**Automatic Text Summarizer Using GUI Application**

**NAME :- Kotturu Sai**

**Regd no.:- 170101120026**

**CSE 7th Semester**

**Campus :- Perlakhemundi**

**Table of Content**

**1. Abstract……………………………………………………………………..3**

**2. Introduction………………………………………… ……………..…3-4**

**2.1 Avantages of summarizer………………………………..…..4**

**2.2 Important functions of summarizer…………….…….….4**

**2.3 Main steps for summarization……………………….….….5**

**2.4 Features of summarizer………………………….………..….5**

**3. Proposed System……………………………………………….…..…6**

**4. Technology usssed………………………………………………...….6**

**4.1 NLTK……………………………………………………………...……6**

**4.2 GUI…………………………………………………………….……....8**

**4.3 Tkinter……………………………………………………….……....8**

**4.5 Bs4……………………………………………………………...………8**

**4.6 URL library………………………………………………...………..9**

**4.7 Heapq……………………………………………………….………...9**

**4.8 Stop words…………………………………………………..…….10**

**4.9 Python3……………………………………………………..………10**

**4.10 Machine learning……………………………………….…….10**

**5. Methodology………………………………………………….……10-11**

**6. Flow Chart…………………………………………………..…….….….11**

**7. Algorithm………………………………………………………….…12-14**

**8. Literature Survey………………………………………………….…..15**

**9. NLP……………………………………………………………………….…..16**

**9.1 Introduction to NLP………………………………………….…..16**

**9.2 What is NLP……………………………………………...............17**

**9.3 Text pre-processing………………………………………..…...18**

**9.3.1 Accessing data……………………………………..…18-19**

**9.4 Text-Searching………………………………………..……...19-21**

**9.5Tokenization…………………………………………………….21-22**

**9.6 Stop words……………………………………………………...22-24**

**9.7 Lexican Normalization…………………………………………..24**

**9.8 Stemming…………………………………………………………24-25**

**9.19 Lemitization……………………………………………………25-26**

**9.10 Parts of speech tagging…………………………………..26-27**

**9.11 Chunking……………………………………………………………..28**

**9.12 Named-Entity Recognization…………………………..28-29**

**9.13 TF-IDF……………………………………………………...............30**

**9.14 Lexical richness……………………………………………….31-32**

**9.15 Frequency distribution…………………………………...32-34**

**10. Project source code……………………………………………..34-36**

**11. Result……………………………………………………………………….36**

**11.1 Output 1……………………………………………………………..36**

**11.2 Output 2……………………………………………………………..37**

**11.3 Output 3……………………………………………………………..38**

**12. Conclusion……………………………………………………………….39**

**Abstract:-**

In this new era,where tremondous information is available on the internet,it is most important to provide the improved mechanism to extract the information quickly and most efficiently . It is very difficult for human beings to manually extract the summary of a large documents of text or text from the wikkipidea. There are plenty of text material available on the internet. So there is a problem of searching for relevant documents from the number of documents available, and absorbing relevant information from it.In order to solve the above two problems, the automatic text summarization is very much necessary.Text summarization is the process of identifying the most important meaningful information in a document or set of related documents and compressing them into a shorter version preserving its overall meanings.

**Introduction:-**

Before going to the Text summarization, first we, have to know that what a summary is. A summary is a text that is produced from one or more texts, that conveys important information in the original text, and it is of a shorter form. The goal of automatic text summarization is presenting the source text into a shorter version with semantics.The most important advantage of using a summary is ,it reduces the reading time. Text Summarization methods can be classified into extractive and abstractive summarization. An extractive summarization method consists of selecting important sentences, paragraphs etc. from the original document and concatenating them into shorter form. An Abstractive summarization is an understanding of the main concepts in a document and then express those concepts in clear natural language.

There are two different groups of text summarization : indicative and informative.Inductive summarization only represent the main idea of the text to the user. The typical length of this type of summarization is 5 to 10 percent of the main text.On the other hand, the informative summarization systems gives concise information of the main text .The length of informative summary is 20 to 30 percent of the main text .

**Advantages of text summarizer:-**

1. Several redundancies can be removed. The user does not waste time reading repetitive data.

2. Summarization allows you to remove data that is not essential to the understanding of the text.

3. Important thing is that its work instantly.

4. It works in any language.

5 .It increase productivity.

6. It does not miss important facts.

**Important functons of summarizer are:-**

1. Reducing a single document to a user-defined fraction of its original size while maintaining coherence.
2. Choosing the most relevant and important sentences from the text.
3. Improving the abstraction and/or length of the summary by using a thesaurus to replace semantically related units.

## Main steps for text summarization:-

There are three main steps for summarizing documents.These are topic identification, interpretation and summary generation.Topic Identificatio:The most prominent information in the text is identified .There are different techniques for topic identification are used which are Position, Cue Phrases, word frequency.Methods which are based on the position of phrases are the most useful methods for topic identification.

## Features of Summarizer:-

Most of the current automated text summarization systems use extraction method to produce a summary .Sentence extraction techniques are commonly used to produce extraction summaries. One of the methods to obtain suitable sentences is to assign some numerical measure of a sentence for the summary called sentence scoring and then select the best sentences to form document summary based on the compression rate. In the extraction method, compression rate is an important factor used to define the ratio between the length of the summary and the source text. As the compression rate increases, the summary will be larger, and more insignificant content is contained. While the compression rate decreases the summary to be short, more information is lost. In fact, when the compression rate is 5-30%, the quality of summary is acceptable.

**Proposed system:-**

There are many methods to proceed with automatic text summarization. In our model we use an extractive technique to obtain the summary from the given text. This summary is then improved further by replacing a few parts of it using an abstractive technique. The extraction of sentences from the document is done keeping coherence in mind and therefore the summary maintains the essence of the original document. The sentences are then extracted using a Heap que algorithm , namely heapq and the final cluster or summary is formed.

**Technology used:-**

**NLTK(Natural language tool kit) :-**

The Natural Language Toolkit collection of open source Python modules involved with Natural Language Processing [5]. It is used for tokenizing and extracting important features from the text. It also does part-of-speech tagging, which is useful for identifying proper nouns and statistics, two features which are used in summarizing the text.

**GUI (Graphical user interface) :-**

GUI is nothing but a desktop app that provides you with an interface that helps you to interact with the computers and enriches your experience of giving a command (command-line input) to your code. They are used to perform different tasks in desktops, laptops, and other electronic devices, etc.

**Tkinter :-**

It is commonly comes bundled with Python, using Tk and is Python's standard GUI framework. It is famous for its simplicity and graphical user interface. It is open-source and available under the Python License.

**BS4 (beautiful soup) :-**

Beautiful Soup is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work.These instructions illustrate all major features of Beautiful Soup 4, with examples. I show you what the library is good for, how it works, how to use it, how to make it do what you want, and what to do when it violates your expectations.

**URL library :-**

Urllib module is the URL handling module for python. It is used to fetch URLs (Uniform Resource Locators). It uses the urlopen function and is able to fetch URLs using a variety of different protocols.

**Heapq** :-

Heap queue is a special tree structure in which each parent node is less than or equal to its child node. In python it is implemented using the heapq module. It is very useful is implementing priority queues where the queue item with higher weight is given more priority in processing.

**Stopwords** :- Stopwords are the English words which does not add much meaning to a sentence. They can safely be ignored without sacrificing the meaning of the sentence. For example, the words like the, he, have etc. Such words are already captured this in corpus named corpus.

**Python3:**

Python is an interpreted programming language. It is the primary language used in the project as algorithm and all of the NLP libraries are built to be used by Python. The specific version that was used is Python version 3.4.3.

**Machine Learning :-**

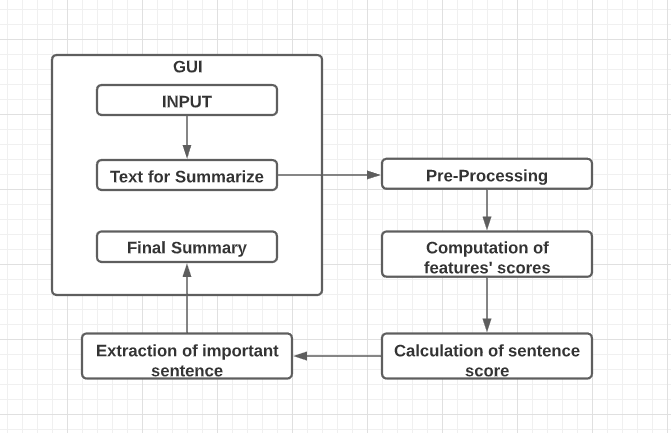
Machine Learning is a sub-area of artificial intelligence, whereby the term refers to the ability of IT systems to independently find solutions to problems by recognizing patterns in databases. In other words: Machine Learning enables IT systems to recognize patterns on the basis of existing algorithms and data sets and to develop adequate solution concepts.

**Methodology:-**

* Take the text as an input and tokenize it into sentences and words using ntlk.tokenize modules namely sent\_tokenize and word\_tokenize and filtering the words by removing the stopwords using nltk.corpus module. On sent\_tokenize the entered text gets split on a period (.) and sentences are obtained.
* These sentences are further operated on by word\_tokenize to obtain tokens in the form of words. Stopwords are the words likes articles, to be verbs (am, is are, was, were, etc) and also the punctuations of which if the frequency is calculated will just increase the complexity of the code. So these words are to be neglected as soon as the text is entered.
* The next step is calculating the frequency of words which belong to ‘word sent’ and not to set ‘a’ containing stopwords. The frequency calculated is stored using ‘collections’ module in Python.
* The further part is calculating the rank of each sentence. The frequency of each word in a sentence is integrated and a rank is given to each sentence and the sentences are sorted in descending order of the rank. This is done using sort method using heapq module in Python Language.
* The last part is displaying the summary. So the highest ‘n’ sentences are returned as summary of the entered text. The GUI (Graphical User Interface Is created In Python Using Tkinter module for working of this program).Here the input entered is the text and the number of statements in which summary is required. The output is the summary, number of statements in the entered text.

**Flow Chart :-**

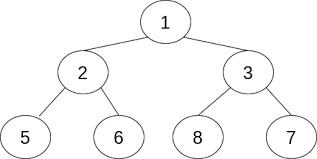
Automatic text summarization will work by using GUI application. We will give input in GUI then with the help of NLTK summarization this whole process will be done at end will get the final summary in Tkinter GUI application.



**Algorithm:-**

The algorithm which we have used is Heap queue algorithm which is also Known as priority queue algorithm. Heaps are arrays for which  a[k] <= a[2\*k+1]  and a[k] <= a[2\*k+2] for all k, counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that a[0] is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are k, not a[k]:



Here we have done with nlargest fuction which is one of the function of heapq To find the largest or sentence to score of text.

heapq.nlargest(n, iterable, key=None)

Return a list with the n largest elements from the dataset defined by iterable. key, if provided, specifies a function of one argument that is used to extract a comparison key from each element in iterable (for example, key=str.lower). Equivalent to: sorted(iterable, key=key, reverse=True)[:n].

In the tree above, each cell k is topping 2\*k+1 and 2\*k+2. In an usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell “wins” over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the “next” winner is to move some loser (let’s say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an O(n log n) sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not “better” than the last 0’th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the “win” condition means the smallest scheduled time. When an event schedules other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer .

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing “runs” (which are pre-sorted sequences, whose size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised . It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to achieve that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you’ll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0’th item on disk and get an input which may not fit in the current tournament (because the value “wins” over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run.

**Literature Survey – Paper I**

**A Summarization System for Scientific Documents :-**

While the science of automatic information retrieval (IR) has been around for quite a long time now basically, as long as databases of documents have been existing , the scientic community only quite recently has been moving to the eld of automatic information extraction (IE), maybe for as long as a decade now. There are at least two reasons for that: rst, the amount of on-line (and o-line) textual data has been increasing exponentially, without a limit in sight (keywords are electronic newswires, World Wide Web (WWW)) secondly, the eld of IE has been pushed forward by the ARPA funded evaluation series of the MUC conferences in the past ten years (see section 5.3). It has been demonstrated that fully automatic IE systems can be built with state-of-the-art technology, and that, for some selected tasks, their performance is as good as the performance of a human expert. Taking into account the enormous diference in throughput, machines can particularly serve as rough preprocessors of the Terabytes of information out there, which then can be further analyzed by human experts. For text summarization, the picture is a bit ambivalent: Early ideas and systems of automatically condensing and or summarizing documents date back into the fties (Luhn, 1958); there have been many different attempts to that goal, combining low level (e.g., statistical) and high level knowledge (e.g. domain models), with the rough conclusion that while it is quite feasible to produce a short indicative abstract or extract) from a text, it is still very hard to produce also a suciently coherent and readable" one if one aims at applying it to general domain real data’

**Observation:-**

The Main task before extractive text summarization is to find important information to be comprised in the summary. Extracted sentences are naturally longer than average it may sometime contain unessential information in the summary. Significant information is present in independent sections of the document, extractive summary sometimes may not catch all informative content proliferated across the document. Redundant information may included in the summarization Extraction based summaries are unappealing to read. There is a lack of flow in a summary text as extracted contents which are taken from different parts of the document leads to sudden topic shift**.**

**Literature Survey – Paper II   
Single Document Automatic Text Summarization using Term Frequency-Inverse Document Frequency (TF-IDF):-**

In the very beginning of the research in the arena of launching artificial intelligence to generate abridged version of a large document, disclosed the paradigms for extracting salient features. Automatic text summarization process model can be divided into three steps in the preprocessing step source text interpretation to source text representation, source representation transform to summary text representation with an algorithm and in the final step, summary text generation from summary representation. Numerous types of research work have been accomplished by various researchers where we can be familiar with multiple types of way of summary generation from single document text. In the following part of this section, methodologies of several researchers that incorporated this topic are depicted in brief, from pioneering works to the era of modern science where the thoughts of similar like human professionals’ abstraction techniques are being explored.

**Observation:-**

Increase efficiency of other researches to choose documents/information from search engines’ output, which usually contain an excess amount of replicated information. Solve the limitation of information presentation on small communication devices such as PDA and mobile phone etc., which is able to display abridged version of the full document. The running time of machine for translation is significantly reduced if a short version of text is given. Academics and researchers are very much benefitted by using automatic text summarization system as a tool to lessen the amount of time spent manually extracting the chief thoughts from large documents.

**Literature survey – Paper III**  
**ALGORITHM FOR TEXT TO GRAPH CONVERSION AND SUMMARIZING USING NLP:-**

Automatic text summarization is an important topic in the field of Natural Language Processing. It is the process of reducing a text document to a smaller format retaining all the key pieces of information that the document possesses. Formation of a coherent summary is a big challenge in the field of natural language processing. This paper aims to present an extraction based automatic text summarisation algorithm. The method constructs a weighted graph of the original text and by assuming each sentence to be a vertex. The weighted edge is determined by using a suitable distortion measure which analyses the semantic relation between two sentences. A ranking algorithm is used to compute the most important sentences in the text and that should be present in the summary. Using these techniques on a wide variety of data sets, Promising results were obtained that compared well to other successful models.

**Observation :-**

The summarizer is tested on different paragraphs with different compression factors. A fairly accurate summary was obtained in all the cases. The summaries contained most of the important sentences that were essential to the original text. The algorithm implemented performs well in comparison to other successful models used in summarization.

**NLP**

**Introduction to Natural language processing:-**

In reality, plain text is the most predominant form of data available today. Text analysis applies analysis of word frequency distributions, pattern recognition, tagging, link and association analysis, sentiment analysis, and visualization. Natural Language Processing is a broad topic, Machine Translation, Summarizing texts, spam detection, sentiment analysis are real big field. Python is readable, fast for prototypes, it has rich library for reading and writing data, running calculations on the information and creating graphical representations of data sets and list support, it includes a lot of NLP-related libraries viz. **NLTK, textblob, scipy, pandas**... also it has great parsing libraries viz. **Beautifulsoup, scrapy**.

Natural Language Toolkit (NLTK) is a Python API for the analysis of texts written in natural languages, such as English. NLTK is a very popular and old library which comes with a collection of sample texts called corpora (collection of text documents).

NLTK is a leading platform for building Python programs to work with human language data (Natural Language Processing).

NLTK is intended to support research and teaching in NLP or closely related areas, including empirical linguistics, cognitive science, artificial intelligence, information retrieval, and machine learning.

**What is Natural language processing:-**

Natural language processing (NLP) is a field of computer science, artificial intelligence, and computational linguistics concerned with the interactions between computers and human (natural) languages. As such, NLP is related to the area of human–computer interaction. Many challenges in NLP involve: natural language understanding, enabling computers to derive meaning from human or natural language input; and others involve natural language generation.

The first thing that jumps out at me about this explanation is the fact that Wikipedia feels the need to qualify “languages” as human or natural. Aren’t all languages human or natural? Well, it tuns out that within computer science there are artificial or machine languages, those that mediate between what a human programmer wants a computer to do and what the computer actually does. In this course, you are going to learn one of those programming languages, called Python.

So there are human or natural languages like English and machine or artificial languages like Python. Let me give you a short example of each. Here is a sentence of English, “If it is 3:50 pm, write ‘It’s time to go.’”

**Text Preprocessing**

**Accessing data:-**

Well you need to have an access to your data before starting to process it. You can access and open your data file in many ways.

#### Opening a local file

When you have the data file on your local machine, you can access you file using ***open()*** method and can read into the lines using ***.readline()***. Don't forget to mention 'r' mode while opening a file.

f=open('text.txt', 'r')

f.readline()

#### Opening file from other online source You might have to process data/text which is available online, then you can fetch the required data like this:-

importrequests

f\_online='http://www.gutenberg.org/files/11111/11111.txt'

f\_rawtext=requests.get(f\_online).text

print (f\_rawtext)

#### Opening file from corpus As mentioned earlier that nltk comes with a collection of sample texts called corpora (collection of text.

You can install nltk data as follows:-

importnltk

nltk.download()

Once you execute the above line of codes, a new window will appear which is a NLTK Downloader. You can download the entire collection by using all, or just the data required for your project/assignment. .

Here you can open Gutenberg from nltk.corpus and create sample text from 'bible-kjv.txt' file from Gutenberg.

From nltk.corpusimportgutenberg

textguten=gutenberg.raw("bible-kjv.txt")

**Text searching:-**

Before getting into the ways of searching text from text files let's import some text.

From nltk.corpusimportgutenberg

textguten=gutenberg.raw("bible-kjv.txt")

You can see that object textguten carries a string. You cannot apply 'Text' funstions on a string. You need to import **Text** module from *nltk.text* and you need to fetch words from gutenberg first than raw string.

From nltk.textimport Text

From nltk.corpusimportgutenberg

textguten=gutenberg.words("bible-kjv.txt")

text=Text(textguten)

However if you run this code:-

From nltk.bookimport\*

***syntax:-* text\_file\_name.concordance("word\_to\_be\_searched")**

**You have fetched text from Gutenberg and converted that string into *Text*. This text is stored in the variable 'text'. You can now try text searching on this.**

Concordance produces all occurrences of a specified word along some context. Using concordance you can search for the incidents where word 'foundation' appears in the text file text1.Concordance allows you to spot lexis (words) in context.

text.concordance("God")

Note that it also specifies the total number of occurrences of the specified word ['Displaying 25 of 4472 matches' as for your word 'God'] at the start.

What if you want to search for words similar to a given word in the text?

**syntax:- text\_file\_name.similar("word\_to\_be\_searched")**

You can try finding similar words of a given word in your 'text'.

text.similar("God")

text.similar("monster")

text.similar("comit")

text.similar("said")

The command .similar() displays words which appear in a similar range of contexts. See output for commands, where similar("monster") and similr("comit").... no match is found!.

\*Note that we get different results for different texts even for the same word.\*

What if you want to look for the contexts which are common for two words? .common\_contexts() allows you to do so.

syntax:- name\_of\_the\_file.common\_contexts(["word1", "word1"])

text.common\_contexts(["God", "thee"])

Usage of collocations is frequent in business and work settings. A collocation is a series of words that happen to appear together often oddly. Like blue sky, feel free, close a deal are collocations. This is how you can look for collocations inside the text:-

**syntax:- name\_of\_the\_file.collocations()**

text.collocations()

Collocation helps you find **bigrams** that occur more often than you would expect based on the frequency of the individual words

**Tokenization:-**

Tokenizing means splitting your document into lexical chunks.

Tokenizers are used to **break text strings** and to find words and punctuation in a string.

Tokenizing can be done at word level and at sentence level. NLTK has a few tokenizers packages. The following sample code requires Punkt sentence tokenization models to be installed.

You need a paragraph kind of text to undersand what tokenization does. Well, at the start of this tutorial, you fetched raw data from the same Gutenberg. You can make use of that **textguten**!

textguten=gutenberg.raw("bible-kjv.txt")

fromnltk.tokenizeimportsent\_tokenize, word\_tokenize

print(sent\_tokenize(textguten[:1000]))#considering only first 1000 alphabets from textguten

print(word\_tokenize(textguten[:1000]))

**Stop Words:-**

These are common terms which rarely add to the meaning of a sentence for the purposes of NLP and information retrieval. These words are recurring many a times as part of speech and need not be considered for classification.

In textguten[:1000], you can see, that you get following tokens:-

['**[**', '**The**', 'King', 'James', 'Bible', ']', 'The', 'Old', 'Testament', 'of', 'the', 'King', 'James', 'Bible', 'The', 'First', 'Book', '**of**', 'Moses', ':', 'Called', 'Genesis', '**1:1**', 'In', 'the', 'beginning', 'God', 'created', 'the', 'heaven', 'and', 'the', 'earth', '.', '**1:2**', 'And', 'the', 'earth', 'was', 'without', 'form', ',', 'and', 'void', ';', '**and**', 'darkness', 'was', 'upon', 'the', 'face', 'of', 'the', 'deep', '**.**', 'And', 'the', 'Spirit', 'of', 'God', 'moved', 'upon', 'the', 'face', 'of', 'the', 'waters', '.', '**1:3**', 'And', 'God', 'said', '**,**', 'Let', 'there', 'be', 'light', ':', 'and', 'there', 'was', 'light', '.', '**1:4**', 'And', 'God', 'saw', 'the', 'light', ',', 'that', 'it', 'was', 'good', ':', 'and', 'God', 'divided', 'the', 'light', 'from', 'the', 'darkness', '.', '1:5', 'And', 'God', 'called', 'the', 'light', 'Day', ',', 'and', 'the', 'darkness', 'he', 'called', 'Night', '.', 'And', 'the', 'evening', 'and', 'the', 'morning', 'were', 'the', 'first', 'day', '.', '1:6', 'And', 'God', 'said', ',', 'Let', 'there', 'be', 'a', 'firmament', 'in', 'the', 'midst', 'of', 'the', 'waters', ',', 'and', 'let', 'it', 'divide', 'the', 'waters', 'from', 'the', 'waters', '.', '1:7', 'And', 'God', 'made', 'the', 'firmament', ',', 'and', 'divided', 'the', 'waters', 'which', 'were', 'under', 'the', 'firmament', 'from', 'the', 'waters', 'which', 'were', 'above', 'the', 'firmament', ':', 'and', 'it', 'was', 'so', '.', '1:8', 'And', 'God', 'called', 'the', 'firmament', 'Heaven', '.', 'And', 'the', 'evening', 'and', 'the', 'morning', 'were', 'the', 'second', 'day', '.', '1:9', 'And', 'God', 'said', ',', 'Let', 'the', 'waters', 'under', 'the', 'heav']

you can create a list of stopwords (like the ones which are **bold**) from the above tokens, as follows:- You can eliminate stop words from the textguten either using **list comprehensioon** or by implementing **for loop**.

stop=['[',']','The','of','1:1',':','.','1:2',',','1:3',':','1:4','1:5','1:6','it','1:7','In','1:9','1:8','so','Let']

words=word\_tokenize(textguten[:1000])

clean\_textguten= [i for i in words ifnot i in stop]#using list comprehension

#using for loop

clean\_textguten= []

for i in words:

if i notin stop:

clean\_textguten.append(i)

print(clean\_textguten)

After this, you have got clean text (clean\_textguten)!!

See the output does not contain the tokens which were present in the list of stopwords.

Try the following code! Another way of creating stopwords is by using stopwords module from nltk.corpus as follows:-

fromnltk.corpusimportstopwords

stop=set(stopwords.words('English'))

Here, you are creating a set of 'English' stop words. Well if you are working on say French text, you can create French stopwords as well.

**Lexicon Normalization:-**

To perform normalization that mainly involves eliminating punctuation, converting the entire text into lowercase or uppercase, converting numbers into words, expanding abbreviations, canonicalization of text, and so on. As text is fetched in the form of a string, .lower() and .upper() can be applied on it for converting text into lowercase or uppercase. The commonly used normalization methods are stemming and lemmatization.

**Stemming:-**

Words need to be stemmed to retrieve their radicals, so that various forms derived from a stem would be taken as the same when counting word frequency. Stemming is used to remove morphological affixes from words, leaving only the word stem. For instance, words “update", “updated" and “updating" would all be stemmed to “update".

For stemming, you need to import some stemmer from *nltk.stem module*. Then create an object for that stemmer. And then you can find the stems for all the words in the clean text using for loop.

fromnltk.stemimportPorterStemmer

ps=PorterStemmer()

stemtext=[]

for i inclean\_textguten:

p=ps.stem(i)

stemtext.append(p)

print(stemtext)

**Lemmatization:-**

This operation is similar to stemming. Let's lemmatize a few words. First you need to import a lemmatizer. You can use WordNetLemmatizer from nltk.stem package. Then create an instance of that lemmatizer and then use .lemmatize() for getting the stems of words, as follows:-

</dl>

fromnltk.stemimportWordNetLemmatizer

lr=WordNetLemmatizer()

lemtext=[]

for i inclean\_textguten:

p=lr.lemmatize(i)

lemtext.append(p)

print(lemtext)

Here, you are importing WordNetLemmatizer, then **lr** object is created for this lemmatizer. And you are creating a list of lemmatized word (*lemtext*) for the words present in \_clean*textguten*.

Note that it returns the input word unchanged if it cannot be found in WordNet.

**Text to Features (Feature Engineering on text data)**

**Part of Speech Tagging:-**

This process converts tokenizes words into a list of containing word and its corresponding POS. A part-of-speech tag signifies whether the word is a noun, adjective, verb, and so on.

You can start with tagging POS to your text!

First of all, you need to tokenize the string in to words using word\_tokenize, (if required) and then you need create a list called 'words' of the tokens so created.

Using pos\_tag from nltk, you can tag tokens to their respective POS. Well, Later you can print those tags.

By now, you have your clean text (clean\_textguten), which is already in the form of tokens, so you will not require to tokenize it!

tagged\_textguten=nltk.pos\_tag(clean\_textguten)

print(tagged\_textguten[:2])#printing results for only first two instances!

It is clear from the output that 'the' is determiner as indicated by DT and 'Project' is 'Noun, Proper, Singular' as indicated by NNP. Note that tagger creates a list of tuple containing word and its corresponding POS.

**Chunking:-**

Chunks are the minimal units that can be processed or we can say that chunks are the non-overlapping linguistic groups. Chunking is a kind of shallow parsing. Chunking is not possible to do until we have done tagging.</dl>You need to define a chunk grammar i.e. You need to create a rule (regular expression based on tags) for the entity to be extracted. To do this, you're going to use regular expressions. A regular expression is a set of characters used to define a search pattern. Let's say you want to search for words begning with 'a', you need to create a regex which will be 'a\w+' in your case. Where 'a' at the start of regex means will only match words that begin with 'a'; '\w' is a special character and it will match any alphanumeric A-z, a-z, 0-9, along with underscores;'+' means the regex can appear many times in strings.

for example:- grammar = "NP: {\<DT>\?

= match 1 or more ? = match 0 or 1 repetitions.

= match 0 or MORE repetitions  
. = Any character except a new line

for i inclean\_textguten[:2]:

chunkGram=r"""Chunk: {<RB.?>\*<VB.?>\*<NNP>+<NN>?}"""

chunkParser=nltk.RegexpParser(chunkGram)

chunked=chunkParser.parse(tagged)

chunked.draw()

print(chunked)

Note that the chunkgram n the above example code means:-<RB.?>*= \"0 or more of any tense of adverb, <VB.?>* = \"0 or more of any tense of verb,

**Named Entity Recognition:-**

Named entity is a real-world object, such as persons, locations, organizations, products, etc., that can be denoted with a proper name (https://en.wikipedia.org/wiki/Named\_entity). Named entity recognition is a way of extracting most common entities such as people, places, things, locations, monetary figures, and more. With named entity recognition you can easily locate proper names of people, organizations, locations, or other entities from the text. After tagging you can use \_\_nltk.ne\_chunk()\_\_, nltk provides a classifier trained to recognize named entities, accessed with the function nltk.ne\_chunk().

</dl>

for i inclean\_text[:2]:

ner=nltk.ne\_chunk(tagged, binary=True)

ner.draw()

Note that if you set binary=True, then named entities are just tagged as NE; otherwise, the classifier adds category labels such as Person, Organization, and GPE.

for i inclean\_text[:2]:

ner=nltk.ne\_chunk(tagged)

ner.draw()

**TF – IDF :-**

A term-document matrix represents the relationship between terms and documents, where each row stands for a term or words as present in the data sheet and each column for a document, and an entry is the number of occurrences of the term in the document.

</dl>Given a word, a document and a collection of documents, Vocabulary of the document is the number of unique/different words that document contains. Or we can say that The vocabulary of a text is just the set of tokens that it uses.

Term Frequency is the count of words in the document (vocab) / length of the document.

Inverse Document Frequency is the log of length of collection / count of document\_containing\_term in the collection.

**Term Frequency – Inverse Document Frequency = tf \* idf**

Note nltk does not have anything to find *Term Frequency – Inverse Document Frequency* directly. However you can do so using **len()** method. This is explained under the head of Lexical richness.

**Lexical richness:-**

Lexical richness means number of the number of distinct words in the document.

Function for finding lexical richness of the text:-

deflexical\_richness(text):

returnlen(set(text)) /len(text)

Function for finding percentage of the text is taken up by a specific word:-

defpercentage(word, text):

return (100\*text.count(word) /len(text))

Now you can call both functions for your text textguten

print ('Lexical richness of the text: '+str(lexical\_richness(textguten)))

print ('Percentage: '+str(percentage('only',(textguten))))

*Using****math.log()****, you can easily find the IDF*

**Frequency Distribution:-**

Frequency distribution notify about the frequency of each vocabulary item in the text.

Using following code you can find the frequency distribution of *textguten*. You need to import *FreqDist* beforehand from *nltk*.

fromnltkimportFreqDist

fdist\_textguten=FreqDist(textguten)

Note that output is a dictionary containing every vocabulary item with the value of its occurrence. You can also display this dictionary using print() command.

~python

print(fdist\_textguten)

You can also find most common 10 vocab items of this text.

fdist\_textguten.most\_common(10)

Well, you can also plot frequency distribution graph. If you wish to plot frequency distribution, try following code, which plots frequency distribution of 10 most common vocab items.

fdist\_textguten.plot(10, cumulative=True)

**Project Source Code:-**

# Core Packages

import tkinter as tk

from tkinter import \*

from tkinter import ttk

from tkinter.scrolledtext import \*

import tkinter.filedialog

from tkinter.scrolledtext import\*

import time

timestr = time.strftime("%Y%m%d-%H%M%S")

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize, sent\_tokenize

import heapq

def nltk\_summarizer(raw\_text):

stopWords = set(stopwords.words("english"))

word\_frequencies = {}

for word in nltk.word\_tokenize(raw\_text):

if word not in stopWords:

if word not in word\_frequencies.keys():

word\_frequencies[word] = 1

else:

word\_frequencies[word] += 1

maximum\_frequncy = max(word\_frequencies.values())

for word in word\_frequencies.keys():

word\_frequencies[word] = (word\_frequencies[word]/maximum\_frequncy)

sentence\_list = nltk.sent\_tokenize(raw\_text)

sentence\_scores = {}

for sent in sentence\_list:

for word in nltk.word\_tokenize(sent.lower()):

if word in word\_frequencies.keys():

if len(sent.split(' ')) < 30:

if sent not in sentence\_scores.keys():

sentence\_scores[sent] = word\_frequencies[word]

else:

sentence\_scores[sent] += word\_frequencies[word]

summary\_sentences = heapq.nlargest(7, sentence\_scores, key=sentence\_scores.get)

summary = ' '.join(summary\_sentences)

return summary

# Web Scraping Pkg

from bs4 import BeautifulSoup

from urllib.request import urlopen

# Structure and Layout

window = Tk()

window.title("Summaryzer GUI")

window.geometry("700x400")

window.config(background='black')

style = ttk.Style(window)

style.configure('lefttab.TNotebook', tabposition='wn',)

# TAB LAYOUT

tab\_control = ttk.Notebook(window,style='lefttab.TNotebook')

tab1 = ttk.Frame(tab\_control)

tab2 = ttk.Frame(tab\_control)

tab3 = ttk.Frame(tab\_control)

tab4 = ttk.Frame(tab\_control)

tab5 = ttk.Frame(tab\_control)

# ADD TABS TO NOTEBOOK

tab\_control.add(tab1, text=f'{"Home":^20s}')

tab\_control.add(tab2, text=f'{"File":^20s}')

tab\_control.add(tab3, text=f'{"URL":^20s}')

tab\_control.add(tab4, text=f'{"Comparer ":^20s}')

tab\_control.add(tab5, text=f'{"About ":^20s}')

label1 = Label(tab1, text= 'Summaryzer',padx=5, pady=5)

label1.grid(column=0, row=0)

label2 = Label(tab2, text= 'File Processing',padx=5, pady=5)

label2.grid(column=0, row=0)

label3 = Label(tab3, text= 'URL',padx=5, pady=5)

label3.grid(column=0, row=0)

label3 = Label(tab4, text= 'Compare Summarizers',padx=5, pady=5)

label3.grid(column=0, row=0)

label4 = Label(tab5, text= 'About',padx=5, pady=5)

label4.grid(column=0, row=0)

tab\_control.pack(expand=1, fill='both')

# Clear entry widget

def clear\_text():

entry.delete('1.0',END)

def clear\_display\_result():

tab1\_display.delete('1.0',END)

def get\_summary():

raw\_text = entry.get('1.0',tk.END)

final\_text = nltk\_summarizer(raw\_text)

print(final\_text)

result = '\nSummary: {}'.format(final\_text)

tab1\_display.insert(tk.END,result)

# Clear Text with position 1.0

def clear\_text\_file():

displayed\_file.delete('1.0',END)

# Clear Result of Functions

def clear\_text\_result():

tab2\_display\_text.delete('1.0',END)

# Clear For URL

def clear\_url\_entry():

url\_entry.delete(0,END)

def clear\_url\_display():

tab3\_display\_text.delete('1.0',END)

# Clear entry widget

def clear\_compare\_text():

entry1.delete('1.0',END)

def clear\_compare\_display\_result():

tab1\_display.delete('1.0',END)

# Functions for TAB 2 FILE PROCESSER

# Open File to Read and Process

def openfiles():

file1 = tkinter.filedialog.askopenfilename(filetypes=(("Text Files",".txt"),("All files","\*")))

read\_text = open(file1).read()

displayed\_file.insert(tk.END,read\_text)

def get\_file\_summary():

raw\_text = displayed\_file.get('1.0',tk.END)

final\_text = nltk\_summarizer(raw\_text)

result = '\nSummary:{}'.format(final\_text)

tab2\_display\_text.insert(tk.END,result)

# Fetch Text From Url

def get\_text():

raw\_text = str(url\_entry.get())

page = urlopen(raw\_text)

soup = BeautifulSoup(page)

fetched\_text = ' '.join(map(lambda p:p.text,soup.find\_all('p')))

url\_display.insert(tk.END,fetched\_text)

def get\_url\_summary():

raw\_text = url\_display.get('1.0',tk.END)

final\_text = nltk\_summarizer(raw\_text)

result = '\nSummary:{}'.format(final\_text)

tab3\_display\_text.insert(tk.END,result)

# COMPARER FUNCTIONS

def use\_nltk():

raw\_text = str(entry1.get('1.0',tk.END))

final\_text = nltk\_summarizer(raw\_text)

print(final\_text)

result = '\nNLTK Summary:{}\n'.format(final\_text)

tab4\_display.insert(tk.END,result)

# MAIN NLP TAB

l1=Label(tab1,text="Enter Text To Summarize")

l1.grid(row=1,column=0)

entry= ScrolledText(tab1,height=10)

entry.grid(row=2,column=0,columnspan=2,padx=5,pady=5)

# BUTTONS

button1=Button(tab1,text="Reset",command=clear\_text, width=12,bg='#03A9F4',fg='#fff')

button1.grid(row=4,column=0,padx=10,pady=10)

button2=Button(tab1,text="Summarize",command=get\_summary, width=12,bg='#ced',fg='#fff')

button2.grid(row=4,column=1,padx=10,pady=10)

button3=Button(tab1,text="Clear Result", command=clear\_display\_result,width=12,bg='#03A9F4',fg='#fff')

button3.grid(row=5,column=0,padx=10,pady=10)

button4=Button(tab1,text="Main Points", width=12,bg='#03A9F4',fg='#fff')

button4.grid(row=5,column=1,padx=10,pady=10)

# Display Screen For Result

tab1\_display = Text(tab1)

tab1\_display.grid(row=7,column=0, columnspan=3,padx=5,pady=5)

#FILE PROCESSING TAB

l1=Label(tab2,text="Open File To Summarize")

l1.grid(row=1,column=1)

displayed\_file = ScrolledText(tab2,height=7)# Initial was Text(tab2)

displayed\_file.grid(row=2,column=0, columnspan=3,padx=5,pady=3)

# BUTTONS FOR SECOND TAB/FILE READING TAB

b0=Button(tab2,text="Open File", width=12,command=openfiles,bg='#c5cae9')

b0.grid(row=3,column=0,padx=10,pady=10)

b1=Button(tab2,text="Reset ", width=12,command=clear\_text\_file,bg="#b9f6ca")

b1.grid(row=3,column=1,padx=10,pady=10)

b2=Button(tab2,text="Summarize", width=12,command=get\_file\_summary,bg='blue',fg='#fff')

b2.grid(row=3,column=2,padx=10,pady=10)

b3=Button(tab2,text="Clear Result", width=12,command=clear\_text\_result)

b3.grid(row=5,column=1,padx=10,pady=10)

b4=Button(tab2,text="Close", width=12,command=window.destroy)

b4.grid(row=5,column=2,padx=10,pady=10)

# Display Screen

# tab2\_display\_text = Text(tab2)

tab2\_display\_text = ScrolledText(tab2,height=10)

tab2\_display\_text.grid(row=7,column=0, columnspan=3,padx=5,pady=5)

# Allows you to edit

tab2\_display\_text.config(state=NORMAL)

# URL TAB

l1=Label(tab3,text="Enter URL To Summarize")

l1.grid(row=1,column=0)

raw\_entry=StringVar()

url\_entry=Entry(tab3,textvariable=raw\_entry,width=50)

url\_entry.grid(row=1,column=1)

# BUTTONS

button1=Button(tab3,text="Reset",command=clear\_url\_entry, width=12,bg='#03A9F4',fg='#fff')

button1.grid(row=4,column=0,padx=10,pady=10)

button2=Button(tab3,text="Get Text",command=get\_text, width=12,bg='#03A9F4',fg='#fff')

button2.grid(row=4,column=1,padx=10,pady=10)

button3=Button(tab3,text="Clear Result", command=clear\_url\_display,width=12,bg='#03A9F4',fg='#fff')

button3.grid(row=5,column=0,padx=10,pady=10)

button4=Button(tab3,text="Summarize",command=get\_url\_summary, width=12,bg='#03A9F4',fg='#fff')

button4.grid(row=5,column=1,padx=10,pady=10)

# Display Screen For Result

url\_display = ScrolledText(tab3,height=10)

url\_display.grid(row=7,column=0, columnspan=3,padx=5,pady=5)

tab3\_display\_text = ScrolledText(tab3,height=10)

tab3\_display\_text.grid(row=10,column=0, columnspan=3,padx=5,pady=5)

# COMPARER TAB

l1=Label(tab4,text="Enter Text To Summarize")

l1.grid(row=1,column=0)

entry1=ScrolledText(tab4,height=10)

entry1.grid(row=2,column=0,columnspan=3,padx=5,pady=3)

# Display Screen For Result

tab4\_display = ScrolledText(tab4,height=15)

tab4\_display.grid(row=7,column=0, columnspan=3,padx=5,pady=5)

# About TAB

about\_label = Label(tab5,text="Summaryzer GUI V.0.0.1 \n Sai kotturu",pady=5,padx=5)

about\_label.grid(column=0,row=1)

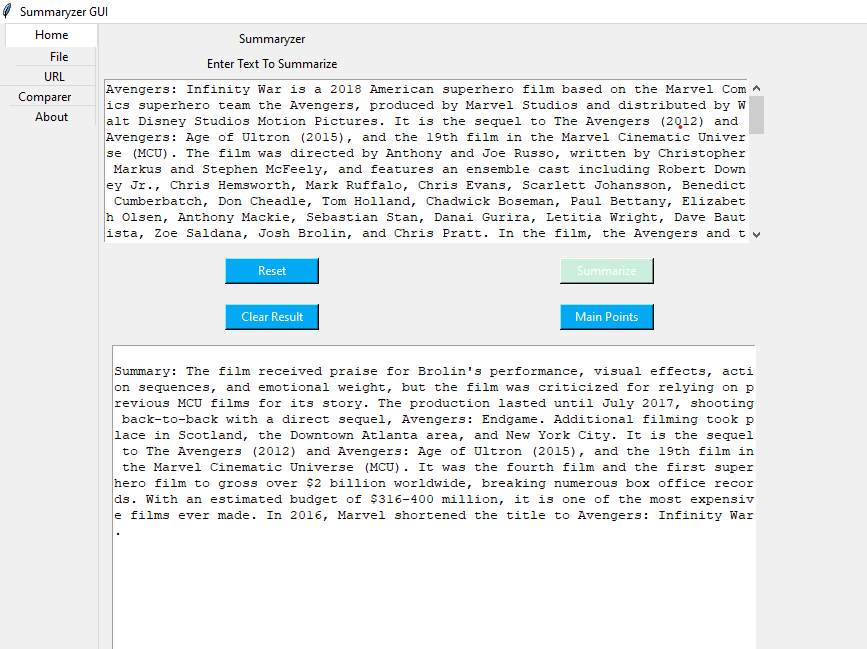
window.mainloop()

**Result :-**

**OUTPUT 1 :-**

Here we can summarize the text by giving the text directly into text box.

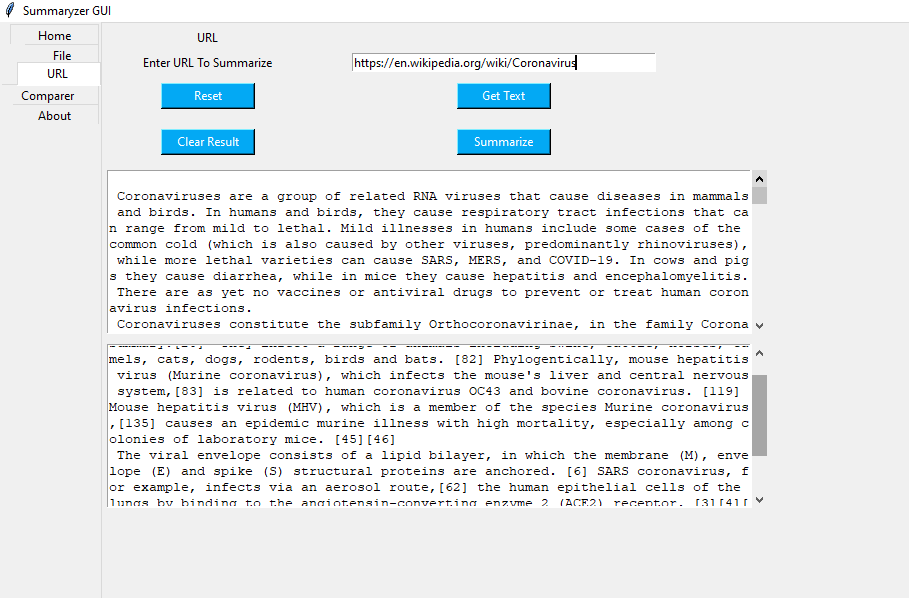
First box is text box and the down box summarized text box. After giving the text we have to click the **Summarize button** which is in the **GUI** application and we also can reset the text by clicking the **Reset button and** we can clear summarized text also by clicking the **Clear Result button in GUI application.** Here summarization done with the help of **NLTK summarization.**



**OUTPUT 2 :-**

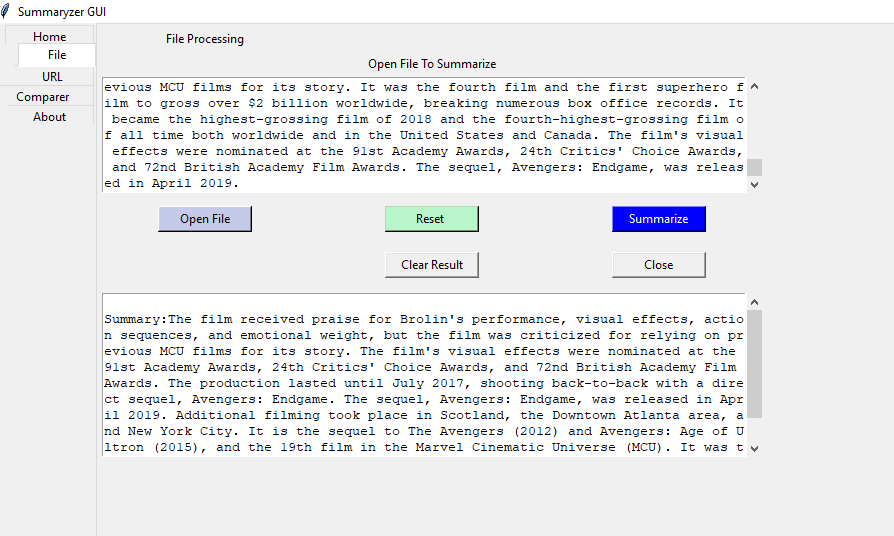
Here we fetching the text data from wikipidea through URL library with the help of NLTK summarization.

First we have to enter the URL then we have to click **Get Text button** then we will get full text which is in wikipidea after that we have click **Summarize button** which is **GUI** to get the summarized text. Here summarization done with the help of **NLTK summarization**



**OUTPUT 3 :-**

Here we have to browse the text file by clicking **Open File button,** after selecting the text file we will get full text wich is in the text document. Then we have to click **Summarize button** which is in the **GUI** application. Here summarization done with the help of **NLTK summarization**

****

**Coclusion :-**

Due to World Wide Web, the rate of information growth has called for a need to develop efficient techniques to reduce data and make it simpler to understand and convey messages effectively. Natural Language Processing is thoroughly being used worldwide for its efficiency in text processing. It’s delicate to analyze human interpretation using various principles related to NLP.

In the mean way we just made the automatic text summarization very inovative we just have to provide url link , text file, or text to summarize in easy and at a time. We can summarize the text in three ways in Tkinter GUI Application. These application were focused on text analysis and thereby reducing time to get the main gist of the input processing output. Our algorithm has proved to perform well for most summarization purposes. The current extractive summary is advantageous for certain formats of documents.

**Future Scope:-**

Today, most of the approaches for summarizing are based on extraction. Hence the use of abstraction in text processing which includes automated text building in response to the input can be further implemented. Various domains for research in wide spectrum of text to gain accuracy using Python are helpful using frameworks like Django and Flask. Text to graph convertors can be used for comparison of datasets having different time domains. For handy purposes, these applications can also be converted into a mobile application in Android or iOS. Databases can be used for handling multiple datasets and thereby reducing the memory as well as time for retrieval.