

Ministry of high education,
Culture and science city at Oct6,
High institute of computer Science & information systems



المعهد العالي لعلوم الحاسب ونظم المعلومات

Graduation Project:

Text summarization Desktop Application

Prepared by

41068 - مصطفى ربيع شحاته عبد العزيز

41001 - إبراهيم عبد الحميد عبد القاصد

41071 - مصطفى محمد مصطفى سعيد

41003 - أحمد أحمد هاشم عبد الفتاح

41073 - ميرنا منير شحاته حنس

41046 - كريم حسن سالم محمد مدنى

Supervised by

Prof. Dr. Araby Kashik

Assistant: Eng. Ahmed Tammam

Project No: 14

Academic Year: 2020/2021

Acknowledgement

Great thanks of Allah at the beginning of anything and being great full for completion of our project. Thanks to the management of the Higher Institute of Computer Science and Information Systems for their outstanding efforts in providing students with the convenience of establishing laboratories and halls in a good way and ensuring cleanliness and providing a complete support with professors to provide the greatest degree of excellence and success. Great thanks to the Dean of the Higher Institute of Computer Science and Information Systems. **Prof. Dr. Sami Abdel Moneim** for his wonderful efforts in helping the student and supporting them and listening to our complaint and then resolving immediately and the initiative to improve the level of the student in all possible ways. Thanks to **Dr.Araby Kashik** for his great efforts in supervising, supporting and guiding the project. We offer our thanks and gratitude to the **Eng.Ahmed Tammam** for his special supervision of the project and his great efforts in directing us to make the project the best form possible. Finally, we cannot forget the full support of our family and our parents and your sincere love and appreciation to the students of the project.

Project Objectives & Summery

The main goal of the project is to convert a large text into a small text, and this text contains the important parts of the main text without losing any part of it, and the program provides many ways to get the text, whether it is an article, website, file, portable document format or About copying text from anywhere to the program.

This application works in two ways, the first method is extractive summarization, and the second method is abstractive summarization. The first method contains three algorithms and this method works by machine learning, the first algorithm calls spacy, the second algorithm calls Gensim, the third algorithm calls NLTK. The second method works via deep learning.

In extractive Summarizing We identify important sentences or phrases from the original text and extract only them from the text. These extracted sentences will be our summary. As for, in abstractive summarizing, here, we generate new sentences from the original text. Sentences generated through abstract summation may not be present in the original text.

Abstract summarization works in a better way and obtains a result more efficiently than extract summarization

Finally, this app summarizes texts with high accuracy

Table of Contents

Graduation Project.....	1
Acknowledgement.....	2
Project Objectives & Summary.....	3
CHAPTER 1: Introduction.....	7
Background.....	8
An overview of the project.....	10
Reasons why we need text summarization.....	11
Types of text summarization.....	12
Benefits.....	15
Use cases in the enterprise.....	17
Project Tools.....	21
CHAPTER 2: THEORETICAL BACKGROUND AND TOOLS.....	22
SYSTEM ANALYSIS.....	23
USE CASE for Text Summarization.....	25
ACTIVITY DIAGRAM for Text Summarization.....	26
TOOLS.....	27
GUI TOOLS.....	27
Programming TOOLS.....	28
CHAPTER 3: Project implementation.....	31
Phase1: Extractive sector.....	32
Phase2: Abstractive sector.....	35
Measurement for Extractive and Abstractive summarization.....	36
The GUI of Text summarization application.....	38

Appendices.....	45
IMPLEMENTATION.....	46
CHAPTER 4: Conclusion.....	60
Conclusion.....	61
References.....	62
ملخص باللغة العربيه.....	63

Table of Figures

Fig.1: USECASE DIAGRAM.....	25
Fig.2: ACTIVITY DIAGRAM.....	26
Fig.3: Dash Board.....	38
Fig.4: sPaCy summarize.....	39
Fig.5: NLTK summarize.....	40
Fig.6: Gensim summarize.....	41
Fig.7: Abstract summarize.....	42
Fig.8: Accuracy for Extractive summarize.....	43
Fig.9: Accuracy for Abstract summarize.....	44
Fig.10: sPaCy code.....	46
Fig.11: NLTK code.....	47
Fig.12.1: Gensim code.....	48
Fig.12.2: Gensim code.....	49
Fig.12.3: Gensim code.....	50
Fig.12.4: Gensim code.....	51
Fig.13: Abstractive code.....	52
Fig.14.1: Main function code.....	53
Fig.14.2: Main function code.....	54
Fig.14.3: Main function code.....	55
Fig.14.4: Main function code.....	56
Fig.14.5: Main function code.....	57
Fig.14.6: Main function code.....	58
Fig.14.7: Main function code.....	59

CHAPTER 1: Introduction

Background:

When you open up news sites, do you just start reading every news article? Mostly not. We usually look at the short news summary and then read more details if you're interested. Short and useful summaries of news are now available everywhere like magazines, news aggregator apps, search sites, etc. There is an enormous amount of textual material, and it is growing every single day. Think of the internet, comprised of web pages, news articles, status updates, blogs and so much more. The data is unstructured and the best that we can do to navigate it is to use search and skim the results.

There is a great need to reduce much of this text data to shorter, focused summaries that capture the salient details, both so we can navigate it more effectively as well as check whether the larger documents contain the information that we are looking for.

This topic provides an explanation of the methods and types of summarization and to what extent the text becomes after summarizing, how to benefit from it and the mechanism of the work of summarization, where summarizing the text, is the reduction of the text to its basic content, is a very complex problem, despite the progress made in this field so far, it poses many challenges to the scientific community. It is also a convenient application in today's information society due to the exponential growth of textual information over the Internet and the need to immediately evaluate the contents of text collections.

Text summarization has been an active area of research for many years. Several approaches have been proposed, ranging from simple methods of position and word frequency, to learning and algorithms based on text summarization. The emergence of human-generated knowledge bases provides another possibility of

text summarization - it can be used to understand the input text in terms of concepts salient from the knowledge base. In this paper, we examine a new approach that benefits in conjunction with summarizing large texts. In this topic, we will rely on more than one model, each model has its own approach, and derive the convergence properties under each model. Then, we adopt a custom query-focused summarization, in which sentence order is additionally dependent on user interests and queries, respectively. We will introduce a multi-document summarization algorithm. An important feature of the proposed algorithms is that they allow incremental summarization in real time - users can view an initial text, then the user chooses his preferred approach, and the results show that taking advantage of many different approaches can significantly improve the quality of the summary. This suggests that using increased summarization can help to better understand news articles.

Text summarization is the problem of creating a short, accurate, and fluent summary of a longer text document. Text summarization methods are greatly needed to address the ever-growing amount of text data available online to both better help discover relevant information and to consume relevant information faster.



After reading this post, you will know:

- (1) Why text summarization is important, especially given the wealth of text available on the internet.
- (2) Examples of text summarization you may encounter every single day.
- (3) The application of deep learning methods for abstractive text summarization.
- (4) The application of machine learning methods for extractive text summarization.
- (5) A method for selecting the main content of the document.

An overview of the project:

Now the world uses technology in all activities and in all practical and scientific fields. Hence the idea of creating an app for summarizing large texts. The tools used to support the application's functioning cover a wide range of different technologies, and the best path to a summary of large texts is in most areas.

Text summarization is an important Natural Language Processing (NLP) task that is bound to have a considerable impact on our lives. With the evolution of the digital world and its consequent impact on the growing publishing industry, dedicating time to sincerely read an article, document, or book to decide its relevance is no longer a feasible option, especially considering time scarcity.

Further, with the increasing number of articles being published accompanied by the digitization of traditional print publications, it has become nearly impossible to keep track of incrementing publications available on the web. This is where text summarization can help to reduce the texts with less time intricacy.

Text identification, interpretation and summary generation, and analysis of the generated summary are some of the key challenges faced in the process of text summarization. The critical tasks in extraction-based summarization are identifying key phrases in the document and using them to discover relevant information to be included in the summary.

We cannot create summaries of all text manually; there is a great need for automatic methods. It creates a linguistically coherent and purposeful summary of the learning content while retaining basic information and general meaning. It automatically summarizes large pieces of learning content into brief and meaningful summaries.

Reasons why we need text summarization.

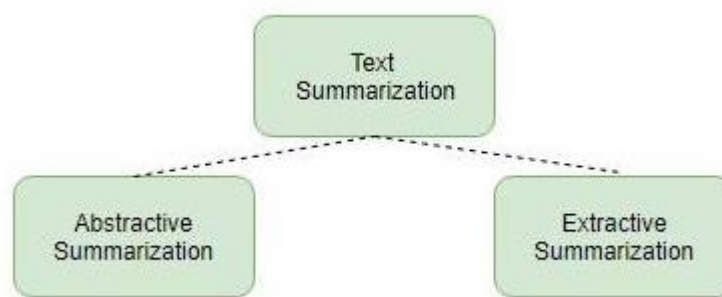
- ❖ Summaries reduce reading time.
- ❖ When researching documents, summaries make the selection process easier.
- ❖ Automatic summarization improves the effectiveness of indexing.
- ❖ Automatic summarization algorithms are less biased than human summarizers.
- ❖ Personalized summaries are useful in question-answering systems as they provide personalized information.
- ❖ Using automatic or semi-automatic summarization systems enables commercial abstract services to increase the number of texts they are able to process.

Summarization has been and continues to be a hot research topic in the data science arena. While there is intense activity in the research field, major advances in natural language processing and deep learning have been made in recent years. There is less literature available regarding real world applications of AI-driven summarization. One of the challenges with summarization is that it is hard to generalize. For example, summarizing a news article is very different to summarizing a financial earnings report. Certain text features like document length or genre (tech, sports, finance, travel, etc.) make the task of summarization a serious data science problem to solve. For this reason, the way summarization works largely depends on the use case and there is no one-size-fits-all solution.

Types of text summarization:

Automatic text summarization is the task of producing a concise and fluent summary while preserving key information content and overall meaning. The different dimensions of text summarization can be generally categorized based on its input type (single or multi document), purpose (generic, domain specific, or query-based) and output type (extractive or abstractive).

There are two different approaches that are used for text summarization:



- **Extractive Text Summarization:**

Here, content is extracted from the original data, but the extracted content is not modified in any way. Examples of extracted content include key-phrases that can be used to "tag" or index a text document, or key sentences (including headings). For text, extraction is analogous to the process of skimming, where the summary, headings and subheadings, figures, the first and last paragraphs of a section, and optionally the first and last sentences in a paragraph are read before one chooses to read the entire document in detail. Other examples of extraction that include key sequences of text in terms of clinical relevance (including patient/problem, intervention, and outcome).

In the extractive summarization, we will rely on three approaches: NLTK, SPACY and GENSIM. Each approach has its own characteristics in the way of summarizing

- **Abstractive Text Summarization:**

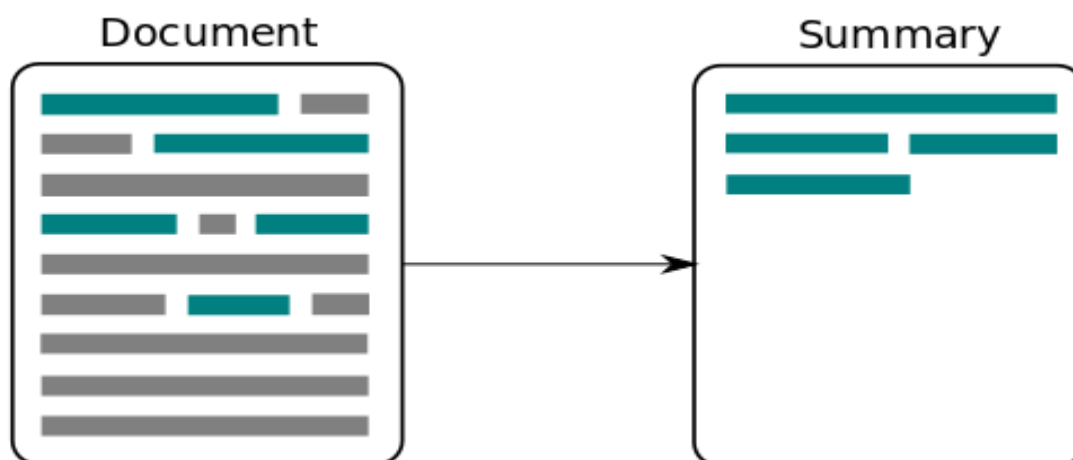
It is a more advanced method, many advancements keep coming out frequently (I will cover some of the best here). The approach is to identify the important sections, interpret the context and reproduce in a new way. This ensures that the core information is conveyed through the shortest text possible. Note that here, the sentences in summary are generated, not just extracted from original text.

Abstractive methods build an internal semantic representation of the original content, and then use this representation to create a summary that is closer to what a human might express. Abstraction may transform the extracted

content by paraphrasing sections of the source document, to condense a text more strongly than extraction. Such transformation, however, is computationally much more challenging than extraction, involving both natural language processing and often a deep understanding of the domain of the original text in cases where the original document relates to a special field of knowledge. "Paraphrasing" is even more difficult to apply to image and video, which is why most summarization systems are extractive.

Abstract summary achieves state-of-the-art results in various NLP tasks such as summarizing, answering questions, machine translation, etc. using a text-to-text converter trained on a large block of text. This means that the abstract summarization algorithm. It will rewrite the sentences as necessary rather than just picking up the sentences directly from the original text.

Classically, most successful text summarization methods are extractive because it is an easier approach, but abstractive approaches hold the hope of more general solutions to the problem.



Benefits:

The benefits of Automatic Text Summarization go beyond solving apparent problems. Some other advantages of Text Summarization include:

- ✓ **Saves Time:**

- By generating automatic summaries, text summarization helps content editors save time and effort, which otherwise is invested in creating summaries of articles manually.

- ✓ **Instant Response:**

- It reduces the user's effort involved in exacting the relevant information. With automatic text summarization, the user can summarize an article in just a few seconds by using the software, thereby decreasing their reading time.

- ✓ **Increases Productivity Level:**

- Text Summarization enables the user to scan through the contents of a text for accurate, brief, and precise information. Therefore, the tool saves the user from the workload by reducing the size of the text and increasing the productivity level as the user can channel their energy to other critical things.

- ✓ **Ensures All Important Facts are Covered:**

- The human eye can miss crucial details; however, automatic software does not. What every reader requires is to be able to pick out what is beneficial to them from any piece of content. The automatic text summarization technique helps the user gather all the essential facts in a document with ease.

- ✓ **highlight the main parts of the text**

- Examines the text to distinguish the important parts that represent the text.

The computational production of condensed copies of documents, is an important technology for the information society. Without summaries, it would be virtually impossible for humans to access the growing mass of information available online.

Although text summarization research is over 50 years old, some efforts are still needed due to the insufficient quality of automatic summaries and the number of interesting summarization topics being suggested in different contexts by end users, there has been a renewed interest in summarization as research focused Early on summarizing individual documents.

Text summarization research has focused many times more on the product: the abstract, and to a lesser extent on the cognitive basis of text understanding and production that underlies human summarization. Some limitations of current systems may benefit from a better understanding of the task's cognitive basis. However, formalizing the content of open domain documents is still a research problem, so most systems rely only on a selection of sentences from the original document set.

In the current web context, many approaches focus on summarizing many related documents (i.e. summarizing multiple documents). Most current abstract algorithms aim to generate snippets due to the difficulties associated with automatic generation of well-formed texts in arbitrary fields. It is generally accepted that there are a number of factors that determine what content to select from a source document and what type of output to produce.

An ongoing issue in this area is the evaluation process. Human judgment contains a large variety of what are considered good summaries, which means that the automatic evaluation process is particularly difficult. Manual evaluation can be used, but this is time and labor costly as it requires humans to read not only the

abstracts but also the source documents. As for the rest of the issues related to cohesion and coverage. One metric used at the annual Nest Document Understanding conferences, where research groups present their systems for both summarizing and translation tasks, compute summaries primarily from the overlap between automated summaries and pre-written human summaries. A high level of overlap is necessary to indicate a high level of mutual understanding between the two abstracts.

Use cases in the enterprise

These are some use cases where automatic summarization can be used across the enterprise:

1. Books and literature

Google has reportedly worked on projects that attempt to understand novels. Summarization can help consumers quickly understand what a book is about as part of their buying process.

2. Newsletters

Many weekly newsletters take the form of an introduction followed by a curated selection of relevant articles. Summarization would allow organizations to further enrich newsletters with a stream of summaries (versus a list of links), which can be a particularly convenient format in mobile.

3. Email overload

Companies like Slack were born to keep us away from constant emailing. Summarization could surface the most important content within email and let us skim emails faster.

4. E-learning and class assignments

Many teachers utilize case studies and news to frame their lectures. Summarization can help teachers more quickly update their content by producing summarized reports on their subject of interest.

5. Science and R&D

Academic papers typically include a human-made abstract that acts as a summary. However, when you are tasked with monitoring trends and innovation in a given sector, it can become overwhelming to read every abstract. Systems that can group papers and further compress abstracts can become useful for this task.

6. Media monitoring

The problem of information overload and “content shock” has been widely discussed. Automatic summarization presents an opportunity to condense the continuous torrent of information into smaller pieces of information.

7. Internal document workflow

Large companies are constantly producing internal knowledge, which frequently gets stored and under-used in databases as unstructured data. These companies should embrace tools that let them re-use already existing knowledge. Summarization can enable analysts to quickly understand everything the company has already done in a given subject, and quickly assemble reports that incorporate different points of view.

8. Financial research

Investment banking firms spend large amounts of money acquiring information to drive their decision-making, including automated stock trading. When you are a financial analyst looking at market reports and news every day, you will inevitably hit a wall and won't be able to read everything. Summarization systems tailored to financial documents like earning reports and financial news can help analysts quickly derive market signals from content.

9. Question answering and bots

Personal assistants are taking over the workplace and the smart home. However, most assistants are fairly limited to very specific tasks. Large-scale summarization could become a powerful question answering technique. By collecting the most relevant documents for a particular question, a summarizer could assemble a cohesive answer in the form of a multi-document summary.

10. Social media marketing

Companies producing long-form content, like whitepapers, e-books and blogs, might be able to leverage summarization to break down this content and make it sharable on social media sites like Twitter or Facebook. This would allow companies to further re-use existing content.

11. Legal contract analysis

More specific summarization systems could be developed to analyze legal documents. In this case, a summarizer might add value by condensing a contract to the riskier clauses, or help you compare agreements.

12. Patent research

Researching patents can be a tedious process. Whether you are doing market intelligence research or looking to file a new patent, a summarizer to extract the most salient claims across patents could be a time saver.

13. Meetings

With the growth of tele-working, the ability to capture key ideas and content from conversations is increasingly needed.

14. Help desk and customer support

Knowledge bases have been around for a while, and they are critical for SAAS platforms to provide customer support at scale. Still, users can sometimes feel overwhelmed when browsing help docs. The multi-document summarization provide key points from across help articles and give the user a well-rounded understanding of the issue.

Project Tools:

Our project is depends on a set of tools that helps for presenting it in an ideal way, these tools are distinguished as follows:

1- The platform are:

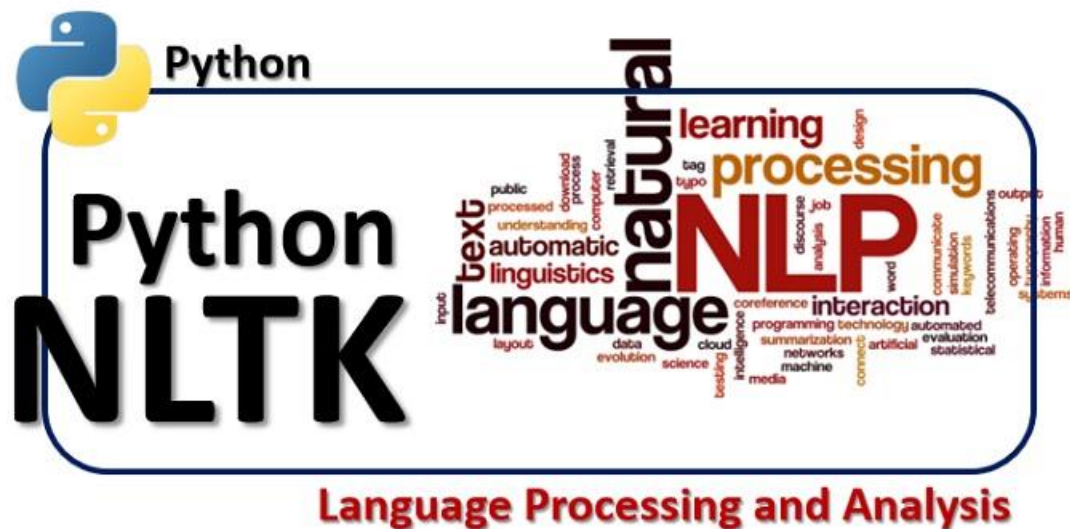
- Pycharm

2-The programming language is:

- Python 3.9

3-The libraries are:

- NLP
- tkinter
- NLTK
- Spacy
- Genism



CHAPTER 2: THEORETICAL BACKGROUND AND TOOLS

SYSTEM ANALYSIS

User Requirement

User can enter any file, whether its type is Link or word file or image (PDF) or can he write it by himself. User can specify which kind of algorithm to (Extractive, Abstractive). User can reset and enter the new file.

System Requirement

OS: Windows XP or higher. Python 3.9 installed on machine.

System display the way to summarize text if Extractive or abstractive.

System allow user to open the file in more than one type image or word file or link and allow user to write the text by himself.

System allow user to clear result or reset and enter the new file.

Functional

The system should allow user to enter the text by different way if the type of text (PDF , word file ,link) or can user write text by himself and then choice the summarize way Extractive or Abstractive and can user reset the text and enter new file and clear the result.

Non Functional

System must load result after 5 Sec, user not allow to change the algorithm it used, a program should be capable enough to handle a lot of text at one time

Availability Requirement

The system is available 100% for the user and is used 24/7 and 365 days a year. The system shall be operational at all times.

The next step after system introduction phase is system analysis, which include analysis the system which help in system design phase so, in this chapter we will discuss each diagram in the systems such as:

USE CASE DIAGRAM

A UML use case diagram is the primary form of system/software requirements for a new software program underdeveloped. Use cases specify the expected behavior (what), and not the exact method of making it happen (how). Use cases once specified can be denoted both textual and visual representation. A key concept of use case modeling is that it helps us design a system from the end user's perspective. It is an effective technique for communicating system behavior in the user's terms by specifying all externally visible system behavior.

ACTIVITY DIAGRAM

Activity diagram is another important behavioral diagram in UML diagram to describe dynamic aspects of the system. Activity diagram is essentially an advanced version of flow chart that modeling the flow from one activity to another activity.

USE CASE for Text Summarization:

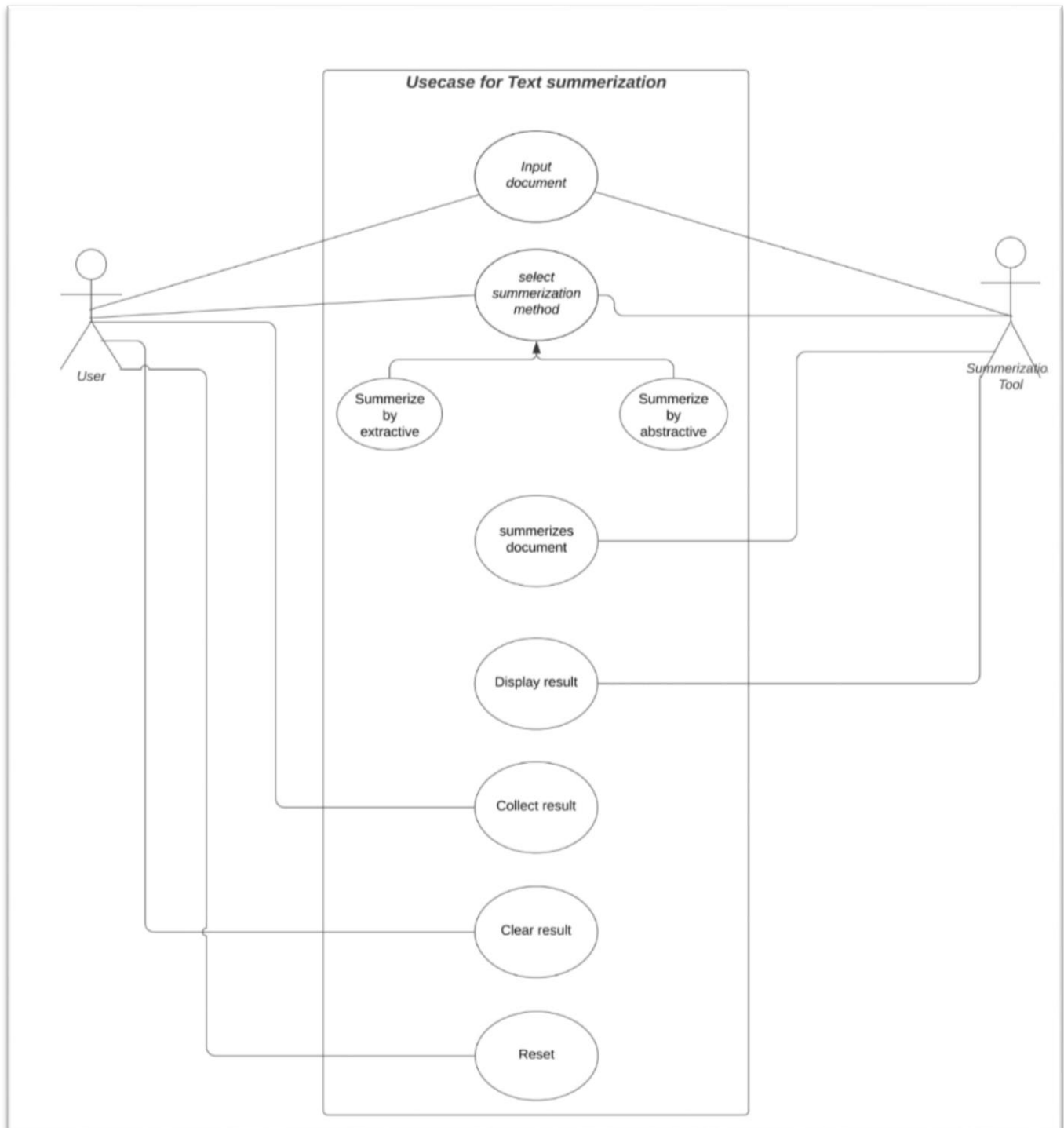


Fig.1: USECASE DIAGRAM

ACTIVITY DIAGRAM for Text Summarization:

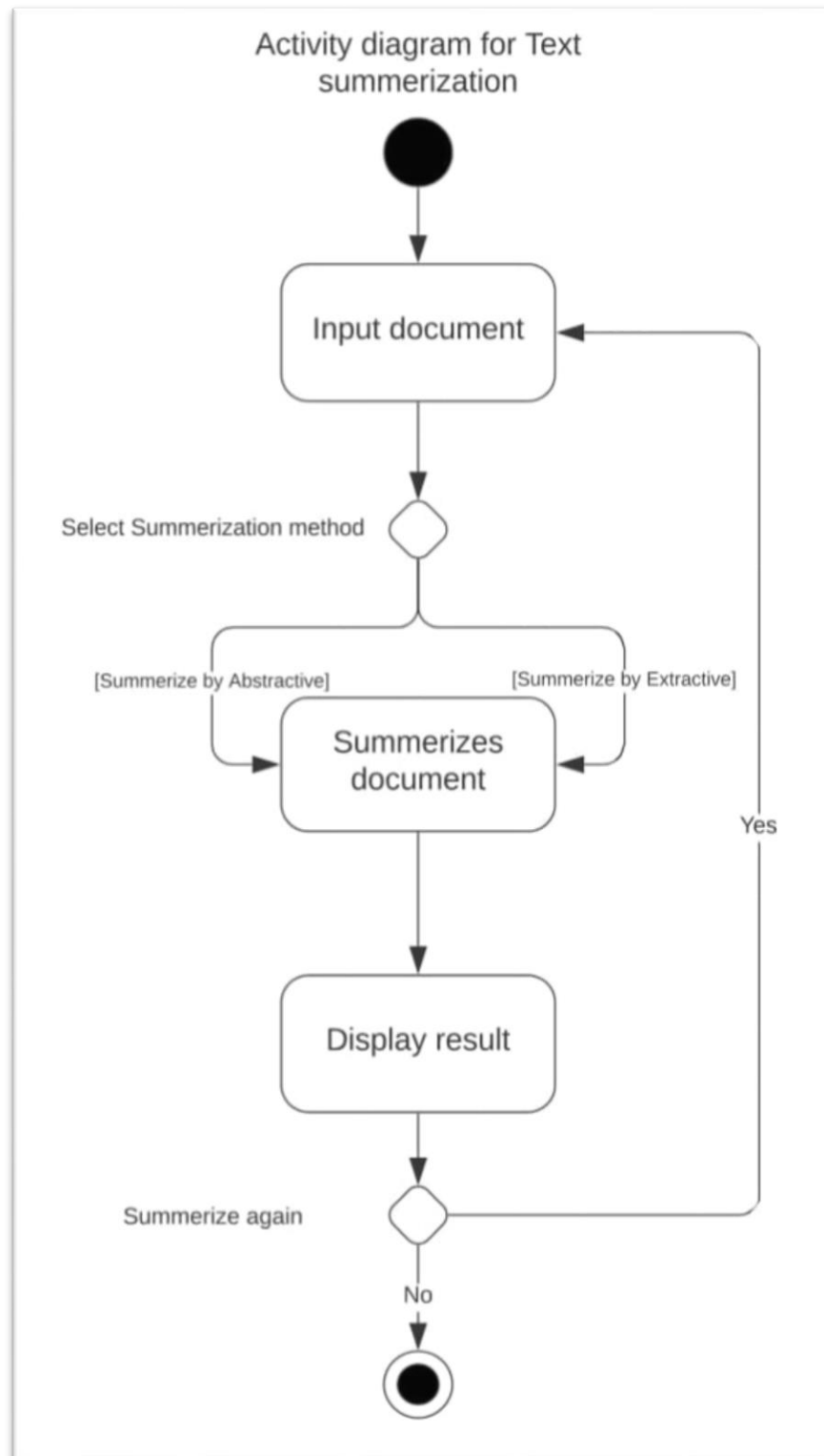


Fig.2: ACTIVITY DIAGRAM

TOOLS

Text Summarization app is desktop application for windows and unix based systems, we made a GUI and programmed in Python and some other libraries, and we will mention them in detail below.

- GUI
- TkInter

Programming

- Python
- NLTK
- Spacy
- Gensim
- Torch

GUI TOOLS

- TkInter: Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the tkinter package, and its extension, the tkinter.tix and the tkinter.ttk modules.

The tkinter package is a thin object-oriented layer on top of Tcl/Tk. To use tkinter, you don't need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. tkinter is a set of wrappers that implement the Tk widgets as Python classes. In addition, the internal module `_tkinter` provides a threadsafe mechanism which allows Python and Tcl to interact. tkinter's chief virtues are that it is fast, and that it usually comes bundled with Python. Although its standard documentation is weak, good material is available, which includes: references, tutorials, a book and others. tkinter is also famous for having an outdated look and feel, which has been vastly improved in Tk 8.5. Nevertheless, there are many other GUI libraries that you could be interested in.

Programming TOOLS

- **Python:** is an interpreted high-level general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant indentation. Its language constructs as well as its object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.



Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural), object-oriented and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.

Guido van Rossum began working on Python in the late 1980s, as a successor to the ABC programming language, and first released it in 1991 as Python 0.9.0. Python 2.0 was released in 2000 and introduced new features, such as list comprehensions and a garbage collection system using reference counting. Python 3.0 was released in 2008 and was a major revision of the language that is not completely backward-compatible and much Python 2 code does not run unmodified on Python 3. Python 2 was discontinued with version 2.7.18 in 2020.


Python consistently ranks as one of the most popular programming languages.

- **NLTK:** The Natural Language Toolkit (NLTK) is a platform used for building Python programs that work with human language data for applying in statistical natural language processing (NLP). It contains text processing libraries for tokenization, parsing, classification, stemming, tagging and semantic reasoning. It also includes graphical demonstrations and sample data sets as well as accompanied by a cook book and a book which explains the principles behind the underlying language processing tasks that NLTK supports.




Also it said it is an open source library for the Python programming language originally written by Steven Bird, Edward Loper and Ewan Klein for use in development and education.

It comes with a hands-on guide that introduces topics in computational linguistics as well as programming fundamentals for Python which makes it suitable for linguists who have no deep knowledge in programming, engineers and researchers that need to delve into computational linguistics, students and educators. NLTK includes more than 50 corpora and lexical sources such as the Penn Treebank Corpus, Open Multilingual Wordnet, Problem Report Corpus, and Lin's Dependency Thesaurus.

- **Spacy:** is an open-source software library for advanced natural language processing, written in the programming languages Python and Cython. The library is published under the MIT license and its main developers are Matthew Honnibal and Ines Montani, the founders of the software company Explosion. 

Unlike NLTK, which is widely used for teaching and research, spaCy focuses on providing software for production usage. spaCy also supports deep learning workflows that allow connecting statistical models trained by popular machine learning libraries like TensorFlow, PyTorch or MXNet through its own machine learning library Thinc. Using Thinc as its backend, spaCy features convolutional neural network models for part-of-speech tagging, dependency parsing, text categorization and named entity recognition (NER). Prebuilt statistical neural network models to perform these task are available for 17 languages, including English, Portuguese, Spanish, Russian and Chinese, and there is also a multi-language NER model. Additional support for tokenization for more than 65 languages allows users to train custom models on their own datasets as well.

Gensim: is an open-source library for unsupervised topic modeling and natural language processing, using modern statistical machine learning.  Gensim is implemented in Python and Cython for performance. Gensim is designed to handle large text collections using data streaming and incremental online algorithms, which differentiates it from most other machine learning software packages that target only in-memory processing.

And it is designed to process raw, unstructured digital texts ("*plain text*") using unsupervised machine learning algorithms.

The algorithms in Gensim, such as Word2Vec, FastText, Latent Semantic Indexing (LSI, LSA, LsiModel), Latent Dirichlet Allocation (LDA, LdaModel) etc, automatically discover the semantic structure of documents by examining statistical co-occurrence patterns within a corpus of training documents. These algorithms are **unsupervised**, which means no human input is necessary – you only need a corpus of plain text documents. Once these statistical patterns are found, any plain text documents (sentence, phrase, word...) can be succinctly expressed in the new, semantic representation and queried for topical similarity against other documents (words, phrases...).

- **Torch:** is an open-source machine learning library, a scientific computing framework, and a script language based on the Lua programming language. It provides a wide range of algorithms for deep learning, and uses the scripting language LuaJIT, and an underlying C implementation. As of 2018, Torch is no longer in active development. However PyTorch, which is based on the Torch library, is actively developed as of December 2020.



The core package of Torch is `torch`. It provides a flexible N-dimensional array or Tensor, which supports basic routines for indexing, slicing, transposing, type-casting, resizing, sharing storage and cloning. This object is used by most other packages and thus forms the core object of the library. The Tensor also supports mathematical operations like `max`, `min`, `sum`, statistical distributions like uniform, normal and multinomial, and BLAS operations like dot product, matrix-vector multiplication, matrix-matrix multiplication, matrix-vector product and matrix product.

CHAPTER 3: Project implementation

An overview of text summarization App:

Text summarization app has some Features. These features are implemented into two phases:

Phase1: Extractive sector:

This sector utilized some features of the following libraries:

- Genism
- NLTK
- Spacy

Genism:

Is an open-source vector space modeling and topic modeling toolkit, implemented in the Python programming language. It uses NumPy, SciPy and optionally Cython for performance. It is specifically intended for handling large text collections, using efficient online, incremental algorithms. Gensim is commercially supported by the startup RaRe Technologies.

Genism includes implementations of tf-idf, random projections, word2vec and document2vec algorithms, hierarchical Dirichlet processes (HDP), latent semantic analysis (LSA) and latent Dirichlet allocation (LDA), including distributed parallel versions.

Latent Semantic Analysis (LSA):

Latent Semantic Analysis is one of the natural language processing techniques for analysis of semantics, which in broad level means that we are trying to dig out some meaning out of a corpus of text with the help of statistical and was introduced by **Jerome Bellegarde** in 2005.

LSA is basically a technique where we identify the patterns from the text document or in simple words we tend to find out relevant and important information from the text document. If we talk about whether it is supervised or unsupervised way, it is clearly an unsupervised approach.

Latent Dirichlet Allocation (LDA):

Latent Dirichlet Allocation (LDA) uses **Dirichlet distribution** (no wonder why it is named latent Dirichlet allocation), so what is **Dirichlet distribution**? It is a probability distribution but is much different than the normal distribution which includes mean and variance, unlike the normal distribution it is basically the sum of probabilities which combine together and added to be 1.

It has distinct K values for the number of k means the number of probabilities needed for example:

$$0.6 + 0.4 = 1 \text{ (k=2)}$$

$$0.3 + 0.5 + 0.2 = 1 \text{ (k=3)}$$

$$0.4 + 0.2 + 0.3 + 0.1 = 1 \text{ (k=4)}$$

Hierarchical Dirichlet processes (HDP):

In statistics and machine learning, the hierarchical Dirichlet process (HDP) is a nonparametric Bayesian approach to clustering grouped data. It uses a Dirichlet process for each group of data, with the Dirichlet processes for all groups sharing a base distribution which is itself drawn from a Dirichlet process.

NLTK:

Natural Language Toolkit is a very popular open source. Initially released in 2001, it is much older than Spacy (released 2015). It also provides many functionalities, but includes less efficient implementations.

SpaCy:

SpaCy is an open-source Python library that parses and "understands" large volumes of text. Separate models are available that cater to specific languages (English, French, German, etc.).

All previous libraries contributed to present an automated summarization in different ways.

Phase2: Abstractive sector:

Abstractive summarization, on the other hand, tries to guess the meaning of the whole text and presents the meaning to you.

It creates words and phrases, puts them together in a meaningful way, and along with that, adds the most important facts found in the text. This way, abstractive summarization techniques are more complex than extractive summarization techniques and are also computationally more expensive.

Measurement for Extractive and Abstractive summarization:

The summary of our application must be accurate, for that; we utilized some measures. These measures is designed to compare a summary with its full text and we will propose a new evaluation measure for assessing the quality of a summary Such as:

1- Comparison of the percentage of presence of a summary of the text in the original text.

2-Precision.

3-Longest Common Subsequence.

The main approach for summary quality determination is

The intrinsic content evaluation which is often done by comparison with an ideal

Each of these ways works differently:

First way: we utilized a library (difflib), this library works as follows:

It is a library that compares texts by knowing the percentage of presence of a summary of the text in the original text; from here we can know the summation amount from the original text.

Second way: it is utilized for extractive summarization through the following mathematical equation:

Equation:

Precision (P) is the length of text occurring in system and system summary

Divided by the length of text in the system summary.

Precision measures can count as a match only exactly the same sentences. This ignores the fact that two sentences can contain the same information even if they are written differently.

The third way:

Is called Longest Common Subsequence (LCS), Use this method for abstractive summarization, (LCS) measures can count with different sentences. This can calculate the two sentences can contain the same information even if they are written differently.

Equation:

$$lcs(x,y) = \frac{lenght(x) + lenght(y) - edit(x,y)}{2}$$

where X and Y are representations based on sequences of words or lemmas, lcs(X, Y) is the length of the longest common subsequence between X and Y , length(X) is the length of the string X, and edit(X, Y) is the edit distance of X and Y .

The GUI of Text summarization application:

-The graphical user interface (GUI) of text summarization app; is presented as follows:

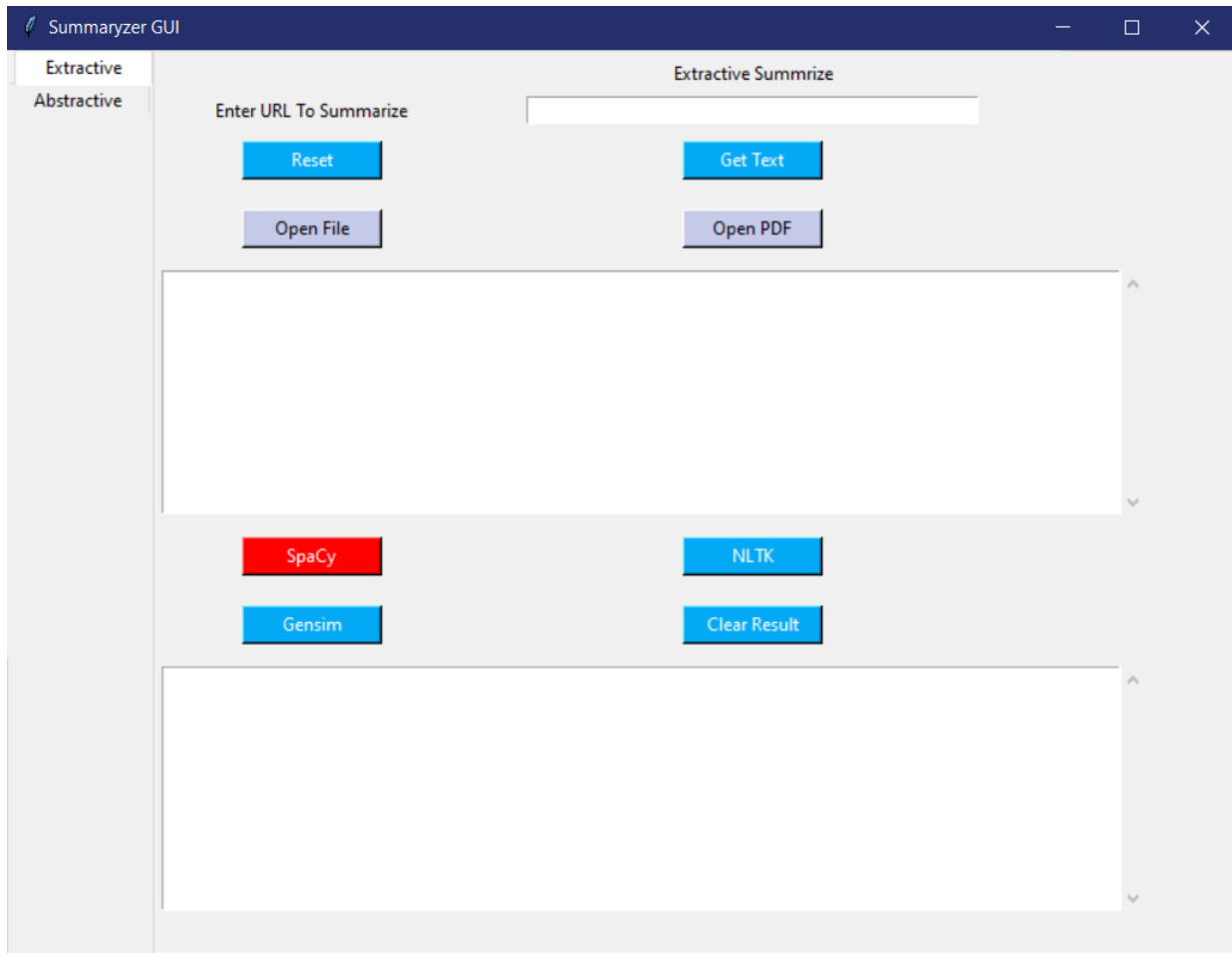


Fig.3: Dash Board

Figure 3:

In the GUI we programmed it, so that we can import an article from a website, from a PDF, from a file on the computer or even write it manually.

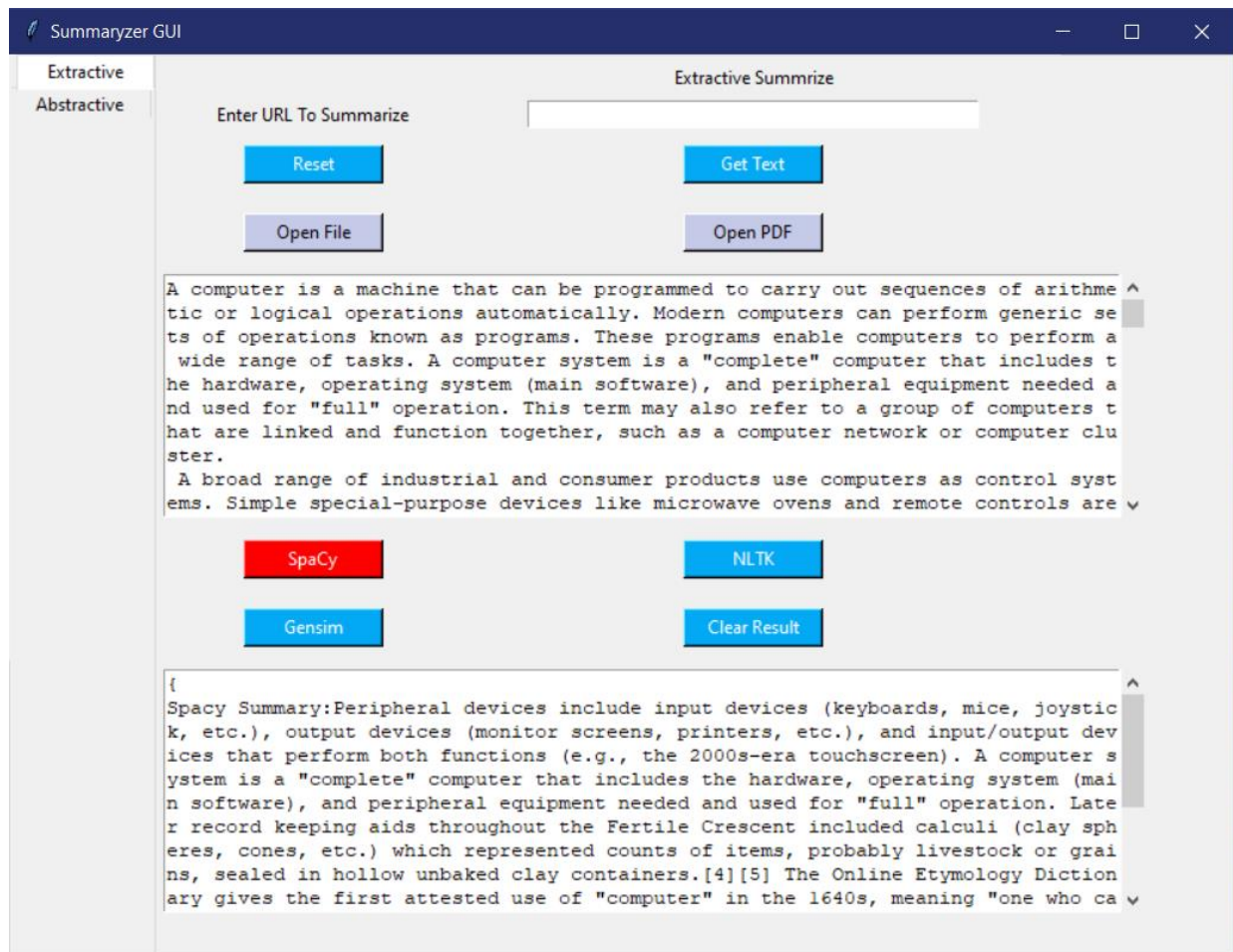


Fig.4: sPaCy summarize

In Figure 4:

We put a text to be summarized in the sPaCy algorithm, then it was summarized and shown on the screen below.

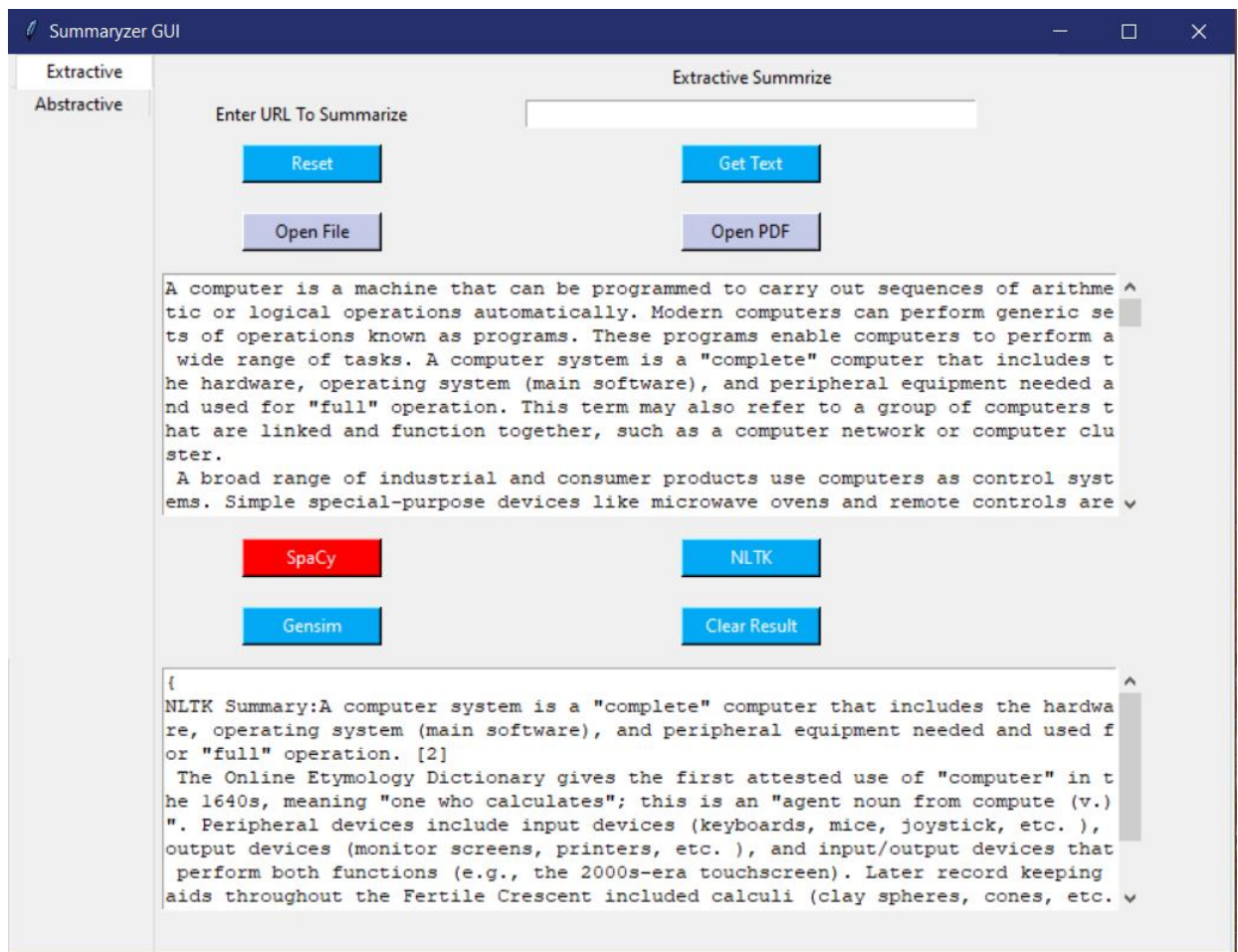


Fig.5: NLTK summarize

In Figure 5:

We put a text to be summarized in the NLTK algorithm, then it was summarized and shown on the screen below.

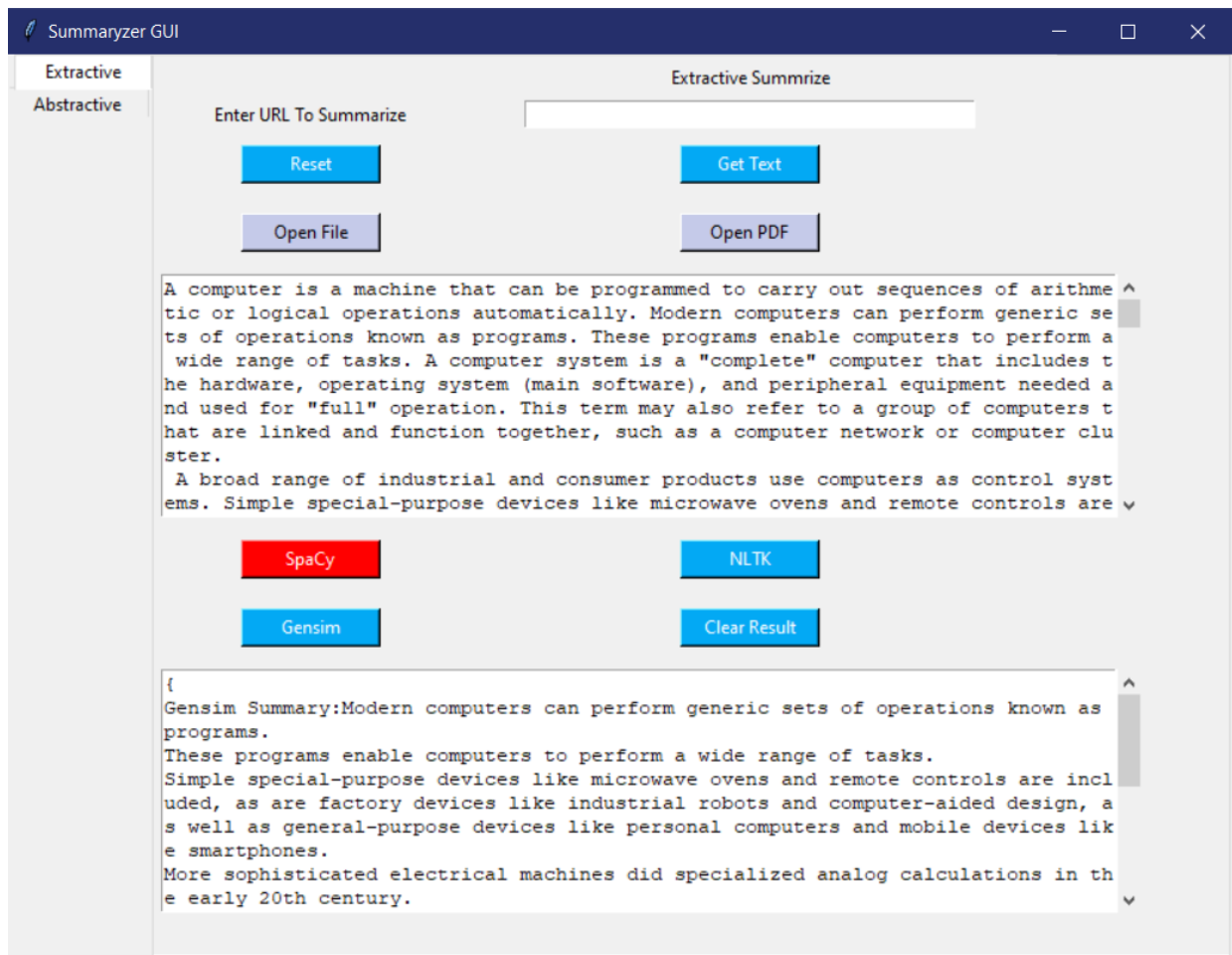


Fig.6: Gensim summarize

In Figure 6:

We put a text to be summarized in the Gensim algorithm, then it was summarized and shown on the screen below.

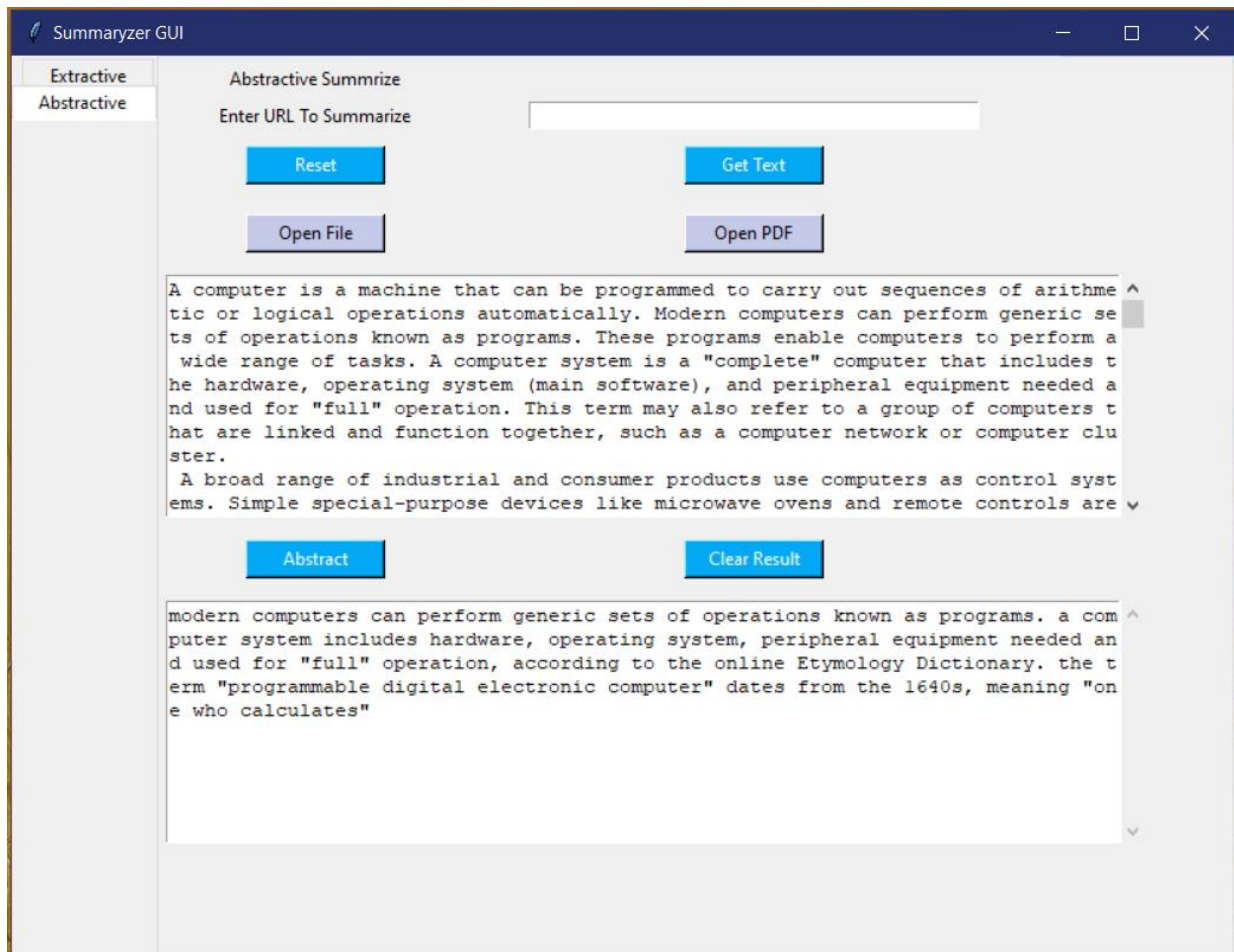


Fig.7: Abstract summarize

In Figure 7:

We put a text to be summarized in the Abstract algorithm, then it was summarized and shown on the screen below.

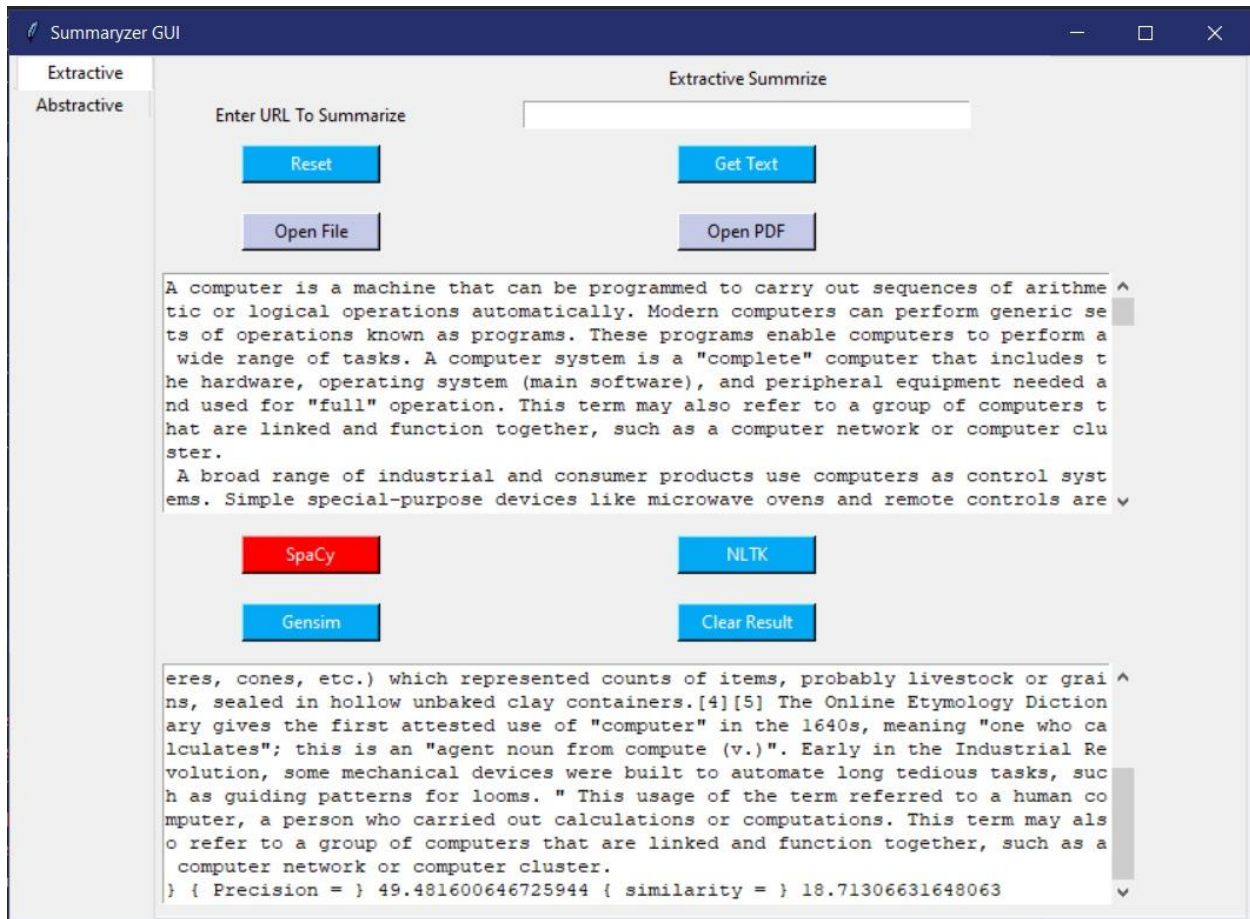


Fig.8: Accuracy for Extractive summarize

In Figure 8:

The accuracy of extractive summarize appears after the summary text below.

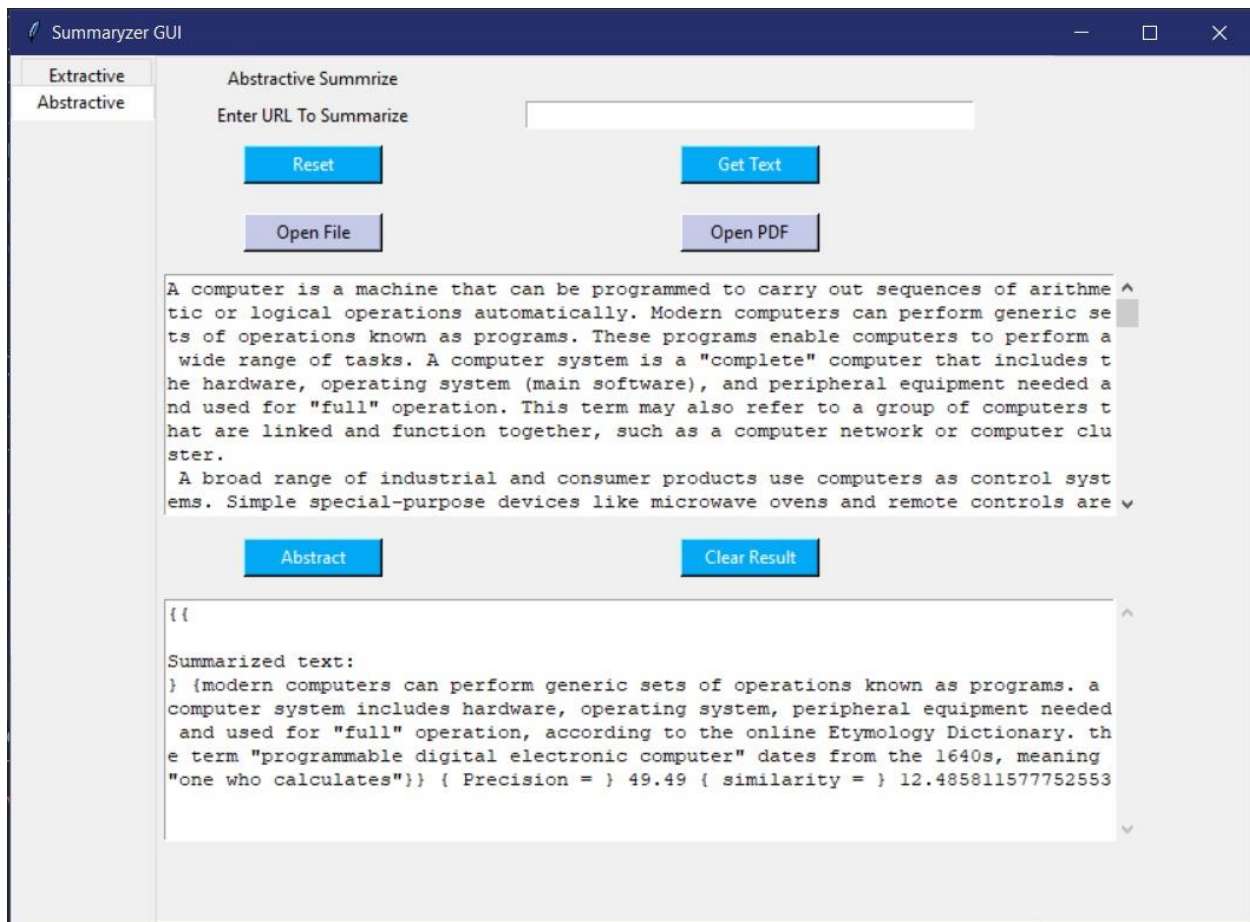


Fig.9: Accuracy for Abstract summarize

In Figure 9:

The accuracy of abstractive summarize appears after the summary text below.

Appendices

IMPLEMENTATION:

Spacy code:

```
# NLP Pkgs
import spacy

nlp = spacy.load('en_core_web_sm')
# Pkgs for Normalizing Text
from spacy.lang.en.stop_words import STOP_WORDS
from string import punctuation
# Import Heapq for Finding the Top N Sentences
from heapq import nlargest

def text_summarizer(raw_docx):
    raw_text = raw_docx
    docx = nlp(raw_text)
    stopwords = list(STOP_WORDS)
    # Build Word Frequency # word.text is tokenization in spacy
    word_frequencies = {}
    for word in docx:
        if word.text not in stopwords:
            if word.text not in word_frequencies.keys():
                word_frequencies[word.text] = 1
            else:
                word_frequencies[word.text] += 1

    maximum_frequency = max(word_frequencies.values())

    for word in word_frequencies.keys():
        word_frequencies[word] = (word_frequencies[word] / maximum_frequency)

    # Sentence Tokens
    sentence_list = [sentence for sentence in docx.sents]

    # Sentence Scores
    sentence_scores = {}
    for sent in sentence_list:
        for word in sent:
            if word.text.lower() in word_frequencies.keys():
                if len(sent.text.split(' ')) < 30:
                    if sent not in sentence_scores.keys():
                        sentence_scores[sent] = word_frequencies[word.text.lower()]
                    else:
                        sentence_scores[sent] += word_frequencies[word.text.lower()]

    summarized_sentences = nlargest(7, sentence_scores, key=sentence_scores.get)
    final_sentences = [w.text for w in summarized_sentences]
    summary = ' '.join(final_sentences)
    return summary
```

Fig.10: sPaCy code

NLTK code:

```
import nltk
nltk.download('stopwords')
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_frequency = max(word_frequencies.values())

    for word in word_frequencies.keys():
        word_frequencies[word] = (word_frequencies[word]/maximum_frequency)

    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
```

Fig.11: NLTK code

Gensim code:

```
import logging
import os
import os.path
import re

import xml.sax # for parsing arXiv articles

from gensim import utils

import sys
if sys.version_info[0] >= 3:
    unicode = str

PAT_TAG = re.compile(r'<(.*?)>(.*?)</.*?>')
logger = logging.getLogger('gensim.corpora.sources')

class ArticleSource:

    def init(self, sourceId):
        self.sourceId = sourceId

    def str(self):
        return self.sourceId

    def findArticles(self):
        raise NotImplementedError('Abstract Base Class')

    def getContent(self, uri):
        raise NotImplementedError('Abstract Base Class')

    def getMeta(self, uri):
        raise NotImplementedError('Abstract Base Class')

    def tokenize(self, content):
        raise NotImplementedError('Abstract Base Class')

    def normalizeWord(self, word):
        raise NotImplementedError('Abstract Base Class')
# endclass ArticleSource

class DmlSource(ArticleSource):

    def init(self, sourceId, baseDir):
        self.sourceId = sourceId
        self.baseDir = os.path.normpath(baseDir)

    def str(self):
        return self.sourceId

    @classmethod
    def parseDmlMeta(cls, xmlfile):

        result = {}
        xml = open(xmlfile)
```

Fig.12.1: Gensim code


```

for line in xml:
    if line.find('<article>') >= 0: # skip until the beginning of <article> tag
        break
for line in xml:
    if line.find('</article>') >= 0: # end of <article>, we're done
        break
p = re.search(PAT_TAG, line)
if p:
    name, cont = p.groups()
    name = name.split()[0]
    name, cont = name.strip(), cont.strip()
    if name == 'msc':
        if len(cont) != 5:
            logger.warning('invalid MSC=%s in %s', cont, xmlfile)
            result.setdefault('msc', []).append(cont)
            continue
    if name == 'idMR':
        cont = cont[2:] # omit MR from MR123456
    if name and cont:
        result[name] = cont
xml.close()
return result

def idFromDir(self, path):
    assert len(path) > len(self.baseDir)
    intId = path[1 + path.rfind('#'):]
    pathId = path[1 + len(self.baseDir):]
    return (intId, pathId)

def isArticle(self, path):
    # In order to be valid, the article directory must start with '#'
    if not os.path.basename(path).startswith('#'):
        return False
    # and contain the fulltext.txt file
    if not os.path.exists(os.path.join(path, 'fulltext.txt')):
        logger.info('missing fulltext in %s', path)
        return False
    # and also the meta.xml file
    if not os.path.exists(os.path.join(path, 'meta.xml')):
        logger.info('missing meta.xml in %s', path)
        return False
    return True

def findArticles(self):
    dirTotal = artAccepted = 0
    logger.info("looking for '%s' articles inside %s", self.sourceId, self.baseDir)
    for root, dirs, files in os.walk(self.baseDir):
        dirTotal += 1
        root = os.path.normpath(root)
        if self.isArticle(root):
            artAccepted += 1
            yield self.idFromDir(root)
    logger.info('%i directories processed, found %i articles', dirTotal, artAccepted)

def getContent(self, uri):
    """
    Return article content as a single large string.
    """
    intId, pathId = uri
    filename = os.path.join(self.baseDir, pathId, 'fulltext.txt')
    return open(filename).read()

def getMeta(self, uri):
    """
    Return article metadata as a attribute->value dictionary.
    """
    intId, pathId = uri
    filename = os.path.join(self.baseDir, pathId, 'meta.xml')
    return DmlSource.parseDmlMeta(filename)

def tokenize(self, content):
    return [token.encode('utf8') for token in utils.tokenize(content, errors='ignore')] if not

def normalizeWord(self, word):
    wordU = unicode(word, 'utf8')
    return wordU.lower().encode('utf8') # lowercase and then convert back to bytestring

```

Fig.12.2: Gensim code

```

class DmlCzSource(DmlSource):

    def idFromDir(self, path):
        assert len(path) > len(self.baseDir)
        dmlczId = open(os.path.join(path, 'dSPACE_id')).read().strip()
        pathId = path[1 + len(self.baseDir):]
        return (dmlczId, pathId)

    def isArticle(self, path):
        # in order to be valid, the article directory must start with '#'
        if not os.path.basename(path).startswith('#'):
            return False
        # and contain a dSPACE_id file
        if not (os.path.exists(os.path.join(path, 'dSPACE_id'))):
            logger.info('missing dSPACE_id in %s', path)
            return False
        # and contain either fulltext.txt or fulltext_dSPACE.txt file
        if not (os.path.exists(os.path.join(path, 'fulltext.txt'))
                or os.path.exists(os.path.join(path, 'fulltext-dSPACE.txt'))):
            logger.info('missing fulltext in %s', path)
            return False
        # and contain the meta.xml file
        if not os.path.exists(os.path.join(path, 'meta.xml')):
            logger.info('missing meta.xml in %s', path)
            return False
        return True

    def getContent(self, uri):
        """
        Return article content as a single large string.
        """
        intId, pathId = uri
        filename1 = os.path.join(self.baseDir, pathId, 'fulltext.txt')
        filename2 = os.path.join(self.baseDir, pathId, 'fulltext-dSPACE.txt')

        if os.path.exists(filename1) and os.path.exists(filename2):
            # if both fulltext and dSPACE files exist, pick the larger one
            if os.path.getsize(filename1) < os.path.getsize(filename2):
                filename = filename2
            else:
                filename = filename1
        elif os.path.exists(filename1):
            filename = filename1
        else:
            assert os.path.exists(filename2)
            filename = filename2
        return open(filename).read()
# endclass DmlCzSource

class ArxmlivSource(ArticleSource):

    class ArxmlivContentHandler(xml.sax.handler.ContentHandler):
        def init(self):
            self.path = [] # help structure for sax event parsing
            self.tokens = [] # will contain tokens once parsing is finished

        def startElement(self, name, attr):
            # for math tokens, we only care about Math elements directly below <p>
            if name == 'Math' and self.path[-1] == 'p' and attr.get('mode', '') == 'inline':
                tex = attr.get('tex', '')
                if tex and not tex.isdigit():
                    self.tokens.append('$$$' % tex.encode('utf8'))
                self.path.append(name)

        def endElement(self, name):
            self.path.pop()

        def characters(self, text):
            # for text, we only care about tokens directly within the <p> tag
            if self.path[-1] == 'p':
                tokens = [
                    token.encode('utf8') for token in utils.tokenize(text, errors='ignore') if not t
                ]
                self.tokens.extend(tokens)

```

Fig.12.3: Gensim code

```

class ArxmlivErrorHandler(xml.sax.handler.ErrorHandler):

    def error(self, exception):
        pass

    warning = fatalError = error
# endclass ArxmlivErrorHandler

def init(self, sourceId, baseDir):
    self.sourceId = sourceId
    self.baseDir = os.path.normpath(baseDir)

def str(self):
    return self.sourceId

def idFromDir(self, path):
    assert len(path) > len(self.baseDir)
    intId = path[1 + path.rfind('#'):]
    pathId = path[1 + len(self.baseDir):]
    return (intId, pathId)

def isArticle(self, path):
    # in order to be valid, the article directory must start with '#'
    if not os.path.basename(path).startswith('#'):
        return False
    # and contain the tex.xml file
    if not os.path.exists(os.path.join(path, 'tex.xml')):
        logger.warning('missing tex.xml in %s', path)
        return False
    return True

def findArticles(self):
    dirTotal = artAccepted = 0
    logger.info("looking for '%s' articles inside %s", self.sourceId, self.baseDir)
    for root, dirs, files in os.walk(self.baseDir):
        dirTotal += 1
        root = os.path.normpath(root)
        if self.isArticle(root):
            artAccepted += 1
            yield self.idFromDir(root)
    logger.info('%i directories processed, found %i articles', dirTotal, artAccepted)

def getContent(self, uri):
    """
    Return article content as a single large string.
    """
    intId, pathId = uri
    filename = os.path.join(self.baseDir, pathId, 'tex.xml')
    return open(filename).read()

def getMeta(self, uri):
    """
    Return article metadata as an attribute->value dictionary.
    """
    # intId, pathId = uri
    # filename = os.path.join(self.baseDir, pathId, 'tex.xml')
    return {'language': 'eng'} # |

def tokenize(self, content):

    handler = ArxmlivSource.ArxmlivContentHandler()
    xml.sax.parseString(content, handler, ArxmlivSource.ArxmlivErrorHandler())
    return handler.tokens

def normalizeWord(self, word):
    if word[0] == '$': # ignore math tokens
        return word
    wordU = unicode(word, 'utf8')
    return wordU.lower().encode('utf8') # lowercase and then convert back to bytestring

```

Fig.12.4: Gensim code

Abstractive code:

```
def get_summary():
    model = T5ForConditionalGeneration.from_pretrained('t5-small')
    tokenizer = T5Tokenizer.from_pretrained('t5-small')
    device = torch.device('cpu')
    text = str(url_display1.get('1.0', tk.END))
    preprocess_text = text.strip().replace("\n", "")
    t5_prepared_Text = "summarize: " + preprocess_text
    tokenized_text = tokenizer.encode(t5_prepared_Text, return_tensors="pt").to(device)

    summary_ids = model.generate(tokenized_text,
                                num_beams=4,
                                no_repeat_ngram_size=2,
                                min_length=30,
                                max_length=100,
                                early_stopping=True)

    output = tokenizer.decode(summary_ids[0], skip_special_tokens=True)

    Str1 = text
    str2 = output
    printt = difflib.SequenceMatcher(None, Str1, str2, False).ratio() * 100

    edited = len(text)-len(output)
    Precision = (len(text)+len(output)+edited)/2
    Precisioncalc = Precision / 100

    result = ("\n\nSummarized text: \n", output), " Precision = ", Precisioncalc, " similarity = ", printt

    tab2_display_text.insert(tk.END, output)
```

Fig.13: Abstractive code

Main function:

```
# Core Packages
import difflib
import tkinter as tk
from tkinter import *
from tkinter import ttk
from tkinter.scrolledtext import *
import tkinter.filedialog
import PyPDF2
from tkinter import filedialog
import torch
import json
from transformers import T5Tokenizer, T5ForConditionalGeneration, T5Config

# NLP Pkgs
from spacy_summarization import text_summarizer
from gensim.summarization import summarize
from nltk_summarization import nltk_summarizer

# 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')

tab2 = ttk.Frame(tab_control)
tab3 = ttk.Frame(tab_control)

# ADD TABS TO NOTEBOOK
tab_control.add(tab3, text=f'{"Extractive":^20s}')
tab_control.add(tab2, text=f'{"Abstractive":^20s}')
```

Fig.14.1: Main function code

```

label1 = Label(tab3, text='Extractive Summrize', padx=5, pady=5)
label1.grid(column=1, row=0)

label2 = Label(tab2, text='Abstractive Summrize', padx=5, pady=5)
label2.grid(column=0, row=0)

tab_control.pack(expand=1, fill='both')

def get_summary():
    model = T5ForConditionalGeneration.from_pretrained('t5-small')
    tokenizer = T5Tokenizer.from_pretrained('t5-small')
    device = torch.device('cpu')
    text = str(url_display1.get('1.0', tk.END))
    preprocess_text = text.strip().replace("\n", " ")
    t5_prepared_Text = "summarize: " + preprocess_text
    tokenized_text = tokenizer.encode(t5_prepared_Text, return_tensors="pt").to(device)

    summary_ids = model.generate(tokenized_text,
                                num_beams=4,
                                no_repeat_ngram_size=2,
                                min_length=30,
                                max_length=100,
                                early_stopping=True)

    output = tokenizer.decode(summary_ids[0], skip_special_tokens=True)

    Str1 = text
    str2 = output
    printt = difflib.SequenceMatcher(None, Str1, str2, False).ratio() * 100

    edited = len(text)-len(output)
    Precision = (len(text)+len(output)+edited)/2
    Precisioncalc = Precision / 100

    result =("\n\nSummarized text: \n", output), " Precision = ", Precisioncalc, " similarity = ", printt

    tab2_display_text.insert(tk.END, output)

```

Fig.14.2: Main function code

```

def open_pdf():
    open_file = filedialog.askopenfilename(
        initialdir="C:/gui/",
        title="Open PDF File",
        filetypes=(
            ("PDF Files", "*.pdf"),
            ("All Files", "*.*"))

    if open_file:
        pdf_file = PyPDF2.PdfFileReader(open_file)
        page = pdf_file.getPage(0)
        page_stuff = page.extractText()
        io = page_stuff.split()
        url_display.insert(3.0, io)

def open_pdf1():
    open_file = filedialog.askopenfilename(
        initialdir="C:/gui/",
        title="Open PDF File",
        filetypes=(
            ("PDF Files", "*.pdf"),
            ("All Files", "*.*"))

    if open_file:
        pdf_file = PyPDF2.PdfFileReader(open_file)
        page = pdf_file.getPage(0)
        page_stuff = page.extractText()
        io = page_stuff.split()
        url_display1.insert(3.0, io)

def clear_display_result():
    tab3_display_text.delete('1.0', END)

# Clear For URL
def clear_url_entry():
    url_entry.delete(0, END)

```

Fig.14.3: Main function code


```

# Open File to Read and Process
def openfiles():
    file1 = tkinter.filedialog.askopenfilename(filetypes=(("Text Files", ".txt"), ("All files", "*")))
    read_text = open(file1).read()
    url_display.insert(tk.END, read_text)

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 = text_summarizer(raw_text)
    result = '\nSummary:{}'.format(final_text)
    tab3_display_text.insert(tk.END, result)

def use_spacy():
    raw_text = url_display.get('1.0', tk.END)
    final_text = text_summarizer(raw_text)
    print(final_text)

    Str1 = raw_text
    str2 = text_summarizer(raw_text)
    printt = difflib.SequenceMatcher(None, Str1, str2, False).ratio() * 100

    Precision = len(raw_text)*len(text_summarizer(raw_text))/len(raw_text)
    Precisioncalc = Precision / 100
    result = '\nSpacy Summary:{}\n'.format(final_text), " Precision = ", Precisioncalc, " similarity = ", printt
    tab3_display_text.insert(tk.END, result)

```

Fig.14.4: Main function code


```

def use_nltk():
    raw_text = url_display.get('1.0', tk.END)
    final_text = nltk_summarizer(raw_text)
    print(final_text)

    Str1 = raw_text
    str2 = nltk_summarizer(raw_text)
    printt = difflib.SequenceMatcher(None, Str1, str2, False).ratio() * 100

    Precision = len(raw_text)*len(nltk_summarizer(raw_text))/len(raw_text)
    Precisioncalc = Precision / 100
    result = '\nNLTK Summary:{}\n'.format(final_text), " Precision = ", Precisioncalc, " similarity = ", printt
    tab3_display_text.insert(tk.END, result)

def use_gensim():
    raw_text = url_display.get('1.0', tk.END)
    final_text = summarize(raw_text)
    print(final_text)

    Str1 = raw_text
    str2 = summarize(raw_text)
    printt = difflib.SequenceMatcher(None, Str1, str2, False).ratio() * 100

    Precision = len(raw_text)*len(summarize(raw_text))/len(raw_text)
    Precisioncalc = Precision / 100
    result = '\nGensim Summary:{}\n'.format(final_text), " Precision = ", Precisioncalc, " similarity = ", printt
    tab3_display_text.insert(tk.END, result)

# 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)

```

Fig.14.5: Main function code

```

# BUTTONS
button1 = Button(tab3, text="Reset", command=clear_url_entry, width=12, bg='#03A9F4', fg='ffff')
button1.grid(row=4, column=0, padx=10, pady=10)

button2 = Button(tab3, text="Get Text", command=get_text, width=12, bg='#03A9F4', fg='ffff')
button2.grid(row=4, column=1, padx=10, pady=10)

button3 = Button(tab3, text="Open File", width=12, command=openfiles, bg='#c5cae9')
button3.grid(row=5, column=0, padx=10, pady=10)

button4 = Button(tab3, text="Open PDF", width=12, command=open_pdf, bg='#c5cae9')
button4.grid(row=5, column=1, padx=10, pady=10)

button5 = Button(tab3, text="SpaCy", command=use_spacy, width=12, bg='red', fg='ffff')
button5.grid(row=8, column=0, padx=10, pady=10)

button6 = Button(tab3, text="Clear Result", command=clear_display_result, width=12, bg='#03A9F4', fg='ffff')
button6.grid(row=9, column=1, padx=10, pady=10)

button7 = Button(tab3, text="NLTK", command=use_nltk, width=12, bg='#03A9F4', fg='ffff')
button7.grid(row=8, column=1, padx=10, pady=10)

button8 = Button(tab3, text="Gensim", command=use_gensim, width=12, bg='#03A9F4', fg='ffff')
button8.grid(row=9, column=0, 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=11, column=0, columnspan=3, padx=5, pady=5)

l1 = Label(tab2, text="Enter URL To Summarize")
l1.grid(row=1, column=0)

raw_entry1 = StringVar()
url_entry1 = Entry(tab2, textvariable=raw_entry, width=50)
url_entry1.grid(row=1, column=1)

```

Fig.14.6: Main function code

```

# BUTTONS

button9 = Button(tab2, text="Reset", command=clear_url_entry, width=12, bg='#03A9F4', fg='fff')
button9.grid(row=4, column=0, padx=10, pady=10)

button10 = Button(tab2, text="Get Text", command=get_text, width=12, bg='#03A9F4', fg='fff')
button10.grid(row=4, column=1, padx=10, pady=10)

button11 = Button(tab2, text="Open File", width=12, command=openfiles, bg='#c5cae9')
button11.grid(row=5, column=0, padx=10, pady=10)

button12 = Button(tab2, text="Open PDF", width=12, command=open_pdf1, bg='#c5cae9')
button12.grid(row=5, column=1, padx=10, pady=10)

button13 = Button(tab2, text="Clear Result", command=clear_display_result, width=12, bg='#03A9F4', fg='fff')
button13.grid(row=9, column=1, padx=10, pady=10)

button14 = Button(tab2, text="Abstract", command=get_summary, width=12, bg='#03A9F4', fg='fff')
button14.grid(row=9, column=0, padx=10, pady=10)

url_display1 = ScrolledText(tab2, height=10)
url_display1.grid(row=7, column=0, columnspan=3, padx=5, pady=5)

tab2_display_text = ScrolledText(tab2, height=10)
tab2_display_text.grid(row=11, column=0, columnspan=3, padx=5, pady=5)

window.mainloop()

```

Fig.14.7: Main function code

CHAPTER 4: Conclusion

Conclusion

Text Summarization is project that include many algorithms, as well as multiple ways to summarize text and works on computers as a desktop application.

The application is built in the Python language. Text Summarization based on input type. Summarize the text on the basis of the type of document source where there is more than one way to enter the text to be summarize, first writing the text and this is the traditional method, or the file is located on the special device where it is opened if it is (txt, pdf) or you can also put a url and the program brings the text from the url address of the text.

The project is divided into two parts of the summarize:

- The special part of machine learning (extractive). The extractive part is important words and phrases are taken from the original text and put together to make the summary. It is divided into counting forms of the algorithm, including (Spacy – NLTK - Gensim) and the percentage of abbreviation varies according to which of the algorithms is used.
- The special part is deep learning (abstractive). The abstract for the abstractive is a completely different part as it extracts the original and important sentence from a text document and paraphrases it into appropriate synonyms. This way, it will look like a completely different text but has the same meaning as the original text.

Therefore, we suggest to use the text summarization project so that we can benefit more and save a lot of time and publish more information as soon as possible and in this way you can summarize and know the lessons learned and specific goals from more than one text in a short time.

References:

1. Abstractive text summarization :
Nallapati, Ramesh, et al. "Abstractive text summarization using sequence-to-sequence rnns and beyond." arXiv preprint arXiv:1602.06023 (2016).
2. Extractive text summarization :
Miller, Derek. "Leveraging BERT for extractive text summarization on lectures." arXiv preprint arXiv:1906.04165 (2019).
3. Spacy library:
https://github.com/Jcharis/NLP-Web-Apps/blob/master/Summaryzer_GUI/spacy_summarization.py.
4. Nltk library:
https://github.com/Jcharis/NLP-WebApps/blob/master/Summaryzer_GUI/nltk_summarization.py.
5. Gensim library:
<https://github.com/RaRe-Technologies/gensim>.
6. Tkinter library:
 - 6.1. Summerfield, Mark. Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming (paperback). Pearson Education, 2007.
 - 6.2. <https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/index.html>.
7. Evaluation the text summarization techniques:
https://www.researchgate.net/publication/220106310_Evaluation_Measures_for_Text_Summarization.
8. The project has been uploaded to:

ملخص باللغة العربية

تلخيص النص:

تهدف فكره المشروع الي حذف العبارات والأفكار غير الأساسية، مع الاحتفاظ بالأفكار الرئيسية. وهو اختصار نص طويل مع الحفاظ على معناه وبنائه الأساسية حيث ان النظام يساعد علي توفير راحة ووقت لقراءه الاشياء التي يريد الانسان التعرف عليها ويتم فحص المعلومات بشكل مرتب ومنظم ويمكن ادخال النص عن طريق استخدام اكثر من منهجية :

Extractive (machine learning) and abstractive (deep learning)

وقد تم تطبيق المنهجية الاولى باستخدام ثلاث طرق (Gensim-NLTK-SpaCy)

وعرضنا تلخيص المقالة او النص مع اختلاف مصدره سواء كان (docx or pdf or url) اما المنهجية الاخرى فاعتمدت على تلخيص بشكل مختلف مقارنة بتلخيص البشر. حيث يقرأ البشر النص بأكمله ويفهمون معناه ويحاولون تقديم أقصى قدر من المعنى في الحد الأدنى من الجمل أثناء التلخيص. يستخدم التلخيص التجريدي طرقاً لغوية لفهم النص الأصلي. الهدف من هذه الطريقة مشابه ، أي نقل المعلومات بطريقة موجزة. إنه أكثر فاعلية من التلخيص الاستخراجي حيث يمكنه تكثيف جمل متعددة إلى عدد أقل من الجمل عن طريق إنشاء جمل جديدة خاصة به ، بدلاً من التلخيص الاستخراجي حيث لا يتم التلاعب ببنية الجملة. وهذا يتطلب معالجة متقدمة للغة الطبيعية وتقنيات ضغط مقارنة بالتلخيص الاستخراجي. وبالتالي ، يمكن أن يكون أكثر تكلفة من الناحية الحسابية. واخيرا تم تقييم نسبة التلخيص من المقالة او النص بناء على استخدام قياس (Longest Common Subsequence).