        self.__Stemmed_Words = stemmed_words
        self.__Words_in_Sentences = words_in_sentences

        # Count of Document Words
        document_count = 0
        for _, val in self.__Words_in_Sentences.items():
            document_count += len(val)
        self.Document_Count = document_count

        # Count of Sentences and words
        self.__Sentences_Count = len(self.__Lower_Sentences)
        self.__Words_Count = len(self.__Lower_Words)

        # Word to their corresponding count mapping dictionary
        count_dict = FreqDist()
```

```
        for sentence in self.__Lower_Sentences:
            words = self.__Words_in_Sentences[sentence]
            for word in words:
                count_dict[word] += 1
        self.__Count_Dict = count_dict

        # Find the number of sentences with the word mapping
        self.__No_of_Sentences_with_Word = {}

        # Input Matrix with words on rows and sentences as columns
                        self.Input_Matrix   =   [[0   for   i   in
range(self.__Sentences_Count)] for j in range(self.__Words_Count)]

        # Variables instantiation for singular value decomposition
        self.U = []
        self.S = []
        self.Vt = []

    def __binary_representation(self):
        # iterate over preprocessed words
        for i in range(self.__Words_Count):
            # iterate over lowercase sentences
            for j in range(self.__Sentences_Count):
                # find the words in sentence
                                                sentence_words   =
self.__Words_in_Sentences[self.__Lower_Sentences[j]]

                # if word is present change the entry to 1
                for word in sentence_words:
                                    if  self.__Stemmed_Words[word]  ==
self.__Lower_Words[i]:
                        self.Input_Matrix[i][j] = 1

    def __tfidf_representation(self):
```

```
        # iterate over preprocessed words
        for i in range(self.__Words_Count):
            # iterate over lowercase sentences
            for j in range(self.__Sentences_Count):
                # find the words in sentence
                                                sentence_words   =
self.__Words_in_Sentences[self.__Lower_Sentences[j]]

                # if word is present increase the entry to 1
                for word in sentence_words:
                                    if  self.__Stemmed_Words[word]  ==
self.__Lower_Words[i]:
                        self.Input_Matrix[i][j] += 1

        # find the number of sentences with words count
        for i in range(self.__Words_Count):
            count = 0
            for j in range(self.__Sentences_Count):
                if self.Input_Matrix[i][j] != 0:
                    count += 1
             self.__No_of_Sentences_with_Word[self.__Lower_Words[i]]
= count

         # term frequency(t) = number of times word t appears in a
document/ total number of words in the document
                    self.Input_Matrix   =   divide(self.Input_Matrix,
self.Document_Count)

        for i in range(self.__Words_Count):
            for j in range(self.__Sentences_Count):
                  # inverse document frequency(t) = log( total no of
documents / no of documents with term t )
                            idf  =  math.log(self.__Sentences_Count  /
self.__No_of_Sentences_with_Word[self.__Lower_Words[i]])
```

```
                # input matrix = tf * idf
                 self.Input_Matrix[i][j] = self.Input_Matrix[i][j] *
idf

    def input_matrix_creation(self, method="Binary"):
        # Check which technique to use
        if method == "Binary":
            self.__binary_representation()
        else:
            self.__tfidf_representation()

    def __validate_num_of_concepts(self, num_of_concepts):
        # Find the number of linearly independent sentences
        # To calculate the rank of the input matrix
          set_of_sentences = set(frozenset(sentence.split(' ')) for
sentence in self.__Lower_Sentences)
        input_matrix_rank = len(set_of_sentences)

          # Validating num_of_concepts by comparing it with rank of
the input_matrix_rank
        if input_matrix_rank <= 1:
                print("The input matrix does not have the ranks to
compute SVD")
            exit(-1)
        if num_of_concepts > input_matrix_rank - 1:
            num_of_concepts = input_matrix_rank - 1

        return num_of_concepts

    def singular_value_decomposition(self, num_of_concepts=5):
        # SVD Calculation
                self.U,  self.S,  self.Vt  =  svd(self.Input_Matrix,
full_matrices=False)
```

```
        # Validate num of concepts
  num_of_concepts= self.__validate_num_of_concepts(num_of_concepts)

        # Resizing SVD matrix based on num_of_concepts
        self.Vt = self.Vt[:num_of_concepts, :]
        self.U = self.U[:, :num_of_concepts]
        self.S = self.S[0:num_of_concepts]
```

index.py

```
from latentSemanticAnalysis import LatentSemanticAnalysisModule
from preprocessing import PreProcessingModule

#PREPROCESSING MODULE
preprocessing = PreProcessingModule('input.txt')

# Removal of stop words
preprocessing.stop_words_removal()

# Tagging of Parts-of-Speech
preprocessing.pos_tagging()

# Stemming of words
preprocessing.word_stemming()

# Results of Preprocessing Module
#   Paragraph,   Sentences,   Words,   Lower_Sentences,   Lower_Words,
Words_in_Sentences, Stemmed_Words

#LATENT SEMANTIC ANALYSIS MODULE
lsa   =   LatentSemanticAnalysisModule(preprocessing.Lower_Sentences,
preprocessing.Lower_Words,
```

```
                                        preprocessing.Stemmed_Words,
preprocessing.Words_in_Sentences)

# Input Matrix Creation using Binary Representation
lsa.input_matrix_creation(method="Term")

# Singular value decomposition
lsa.singular_value_decomposition()

# Results of Latent Semantic Analysis Module
# Input Matrix, Singular Value Decomposition ( Input_Matrix = U . S
. Vt )
print(lsa.U.shape)
print(lsa.S.shape)
print(lsa.Vt.shape)
```

OUTPUTS(from index.py):-

After initializing the preprocessing Module, the sentences, word separation are done. The punctuations are removed from the sentence and words. The words and sentences are changed to lowercase. The output of first 10 words and 2 sentences are

In [1]:
```
from preprocessing import PreProcessingModule
from latentSemanticAnalysis import LatentSemanticAnalysisModule
```

In [31]:
```
# ------------------------------------ PREPROCESSING MODULE ------------------
preprocessing = PreProcessingModule('input.txt')
print("The first 10 words are")
for i in range(10):
    print(preprocessing.Lower_Words[i])
print("The first 2 sentences are")
for i in range(2):
    print(preprocessing.Lower_Sentences[i])
```

```
The first 10 words are
in
a
no-holds-barred
email
to
the
board
seen
by
the
The first 2 sentences are
in a no-holds-barred email to the board seen by the bbc cyrus mistry says he
had become a lame duck chairman and alleges constant interference including b
eing asked to sign off on deals he knew little about
he also warned the company risks huge writedowns across the business
```

After the stop words removal function, the stop words are removed and important words are kept along. The first 10 of important words are

In [33]:
```
# Removal of stop words
preprocessing.stop_words_removal()
print("The first 10 words are")
print()
for i in range(10):
    print(preprocessing.Lower_Words[i])
```

```
The first 10 words are

no-holds-barred
email
board
seen
bbc
cyrus
mistry
says
become
lame
```

After the stop words removal, The Parts-of-Speech is tagged for each word. The first 10 of those words and their POS are

In [6]:
```
# Tagging of Parts-of-Speech
preprocessing.pos_tagging()
i=0
for k,v in preprocessing.POS.items():
    if i==20:
        break
    print(k," ",v)
    i+=1
```

```
in   n
a   n
no-holds-barred   a
email   n
to   n
the   n
board   n
seen   v
by   n
bbc   n
cyrus   n
mistry   n
says   v
he   n
had   v
become   v
lame   a
duck   n
chairman   n
and   n
```

After the POS tagging, the stemming of words happens and the first 10 stemmed words are

```
In [11]: # Stemming of words
         preprocessing.word_stemming()
         print("The first 10 words are")
         print()
         for i in range(10):
             print(preprocessing.Lower_Words[i])
```

```
The first 10 words are

no-holds-barred
email
board
see
bbc
cyrus
mistry
say
become
lame
```

Also the first 2 sentences after the preprocessing module are

```
In [13]: for i in range(2):
             print(preprocessing.Lower_Sentences[i])
```

```
in a no-holds-barred email to the board seen by the bbc cyrus mistry says he
had become a lame duck chairman and alleges constant interference including b
eing asked to sign off on deals he knew little about
he also warned the company risks huge writedowns across the business
```

Now the latent semantic analysis module is initialized and the counts are calculated. The document count is

```
In [15]: # ---------------------------------- LATENT SEMANTIC ANALYSIS MODULE -------
         lsa = LatentSemanticAnalysisModule(preprocessing.Lower_Sentences, preprocessin
                                            preprocessing.Stemmed_Words, preprocessing.
         print(lsa.Document_Count)
```

```
580
```

After this the input matrix are created the one way is binary representation

```
In [13]: # Input Matrix Creation using Binary Representation
         lsa.input_matrix_creation(method="Binary")
```

```
In [14]: for row in lsa.Input_Matrix:
             print(row)
```

```
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 1. 0. 1. 0. 0. 1. 0. 1. 0. 0. 1. 0. 0. 1. 0. 1. 0. 0.]
[1. 0. 1. 0. 0. 1. 0. 0. 1. 1. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 0. 1. 1. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

Another way is using the tf-idf representation. The output matrix of this representation is

```
In [15]: # Input Matrix Creation using Tfidf Representation
         lsa.input_matrix_creation(method="Term")

In [16]: print(lsa.Input_Matrix)
```

```
[[0.01065877 0.         0.         ... 0.         0.         0.        ]
 [0.01065877 0.         0.         ... 0.         0.         0.        ]
 [0.0082686  0.         0.         ... 0.         0.         0.        ]
 ...
 [0.         0.         0.         ... 0.         0.         0.01065877]
 [0.         0.         0.         ... 0.         0.         0.01065877]
 [0.         0.         0.         ... 0.         0.         0.01065877]]
```

After the singular value decomposition is done and the corresponding three matrices are created.

```
In [17]: # Singular value decomposition
         lsa.singular_value_decomposition()

In [18]: # Results of Latent Semantic Analysis Module
         # Input Matrix, Singular Value Decomposition ( Input_Matrix = U . S . Vt )

In [20]: print(lsa.U)
         print(lsa.U.shape)
```

```
[[-0.00970837 -0.0485344   0.07930056  0.18957232 -0.05401419]
 [-0.00970837 -0.0485344   0.07930056  0.18957232 -0.05401419]
 [-0.00901322 -0.05899502  0.06402295  0.16251942 -0.01647585]
 ...
 [-0.0099493  -0.00469802  0.01397575  0.01389211  0.0038481 ]
 [-0.0099493  -0.00469802  0.01397575  0.01389211  0.0038481 ]
 [-0.0099493  -0.00469802  0.01397575  0.01389211  0.0038481 ]]
(215, 5)
```

```
In [21]: print(lsa.S)
         print(lsa.S.shape)
```

```
[0.05885848 0.04722967 0.04432922 0.04270219 0.04164473]
(5,)
```

```
In [22]: print(lsa.Vt)
         print(lsa.Vt.shape)
```

```
[[-5.36103370e-02 -2.13446698e-02 -1.07386225e-02 -7.87175814e-03
  -2.35856936e-02 -1.09019066e-01 -5.50586160e-03 -1.05485557e-02
  -1.56371066e-02 -1.42097395e-01 -3.07369251e-02 -9.76642020e-01
  -3.16492893e-03 -8.50841866e-03 -5.42407349e-02 -3.81534223e-02
  -1.52170527e-02 -2.46311036e-02 -1.28288776e-02 -7.08189188e-03
  -1.21582718e-02 -5.49407383e-02]
 [-2.15059001e-01 -1.03676478e-01 -1.42864088e-02 -2.80383156e-02
  -6.24137810e-02 -4.74375084e-01 -8.76634861e-03 -1.21916327e-01
  -8.69323424e-02 -7.45776981e-01 -2.23234889e-01  2.01409031e-01
  -2.22347612e-02 -1.83888507e-02 -3.87284395e-02 -1.58208860e-01
  -7.13260203e-02 -8.22753339e-02 -4.75531363e-02 -2.47321020e-02
  -3.84387477e-02 -2.08172393e-02]
 [ 3.29806627e-01 -2.68363317e-02  8.22917733e-03  4.54780758e-02
   7.77730458e-02  5.99729916e-01  1.88259159e-03  1.34299666e-02
  -4.05461247e-02 -6.11391939e-01  3.10491066e-01 -1.99393536e-02
   3.06705834e-02  8.48021318e-03  6.49055584e-02 -2.36622001e-02
   1.33458778e-01  1.17350446e-01  7.70304104e-02 -1.14490076e-02
   5.83585311e-02  5.81243779e-02]
 [ 7.59482978e-01 -1.70459060e-02  5.19557756e-03 -2.87823771e-02
   3.19837855e-02 -4.43205678e-01  1.13577670e-02  7.98284521e-02
   3.88470671e-02 -3.06635795e-02 -1.48951369e-01 -1.51538570e-02
  -1.81616738e-02  7.95070276e-03  1.10085442e-01  3.56527905e-01
   1.04588072e-01  1.90084936e-01  5.62500842e-02  1.27112331e-02
  -1.00058902e-02  5.56559362e-02]
 [-2.11038173e-01  3.09281715e-02  2.58538365e-03  5.93905555e-03
   2.72418279e-02 -3.19569951e-02  1.73311673e-03  1.28057754e-01
   3.43770028e-03  6.49940837e-03 -1.73476738e-01 -8.29224896e-03
  -5.14089749e-05 -1.93483832e-03  1.82031363e-01 -3.59044836e-02
   9.32222678e-01  9.11483468e-03  4.56974791e-02  5.02154626e-03
   1.38637280e-02  1.50348500e-02]]
(5, 22)
```

CONCLUSION

Text summarization is one of the major problems in the field of Natural Language Processing, and yet it is even after years of research and implementations, fraught with complications. However, there have been some major breakthroughs in the past, such as Columbia University's Multigen (1999) and Copy and Paste (1999), and USC's ISI Summarist. Many different methods were used to arrive at the final summary, whether that summary was abstractive or extractive. Methods such as Deep Understanding, Sentence Extraction, Paragraph Extraction, Machine Learning, and even some which employ all these methods along with

Traditional NLP Techniques(Semantic Analysis, etc.). As such, keeping these accomplishments in mind, there is still ample amount of research left in the domain of Text Summarization, as a meaningful summary is still difficult to attain in all domains and languages.

REFERENCES

1. “Text summarization using Latent Semantic Analysis”,

Makbule Gulcin Ozsoy and *Ferda Nur Alpaslan* Department of Computer Engineering,Middle East Technical University, Turkey Ilyas Cicekli Department of Computer Engineering, Hacettepe University, Turkey

2. “Text Summarization Using Latent Semantic Analysis Model in Mobile Android Platform”

Oi-Mean Foong, Suet-Peng Yoong and Farha-Am Jaid Computer and Information Sciences Department, Universiti Teknologi PETRONAS, Tronoh, Malaysia

3. Josef Steinberger, KarelJežek, “Using latent Semantic analysis In Text Summarization and Summary Evaluation”, Department of Computer Science and Engineering, UniverzitníCZ-306 14 Plzeň.

4. Jen-Yuan Yeh, Hao-RenKe, Wei-Pang Yang, IHengMeng, “Text summarization using a trainable summarizer and Latent semantic analysis”, Elsevier Proceeding of Information processing and management, No. 41, pp 75-95, 2016.

5. Edward Hovy, Chin-Yew Lin, “Automated Text Summarization and the Summarist System”, Information Sciences Institute of the University of Southern California.