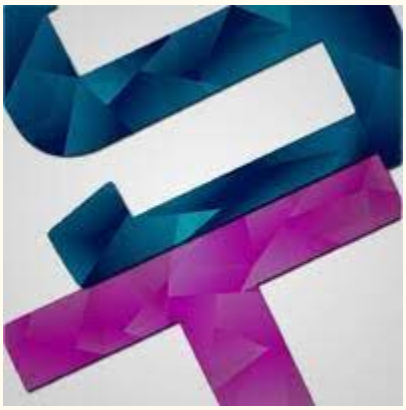


SNT HACKATHON 2023

PROBLEM STATEMENT: GENERATIVE AI FOR IMPACT

Summarizer using LED 🥸

GitHub Repo: https://github.com/Attention-is-All-We-Need/Document-Summary-Generator

TEAM NAME: AttentionIsAllWeNeed

Team Members :

1. Anwesh Saha (210178)
2. Arindom Bora (210183)
3. Ajay Sankar Makenna (210077)
4. Khush Khandelwal (210511)
5. Shreyash Nallawar (211010)
6. Vineet Kumar (201121)

Table of Contents

About the Project

It can :

- Summarize Research articles or any other PDFs
- Summarize the user's text
- Summarize web pages, including news and blogs or YouTube videos.

The Summarizer Model

The heart of the model - Longformers!

This app uses the ***Longformer Encoder-Decoder (LED)*** *for Narrative-Esque Long Text Summarization* architecture to summarize texts retrieved from PDFs, Documents, or web pages.

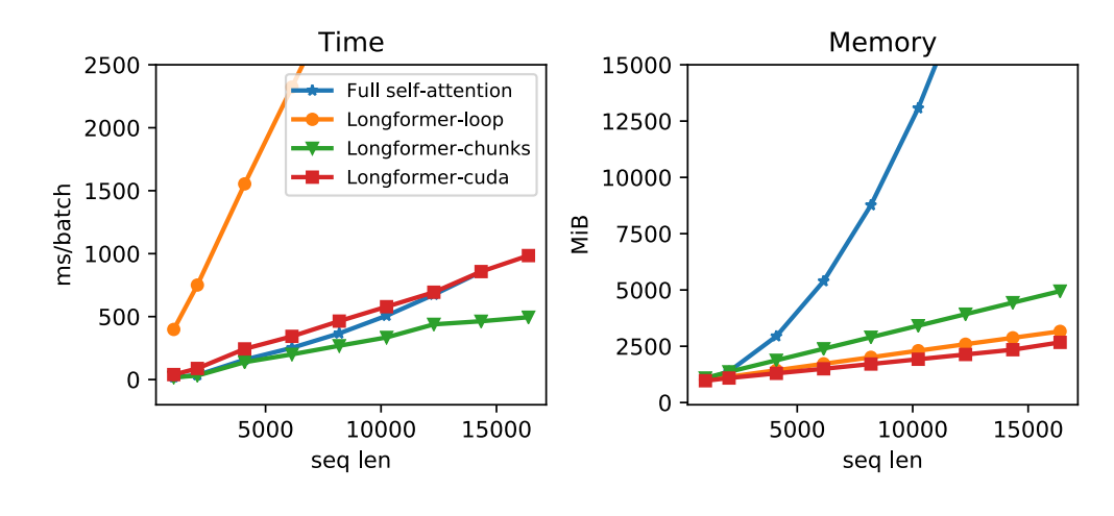
We had a couple of options in hand before we started our build:

1. Google's T5
2. Facebook's BART
3. **Longformer Encoder-Decoder (LED) for Narrative-Esque Long Text Summarization**

We chose the third one because of the following reasons:

- **Performance and Quality:**
  Transformer-based models cannot process long sequences due to their self-attention operation, which scales quadratically with the sequence length. Longformers, on the other hand, can summarize text with a more significant number of words.

  The original Transformer consisted of an encoder-decoder architecture for sequence-to-sequence tasks, such as summarization and translation. While encoder-only Transformers are effective on various NLP tasks, pre-trained encoder-decoder Transformer models (e.g., BART and T5 have achieved strong results on tasks like summarization.) Yet, such models can't efficiently scale to seq2seq tasks with longer inputs.



. Longformer's attention mechanism is a drop-in replacement for the standard self-attention and combines local windowed attention with task-motivated global attention. It's performance and memory usage is more efficient compared to traditional self-attention.

- **Training**
  The pretraining dataset used for this model was one of the primary factors influencing our decision. LED-base-book-summary is trained on diverse sources, including books and research articles, providing a more extensive and comprehensive understanding of language. This diverse training data helps the model capture a broader range of contexts and improves its ability to generate meaningful summaries.

The architecture of Longformer Encoder-Decoder (LED)

The model uses an encoder-decoder architecture for *seq2seq* modeling. What differentiates it from a traditional transformer is **'local'** and **'global'** attention.

- **Self-Attention in Traditional Transformers**: In traditional Transformers, self-attention simultaneously operates on the entire input sequence. It computes the attention scores between each token in the sequence and all other tokens, capturing their dependencies and relationships. This allows the model to establish both local and global connections. However, the quadratic complexity of self-attention concerning the sequence length could be more efficient for long sequences.
- **Local Windowed Attention**: Local windowed attention allows the model to attend to a subset of the input sequence, or a "local window," rather than the entire sequence simultaneously. This approach reduces the computational complexity

and memory requirements compared to full self-attention. Limiting the attention scope allows the model to capture local dependencies within a fixed window size.

- **Global Attention**: As the name suggests, the model can attend to the entire input sequence. It is often used with local windowed attention to capturing local and global dependencies. While local attention focuses on nearby tokens, global attention provides a broader context by attending to all tokens in the sequence.

Training Data: This model is pre-trained on the ***Booksum*** dataset by Salesforce. BookSum is a new collection of datasets specifically designed to address the limitations of existing text summarization datasets. Unlike other datasets mainly consisting of short-form source documents with limited dependencies, BookSum focuses on long-form narrative sources such as novels, plays, and stories. This dataset includes human-written summaries at three levels of granularity: paragraph, chapter, and book.

Number of parameters:- 162 M params

Trained on	Booksum dataset (salesforce/booksum (github.com))
Model Size	~1.6G
Number of parameters	162M params

```
class LED:
    def __init__(self):
        pass
    def summarize(self, text, max_length=500):
        hf_name = 'pszemraj/led-base-book-summary'
        summarizer = pipeline(
            "summarization",
            hf_name,
            device=0 if torch.cuda.is_available() else -1,
        )

        result = summarizer(
            text,
            min_length = 8,
            max_length = max_length,
            no_repeat_ngram_size=3,
            encoder_no_repeat_ngram_size=3,
            repetition_penalty=3.5,
            num_beams=4,
            do_sample=False,
            early_stopping=True,
        )

        return result[0]['summary_text']
```

BERT Embeddings

BERT (Bidirectional Encoder Representations from Transformers) is a state-of-the-art language representation model that has revolutionized various natural language processing (NLP) tasks. BERT embeddings capture contextual information by considering the entire sentence rather than individual words. This model can be fine-tuned on specific downstream tasks, such as text summarization, to generate meaningful representations of input text.

- BERT Pretraining: BERT is initially pre-trained on large corpora using masked language modeling and next-sentence prediction tasks. This pretraining helps BERT to learn general language representations.
- Fine-tuning BERT: After pretraining, BERT can be fine-tuned on specific tasks, such as text summarization. During fine-tuning, BERT is trained on labeled data specific to the task at hand, which helps it learn task-specific features and nuances.
- BERT Embeddings: The output of BERT's hidden layers can be used as embeddings for downstream tasks. These embeddings capture the contextual understanding of the input text, enabling better representation of words and sentences.

```python
class Embeddings:
    def __init__(self):
        logging.info("Initialising BERT transformer.")
        try :
            from transformers import BertTokenizer,BertModel
            self.model = BertModel.from_pretrained("bert-base-uncased")
            self.tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')


        except Exception as e:
            raise CustomException(e,sys)

    def tokenize_sentences(self,text):
        try :
            token=self.tokenizer.tokenize(text)
            token = self.tokenizer.convert_tokens_to_ids(token)
            return token
        except Exception as e:
            raise CustomException(e,sys)

    def embeddings(self,text):
        try :
            token = self.tokenize_sentences(text)
            with torch.no_grad():
                output = self.model(torch.tensor(token).unsqueeze(0))
                return output[1][0].tolist()
        except Exception as e:
            raise CustomException(e,sys)
```

HNSWlib for KNN based on Cosine similarity.

HNSWlib (Hierarchical Navigable Small World) is a library that provides an efficient data structure for approximate nearest neighbor search. In the context of text summarization, HNSWlib can perform a cosine similarity-based k-nearest neighbor (KNN) search on BERT embeddings.

- Building the HNSW Index: BERT embeddings of the documents in the corpus are indexed using the HNSWlib library. The HNSW index structure organizes the embeddings for efficient nearest neighbor search.
- Cosine Similarity: Cosine similarity is a widely used metric to measure the similarity between two vectors. In the case of BERT embeddings, cosine similarity helps determine the semantic similarity between documents.
- K-Nearest Neighbor Search: Given a query document, the HNSW index efficiently finds the k nearest neighbors in the embedding space based on cosine similarity. These nearest neighbors can then generate a summary by extracting meaningful information from similar documents.

```
text_lines = text.split('.')
emb= Embeddings()
input_embeddings=[]
for line in text_lines:
    input_embeddings.append(emb.embeddings(line))
logging.info('Generated embeddings for input text.')
input_embeddings=np.array(input_embeddings)
key_embed = emb.embeddings(key_word)
key_embed=np.array(key_embed)
logging.info('Generated embeddings for keyword.')

num_elements = len(input_embeddings)
ids = np.arange(num_elements)
dim = input_embeddings.shape[1]

p = hnswlib.Index(space = 'cosine', dim = dim)
p.init_index(max_elements = num_elements, ef_construction = 200, M = 16)
logging.info('Initialised hnswlib.')
p.add_items(input_embeddings, ids)
p.set_ef(50)
logging.info('Added items to hnswlib.')

labels, distances = p.knn_query(key_embed, k = min(20,num_elements))
```

Text Extraction

1. PDFs: PyPDF2 is a Python library for working with PDF files. We used it to extract text from PDF documents.

```python
def gen_para_pdf(file_name):
    try :
        with open(file_name, 'rb') as file:
            pdf_reader = PyPDF2.PdfReader(file)
            paragraphs=[]
            for page_number in range(len(pdf_reader.pages)):
                page = pdf_reader.pages[page_number]
                text = page.extract_text()
                paragraph = text.split('.\n')
                for para in paragraph:
                    paragraphs.append(para)
            return paragraphs
    except Exception as e:
        logging.info("PDF File not found.")
        paragraph=[]
        return paragraph
```

2. Word Files: We used **python-docx,** another Python library for extracting text from Microsoft Word (.docx or .doc) files.

```python
def gen_para_doc(file_name):
    try:
        doc = docx.Document(file_name)
        text = []
        for paragraph in doc.paragraphs:
            if (paragraph) :
                p_text = paragraph.text
                if len(p_text):
                    text.append(p_text)
        return text
    except Exception as e:
        logging.info("Doc File not found.")
        paragraph=[]
        return paragraph
```

3. Text Files: Python has inbuilt features to deal with text files using the **open** command.

```
def gen_para_txt(file_name):
    try:
        paragraphs=[]
        with open(file_name, 'r') as file:
            text = file.read()
            paragraph = str(text).split('.\n')
            for para in paragraph:
                if (para):
                    paragraphs.append(para)
        return paragraphs
    except Exception as e:
        logging.info("File not found.")
        paragraph=[]
        return paragraph
```

A general function that checks the filetype by the extension of the file and redirects the file to the specific function:

```
def gen_para_file(file_name):
    file_name = os.path.join(folder,file_name)
    file_extension = os.path.splitext(file_name)[1]
    if (file_extension=='.pdf'):
        return gen_para_pdf(file_name)
    elif (file_extension == '.doc' or file_extension=='.docx'):
        return gen_para_doc(file_name)
    elif (file_extension=='.txt'):
        return gen_para_txt(file_name)
    logging.info("Paragraph of {} is generated.".format(file_name))
```

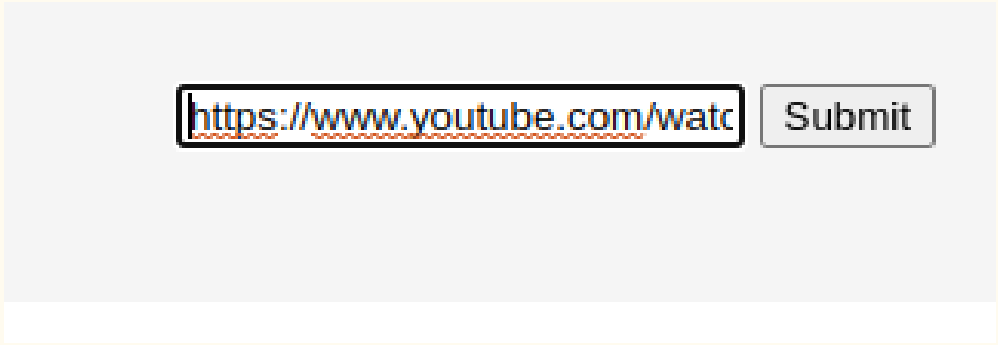
4. Web Pages: We used Python's Request and BeautifulSoup libraries to scrape web pages and extract their headings and paragraph elements.

```
import requests
from bs4 import BeautifulSoup

def get_website(url):
    try :
        response = requests.get(url)
        if response.status_code == 200:
            soup = BeautifulSoup(response.content, 'html.parser')

            article_title = soup.find('h1').get_text()
            paragraphs = soup.find_all('p')

            article_text = '\n'.join([p.get_text() for p in paragraphs])

            return article_title+ " "+article_text
        else:
            logging.info('URL not found. Error {}'.format(response.status_code))

    except Exception as e:
        raise CustomException(e,sys)
def get_yt_id(url, ignore_playlist=False):
```

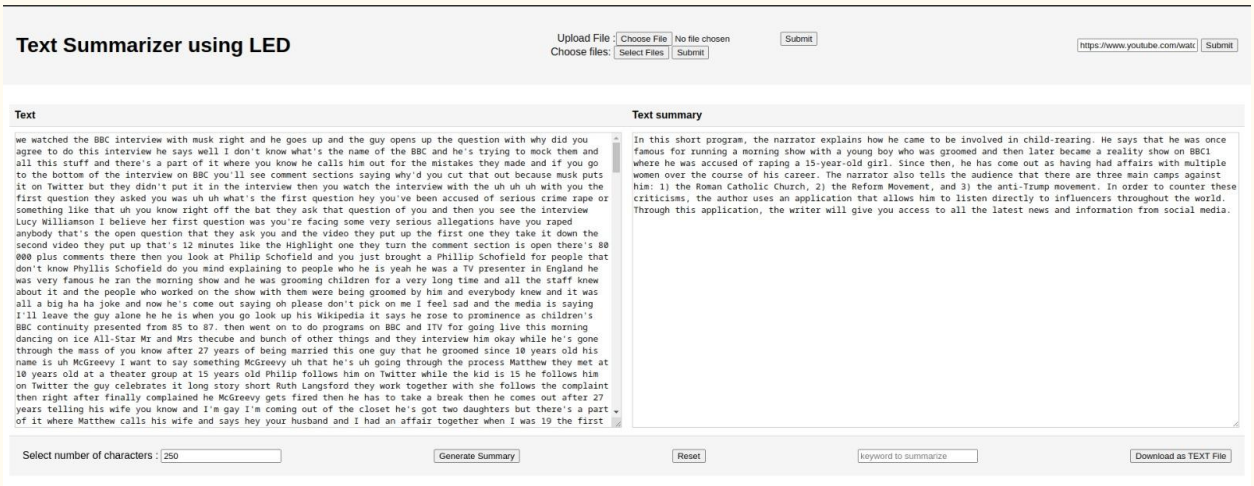
To process the URL of a youtube video, we retrieved the video ID, typically found in the YouTube video URL after the "v=" parameter. We then retrieve the transcript of the video using the YouTube Transcript API. We extracted and displayed the text from the transcript in the text area. On clicking the Generate Summary button, a summary can be generated.

```
def get_yt_id(url, ignore_playlist=False):

    query = urlparse(url)
    if query.hostname == 'youtu.be': return query.path[1:]
    if query.hostname in {'www.youtube.com', 'youtube.com', 'music.youtube.com'}:
        if not ignore_playlist:
            with suppress(KeyError):
                return parse_qs(query.query)['list'][0]
        if query.path == '/watch': return parse_qs(query.query)['v'][0]
        if query.path[:7] == '/watch/': return query.path.split('/')[1]
        if query.path[:7] == '/embed/': return query.path.split('/')[2]
        if query.path[:3] == '/v/': return query.path.split('/')[2]

def get_yt(url):
    try :
        response = requests.get(url)
        if response.status_code == 200:
            id = get_yt_id(url)
            transcript = YouTubeTranscriptApi.get_transcript(id)
            yt_text = ' '.join([item['text'] for item in transcript])

            return yt_text
        else:
            logging.info('URL not found. Error {}'.format(response.status_code))

    except Exception as e:
        raise CustomException(e,sys)
```

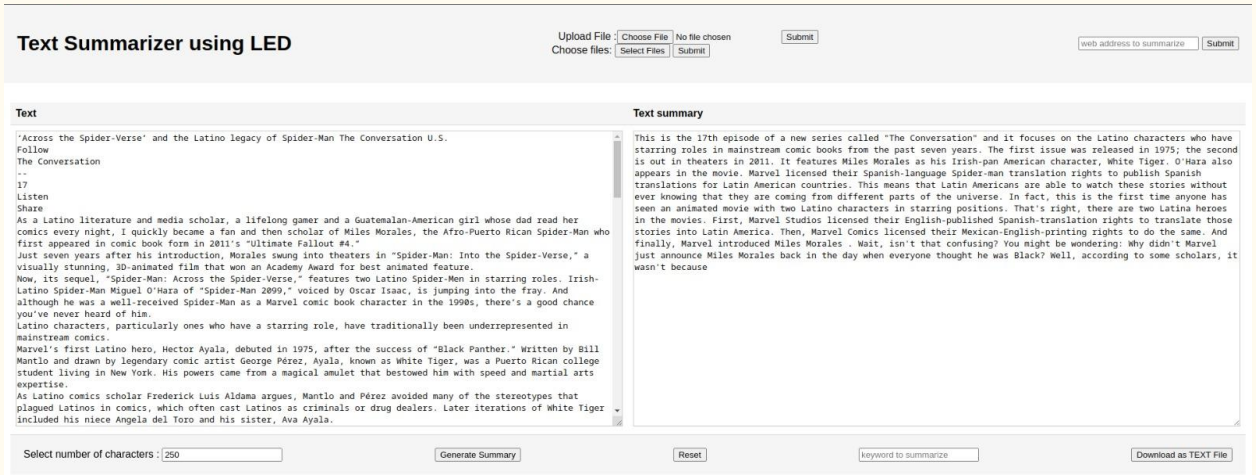
Enter your URL in the search box and click submit:



Example of Summary extracted from YouTube Transcript:



Example of Summary extracted from a Blog Post:



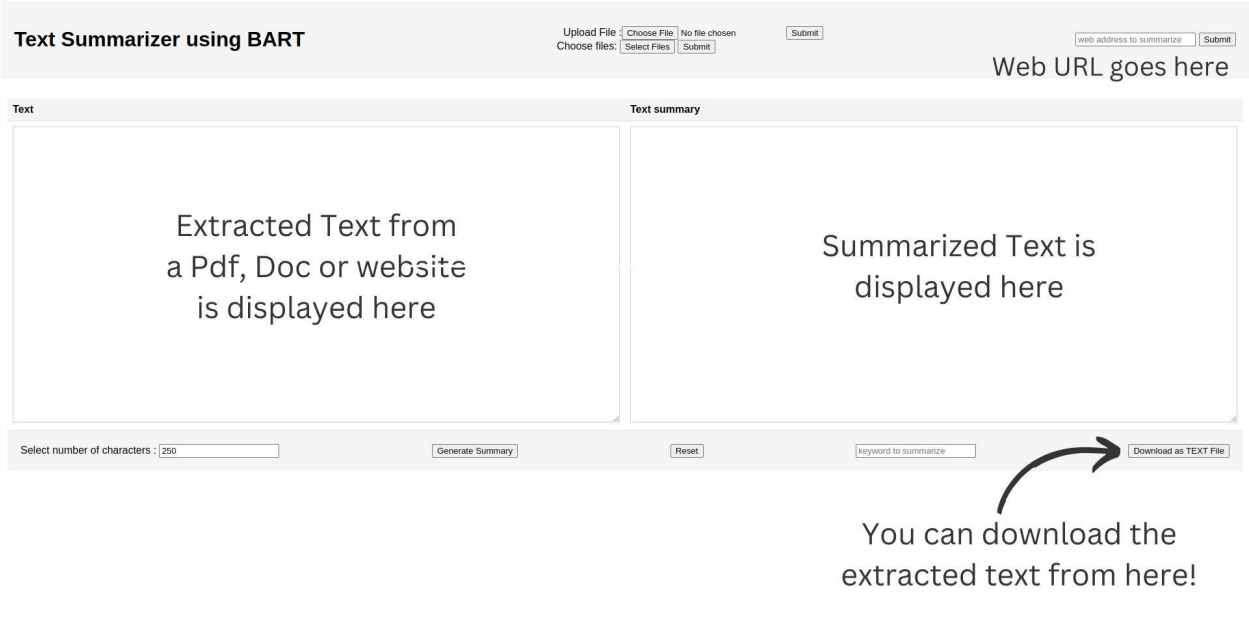
Generating Text Summaries

By combining BERT embeddings and HNSWlib for cosine KNN, a generative artificial intelligence system for text summarization can be developed. The following steps outline the process:

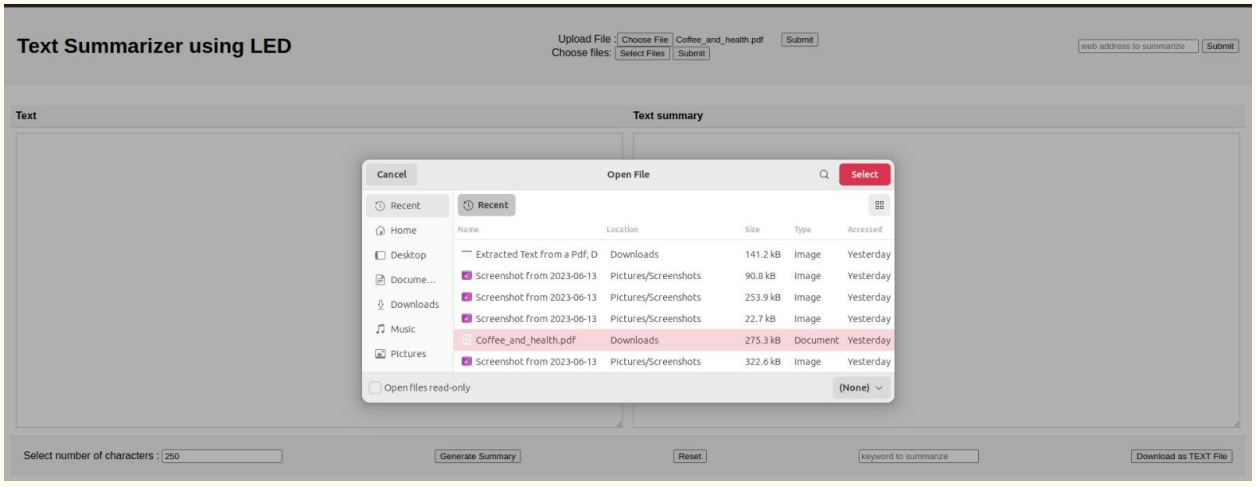
- Preprocessing: The input text, such as research papers, news articles, or documents, undergoes preprocessing to perform sentence segmentation and tokenization.
- BERT Embeddings: The preprocessed text is passed through a fine-tuned BERT model to generate embeddings that capture the contextual understanding of the input.
- HNSW Indexing: The BERT embeddings of the documents are indexed using HNSWlib to create an efficient search index.
- Cosine KNN Search: Given a query document, the BERT embeddings perform a cosine similarity-based KNN search on the HNSW index, retrieving the most similar documents.
- Retrieval: The top 20 sentences similar to the query are retrieved from the HNSWlib index and post-processed to form a document.
- Summarization: Extractive or abstractive summarization techniques are applied to the retrieved documents using LED to generate concise summaries.

User Interface

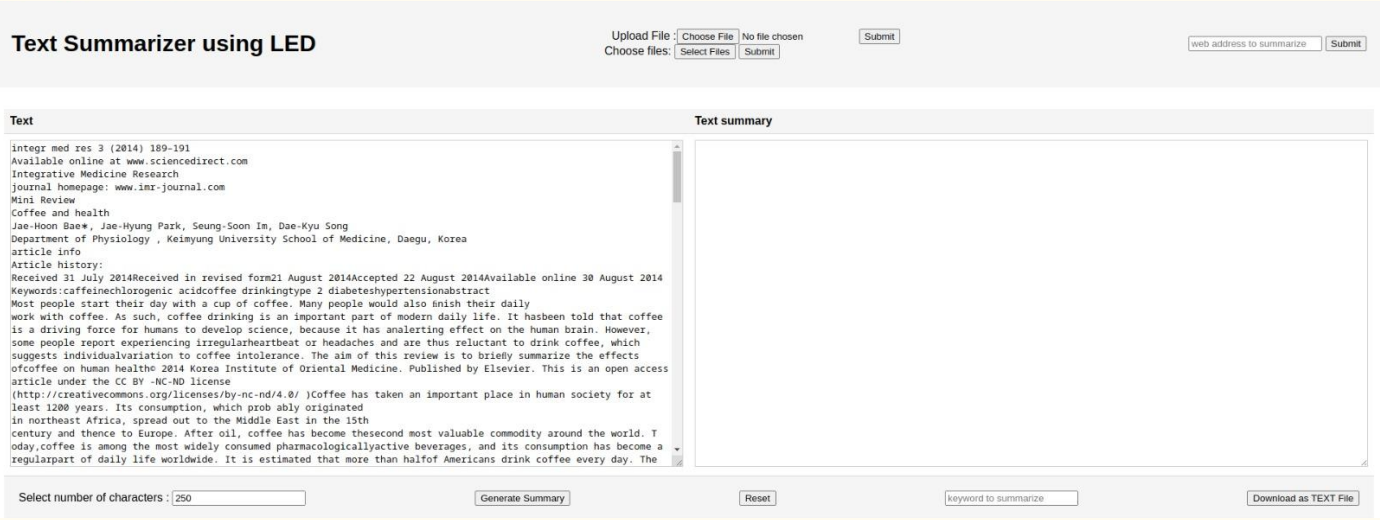
How to run: Look up the Readme file in the GitHub repo for instructions to run it. The general layout of our app is as follows :



Uploading Files :

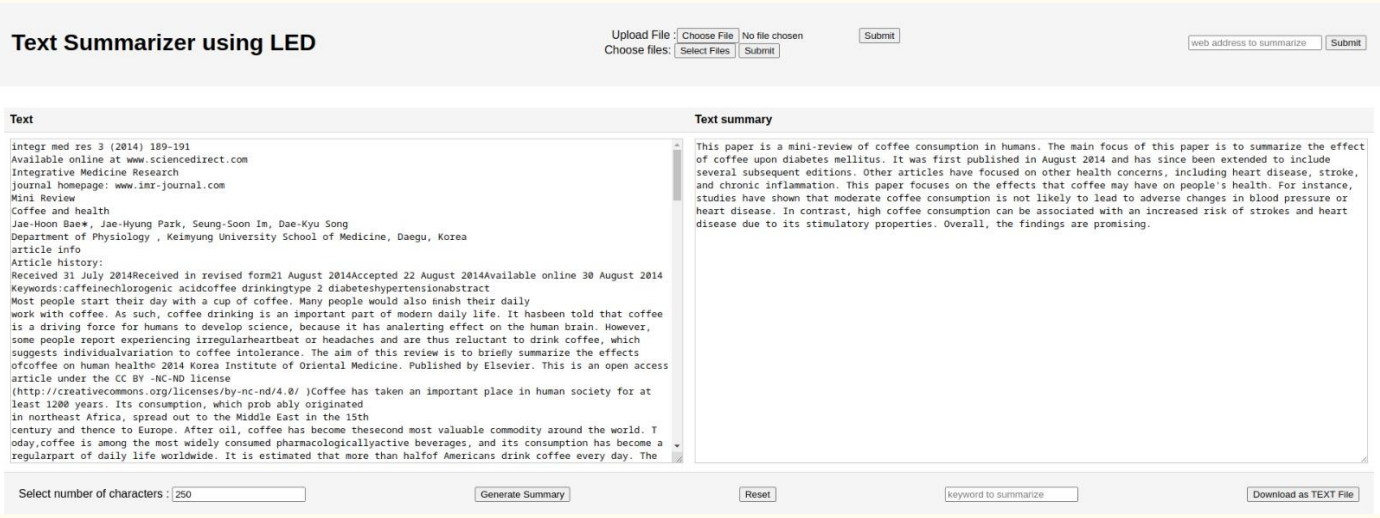


Use the choose file button to choose any PDF/DOC file from a directory. Alternatively, you can paste text in the text field. You can select multiple previously uploaded files using the “Choose Files” button.



After uploading a file, press "Submit," and the text will be extracted from it (PDF, DOC, Website, or Youtube video) and displayed in the text field. You may then specify the number of words you desire in the summary. It is set to 250 by default.

Generate Summary :



Text Summary from PDF

Press the "Generate Text" button and wait a few minutes (Performance depends on the local system's hardware). You will get the summarized text in the Text Summary field.

Also, if you want to store the summarized text in a Txt file, press the Download as TEXT button in the bottom left corner.

Future Work

With the constrained time frame, we could only implement some ideas in our minds. However, the following features can make this app better:

- **Integration of support for multiple languages:** Users can generate summaries for different languages in other languages.
- **Speech to Text:** Users can directly feed in an audio file to summarize.
- **Text to Speech:** Users who hate reading can generate audio for the summary.