Project Report Format

1. INTRODUCTION

1.1 Project Overview

"AI-Based Intelligent Insight Extractor" is an innovative project focused on harnessing the power of artificial intelligence, specifically leveraging the Pegasus-xsum model. The objective of this project is to develop an intelligent system capable of extracting meaningful insights and summarizing complex textual information. Pegasus-xsum, renowned for its prowess in abstractive text summarization, serves as the cornerstone for this model.

1.2 Purpose

The project aims to empower users with an advanced tool that automates the extraction of key insights from extensive datasets, documents, or articles. By employing cuttingedge natural language processing techniques, the AI model transforms verbose content into concise and coherent summaries, facilitating efficient comprehension and decisionmaking.

Whether applied in business intelligence, research, or content analysis, the AI-Based Intelligent Insight Extractor stands to streamline information processing, saving time and enhancing productivity. Through the utilization of state-of-the-art AI technologies, this project endeavors to deliver a sophisticated solution for extracting actionable insights from diverse textual sources.

2. LITERATURE SURVEY

2.1 Existing problem

The existing problems prompting a text summarization system in NLP include manual, timeconsuming summarization, information overload, inaccurate automated tools, lack of customization, limited language support, absence of real-time summarization, security concerns, and dependency on external services.

2.2 References

https://medium.com/analytics-vidhya/text-summarization-using-nlp-3e85ad0c6349

https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=94b806e5fbfab1d559557ceb83ca40cd806ea1a5

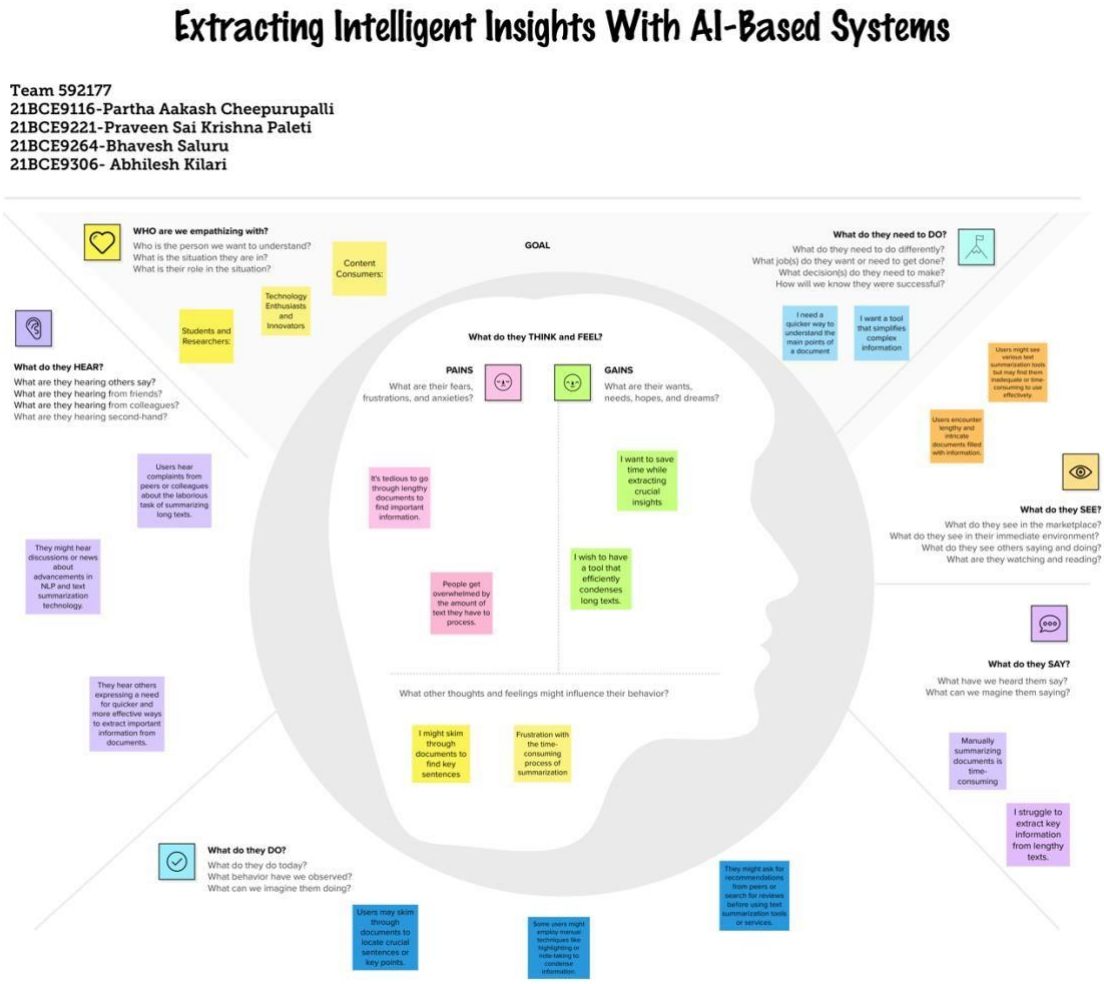
https://link.springer.com/chapter/10.1007/978-981-15-5400-1_54

2.3 Problem Statement Definition

The problem statement for a text summarization system in NLP revolves around the inefficiency of manual summarization, information overload, inaccuracies in existing automated tools, limited customization, language constraints, absence of real-time summarization, and concerns regarding security and dependency on external services. This necessitates a solution that offers accurate, customizable, and real-time text summarization while addressing security and language limitations.

3. IDEATION & PROPOSED SOLUTION

3.1 Empathy Map Canvas



3.2 Ideation & Brainstorming

1

Define your problem statement

What problem are you trying to solve? Frame your problem as a How Might We statement. This will be the focus of your brainstorm.

5 minutes

PROBLEM

How might we [your problem statement]?

Key rules of brainstorming

To run an smooth and productive session

- Stay in topic.
- Defer judgment.
- Go for volume.
- Encourage wild ideas.
- Listen to others.
- If possible, be visual.

2

Brainstorm

Write down any ideas that come to mind that address your problem statement.

10 minutes

Partha Aakash

- Keyword Extraction with Context
- Multi-document Summarization
- Machine Learning Enhanced Summarization

Praveen Sai Krishna

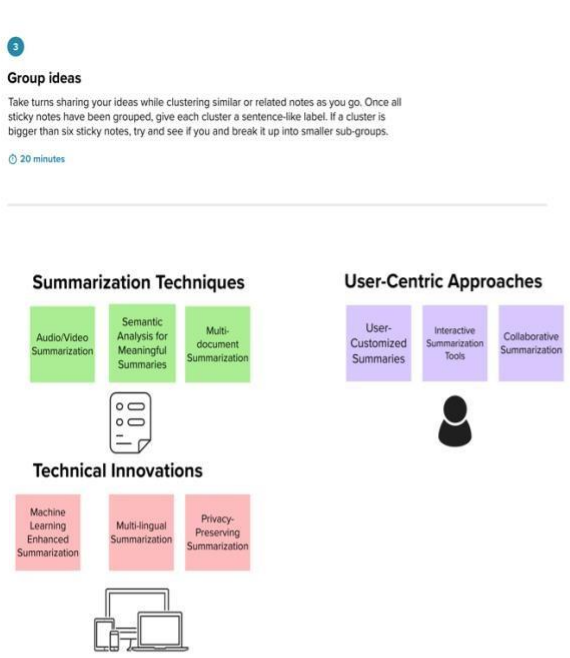
- Semantic Analysis for Meaningful Summaries
- Design interfaces that allow users to interact
- Multi-lingual Summarization

Bhavesh

- User-Customized Summaries
- Real-time Summarization
- Audio/Video Summarization

Abhilesh

- Visual Summarization Representations
- Sentiment-Driven Summaries
- Privacy-Preserving Summarization



4. **REQUIREMENT ANALYSIS**

4.1 Functional requirement

Functional requirements for a text summarization system in NLP encompass text input processing, summarization algorithms, customization options, multilingual support, realtime summarization, semantic analysis, user interaction, feedback mechanisms, integration capabilities, and robust security measures.

4.2 Non-Functional requirements

Non-functional requirements for a text summarization system involve performance, accuracy, scalability, usability, multilingual support, real-time capabilities, security, reliability, and interoperability.

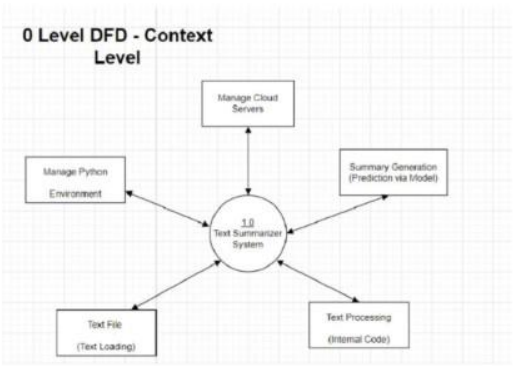
5. **PROJECT DESIGN**

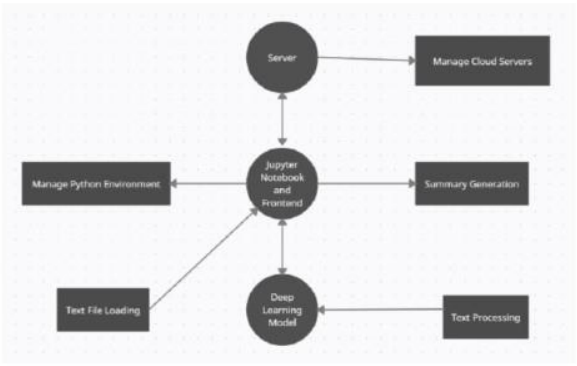
5.1 Data Flow Diagrams & User Stories

Project Design Phase-II
Data Flow Diagram & User Stories

Date	20 November 2023
Team ID	SPSGP-614965
Project Name	Project - Extracting Intelligent Insights With AI-Based Systems
Maximum Marks	2 Marks

Data Flow Diagrams:





User Stories

Use the below template to list all the user stories for the product.

User Type	Functional Requirement (Epic)	User Story Number	User Story / Task	Priority
As a Student Researcher	Summarization	USN-1	I want to customize the summary length and depth to suit my study needs.	High
As a Business Professional	Long Text Summarization	USN-2	I need a tool that summarizes reports and industry papers efficiently, saving me time in gathering crucial insights.	High
As a Content Consumer	Sentiment Analysis with Summarization	USN-3	I'm looking for a tool that provides condensed summaries of news articles, blog posts, or online content.	Low
As a Knowledge Worker		USN-4	I require a system that can summarize technical manuals, guidelines, or policy documents accurately.	Medium
As a Tech Enthusiast		USN-5	I'm interested in an API or developer tool that allows me to integrate text summarization capabilities into my applications or projects.	Low

5.2 Solution Architecture

Project Design Phase-I
Solution Architecture

Date	20 November 2023
Team ID	SPSGP-614965
Project Name	Project - Extracting Intelligent Insights With AI-Based Systems
Maximum Marks	4 Marks

Solution Architecture:

Finding the Best Tech Solution:

- Analyze existing text summarization techniques in NLP.
- Assess available NLP libraries, machine learning models, and semantic analysis tools.
- Explore methods for real-time processing and semantic understanding for accurate summarization.

Describing Structure and Characteristics:

- Outline the system's architecture, including summarization engine, semantic analysis modules, and machine learning components.
- Define interfaces for user interaction, API integrations, and data storage structures.
- Specify system behaviors for document processing, summarization, customization, and feedback incorporation.

Defining Features and Development Phases:

- Identify key features like user-customized summaries, real-time summarization, and semantic analysis.
- Develop a roadmap with phases for data collection, model training, UI development, testing, and deployment.

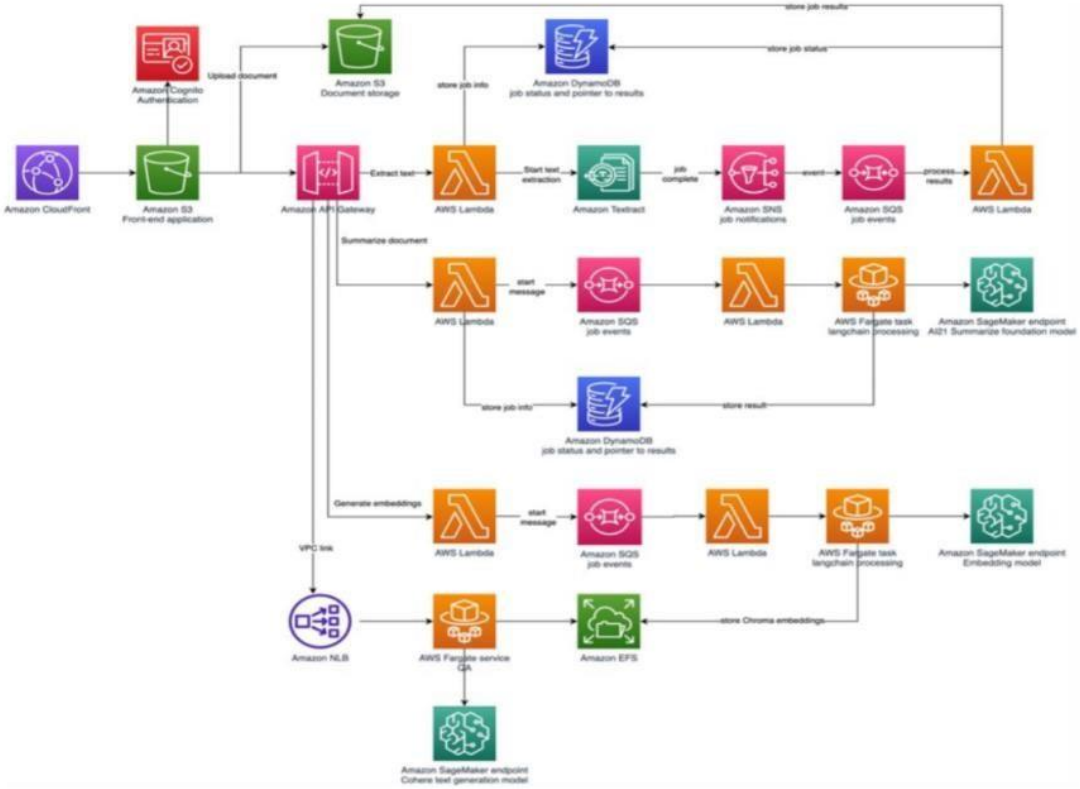
Providing Solution Requirements and Specifications:

- Define hardware requirements, such as computational resources for AI processing.
- Specify software requirements, including NLP libraries, databases for storage, and API integrations.
- Establish security protocols for data encryption, user authentication, and access controls.

Managing Solution Definition and Delivery:

- Collaborate with stakeholders to validate requirements and ensure alignment with business needs.
- Oversee development phases, ensuring adherence to specifications, quality assurance, and testing.
- Monitor project progress, manage risks, and facilitate communication between development teams and stakeholders.

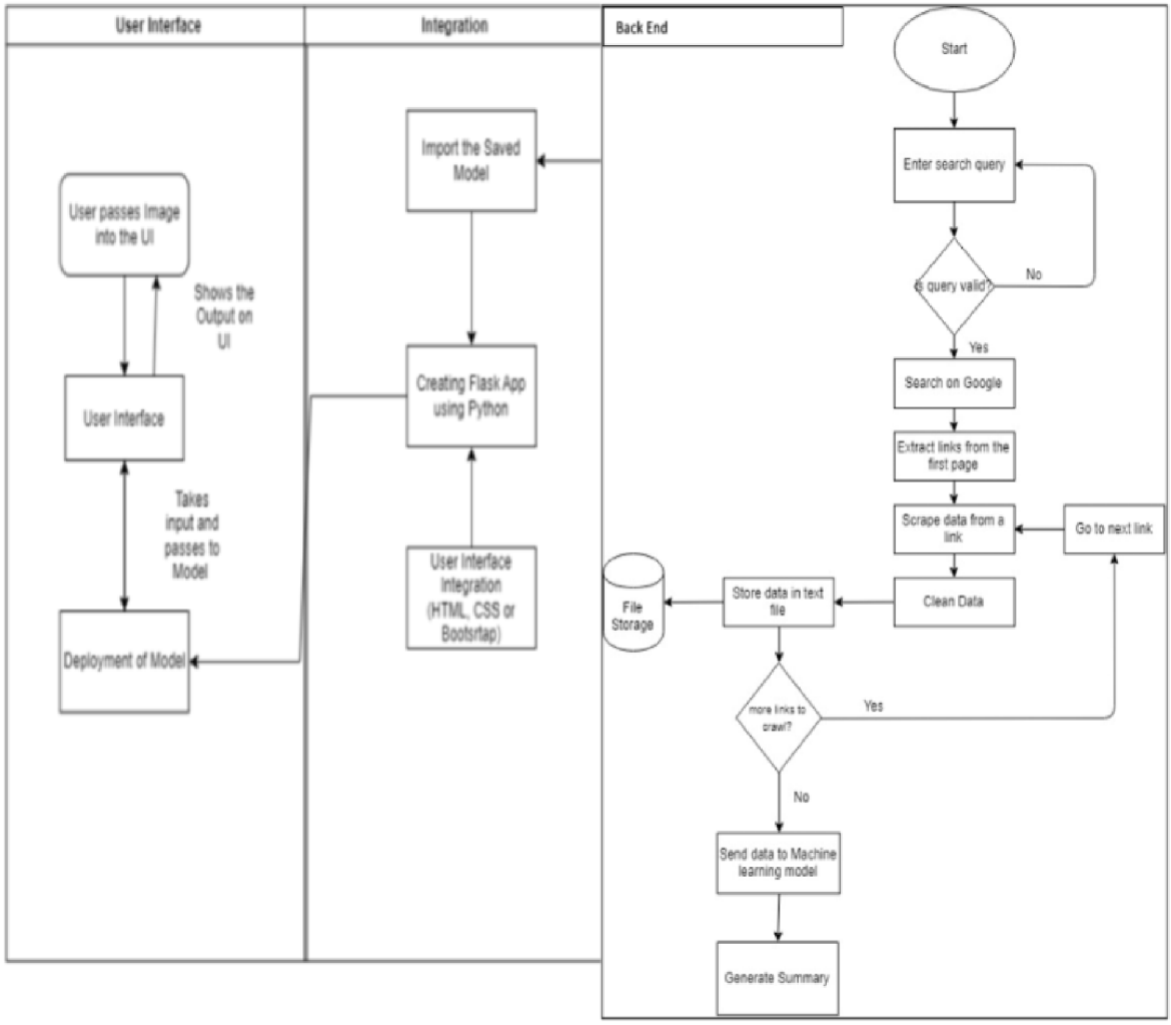
Solution Architecture Diagram:



Reference: https://aws.amazon.com/blogs/machine-learning/use-a-generative-ai-foundation-model-for-summarization-and-question-answering-using-your-own-data/

6. **PROJECT PLANNING & SCHEDULING**

6.1 Technical Architecture



6.2 Sprint Planning & Estimation

Product Backlog, Sprint Schedule, and Estimation (4 Marks)

Use the below template to create product backlog and sprint schedule

Sprint	Functional Requirement (Epic)	User Story Number	User Story / Task	Story Points	Priority	Team Members
Sprint-1	Project setup & Infrastructure	USN-1	Set up the development environment with the required tools and frameworks to start the Extracting Intelligent Insights With AI-Based Systems project.	1	High	Praveen
Sprint-1	development environment	USN-2	Gather a diverse dataset of text for training the deep learning model.	2	High	Bhavesh
Sprint-2	Data collection	USN-3	Pre-process the collected dataset splitting it into training and validation sets.	2	High	Praveen
Sprint-2	data pre-processing	USN-4	Explore and evaluate different deep learning architectures (e.g., CNNs) to select the most suitable model for text summarization.	3	High	Aakash
Sprint-3	model development	USN-5	train the selected deep learning model using the pre-processed dataset and monitor its performance on the validation set.	4	High	Abhilesh
Sprint-3	Training	USN-6	implement data augmentation techniques (e.g., rotation, flipping) to improve the model's robustness and accuracy.	6	medium	Bhavesh
Sprint-4	model deployment & Integration	USN-7	deploy the trained deep learning model as a web service to make it accessible for users. integrate the model into an user-friendly web interface for users to upload text and url and receive summarization results.	1	medium	Aakash
Sprint-5	Testing & quality assurance	USN-8	conduct thorough testing of the model and web interface to identify and report any issues or bugs. fine-tune the model hyperparameters and optimize its performance based on user feedback and testing results.	1	medium	Abhilesh

6.3 Sprint Delivery Schedule

Project Tracker, Velocity & Burndown Chart: (4 Marks)

Sprint	Total Story Points	Duration	Sprint Start Date	Sprint End Date (Planned)	Story Points Completed (as on Planned End Date)	Sprint Release Date (Actual)
Sprint-1	3	4 Days	01 Nov 2023	05 Nov 2023	20	05 Nov 2023
Sprint-2	5	4 Days	06 Nov 2023	10 Nov 2023		
Sprint-3	10	2 Days	11 Nov 2023	13 Nov 2023		
Sprint-4	1	4 Days	14 Nov 2023	18 Nov 2023		
Sprint - 5	1	2 Days	18 Nov 2023	20 Nov 2023		

7. CODING & SOLUTIONING (Explain the features added in the project along with code)

7.1 Feature 1

Milestone 1: Import Required Libraries

Install necessary packages and import the necessary libraries as shown in the figure.

```
!pip install transformers
!pip install sentencepiece
```

```
[2] from transformers import PegasusForConditionalGeneration, AutoTokenizer
    import torch
    from bs4 import BeautifulSoup
    import requests
```

- **PegasusForConditionalGeneration**: This is a class from the Hugging Face Transformers library.
- **AutoTokenizer**: Another class from Hugging Face Transformers.
- **torch**: This is PyTorch, an open-source machine learning library.
- **BeautifulSoup**: This is a library for pulling data out of HTML and XML files.
- **requests**: This is a simple HTTP library for making requests to a specified URL.

Activity 1: Read Dataset

In this project, our input is text data or url .

A variable ‘URL’ is created and the URL is passed to that variable as the data.

```
URL="https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04"
```

Milestone 2: Text Pre-Processing

In natural language processing, text pre-processing is the practice of cleaning and preparing the data.

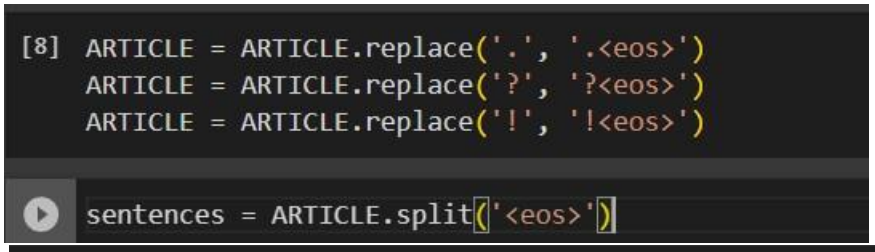
Activity-1: Accessing data from the URL

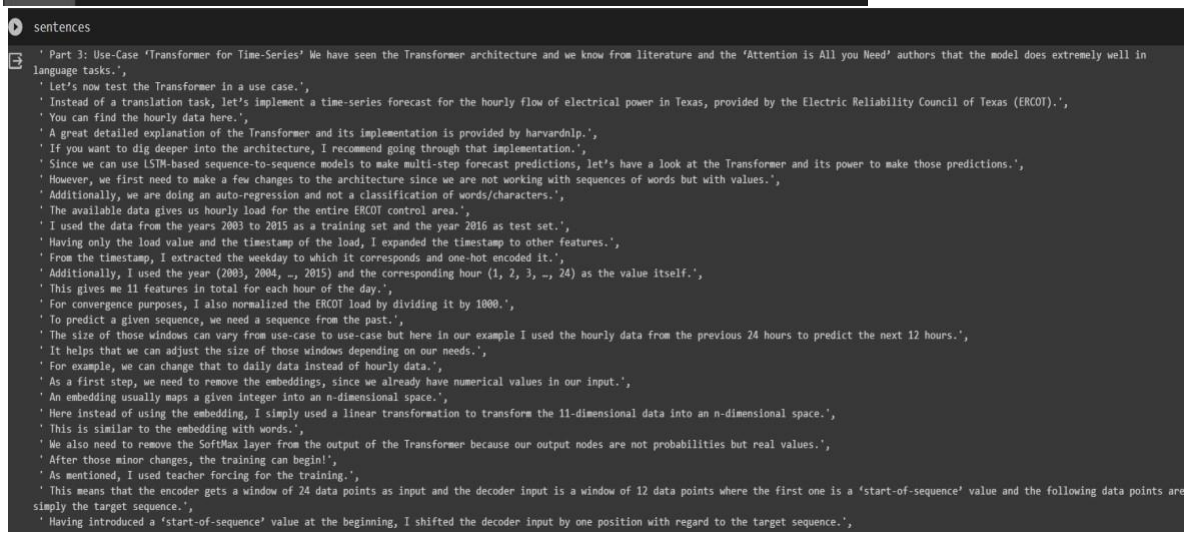
```
[3] URL="https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04"

[5] r = requests.get(URL)

    soup = BeautifulSoup(r.text, 'html.parser')
    results = soup.find_all(['h1', 'p'])
    text = [result.text for result in results]
    ARTICLE = ' '.join(text)
```

- **r = requests.get(URL)**: This line sends an HTTP GET request to the specified **URL** and stores the server's response in the variable **r**. It uses the **requests** library for making HTTP requests.
- **soup = BeautifulSoup(r.text, 'html.parser')**: This line creates a BeautifulSoup object named **soup** by parsing the HTML content of the server's response (**r.text**). The **'html.parser'** argument specifies the parser to be used for parsing HTML.
- **results = soup.find_all(['h1', 'p'])**: This line uses the **find_all** method of the BeautifulSoup object to find all HTML elements that are either **h1** (heading level 1) or **p** (paragraph) tags. The results are stored in the **results** variable as a list.
- **text = [result.text for result in results]**: This line creates a list called **text** by extracting the text content of each HTML element in the **results** list using a list comprehension.
- **ARTICLE = ' '.join(text)**: This line joins the elements in the **text** list into a single string, separated by a space. The resulting string is assigned to the variable **ARTICLE**, representing the combined text content of all the **h1** and **p** elements found on the webpage.

Activity-2: Splitting our data into sentences

We are splitting our data into sentences by adding <eos> and splitting by it





Activity-3: Combining sentences into chunks

Our model can take input as of maximum of 1024 tokens so we are dividing into chunks and giving maximum chunk size as 400

```
max_chunk = 400
current_chunk = 0
chunks = []
for sentence in sentences:
    if len(chunks) == current_chunk + 1:
        if len(chunks[current_chunk]) + len(sentence.split(' ')) <= max_chunk:
            chunks[current_chunk].extend(sentence.split(' '))
        else:
            current_chunk += 1
            chunks.append(sentence.split(' '))
    else:
        print(current_chunk)
        chunks.append(sentence.split(' '))
```

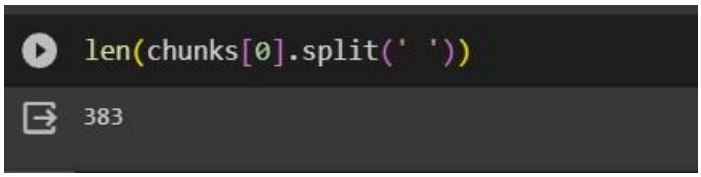
This code chunk is designed to break a list of sentences into chunks with a maximum word limit (**max_chunk=400**). It iterates through the sentences, creating or extending chunks based on the word count. If adding a sentence to the current chunk exceeds the limit, it starts a new chunk. The result is a list of chunks, each containing a subset of the original sentences within the specified word limit.

```
chunks
```

```
'shows',
'that',
'the',
'more',
'steps',
'we',
'want',
'to',
'forecast',
'the',
'higher',
'the',
'error',
'will',
'become.',
'',
'The',
'first',
'graph',
'(Figure',
'3)',
'above',
'has',
'been',
'achieved',
'by',
'using',
'the',
'24',
'hours',
'to',
```

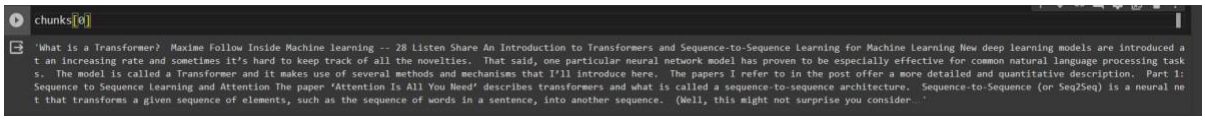
```
for chunk_id in range(len(chunks)):
    chunks[chunk_id] = ' '.join(chunks[chunk_id])
```

This code iterates through each chunk in the list of chunks. For each chunk, it uses the **join** method to concatenate the individual words within the chunk into a single string. The resulting string is then assigned back to the corresponding index in the **chunks** list. Essentially, it transforms each chunk from a list of words into a space-separated string representation of the chunk. After this loop, the **chunks** list contains strings instead of lists of words



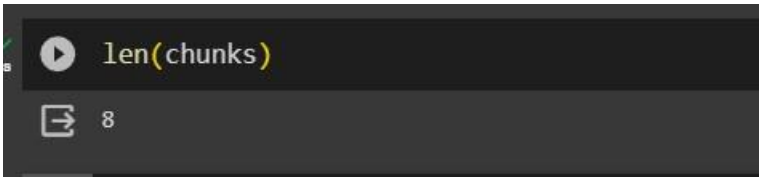
The length of chunk is always less than 400 i.e max_chunk size given

For instance, length of a chunk in index 0 is 383



This is the chunk at index 0

Similarly, our data is divided like this into 8 chunks



So now our whole data is present at chunk divided into 8 paras'

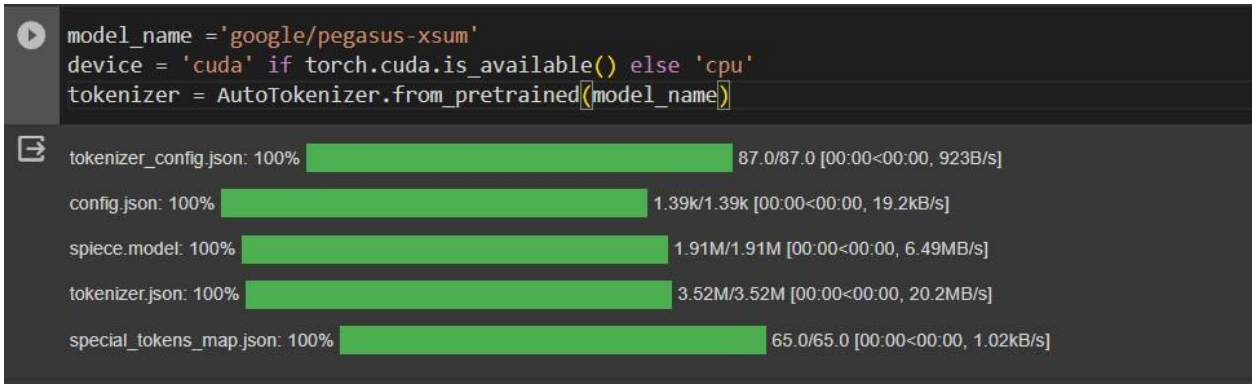
7.2 Feature 2

Milestone 3: Model Building

We want to build our model in such a way that it can interpret our whole data and summarise the data by generating new sentences which can be done by using Huggingface Transformer Pegasus-xsum.

"Pegasus-xsum" is a pre-trained model developed by Google as part of the Pegasus family. Specifically designed for abstractive text summarization, it excels at generating concise and coherent summaries of longer documents. Trained on large datasets, Pegasus-xsum employs a transformer architecture to understand and produce humanlike summaries, making it a powerful tool for various natural language processing tasks where summarization is crucial.

Activity 1: Setting up environment for initializing the model

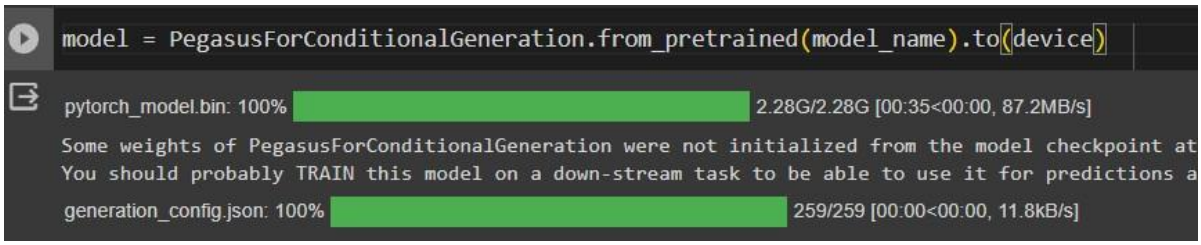


1. **model_name ='google/pegasus-xsum'**: Specifies the name of the Pegasus model to be used, specifically the "pegasus-xsum" model from Google. This model is trained for abstractive text summarization.

2. **device = 'cuda' if torch.cuda.is_available() else 'cpu'**: Determines the device for computation. If a CUDA-compatible GPU is available, it sets the device to 'cuda' (GPU); otherwise, it sets it to 'cpu' (CPU). This is useful for leveraging GPU acceleration if it's available.

3. **tokenizer = AutoTokenizer.from_pretrained(model_name)**: Initializes a tokenizer using the **AutoTokenizer** class from Hugging Face Transformers. It automatically selects the appropriate tokenizer for the specified Pegasus model (**model_name**). The tokenizer is responsible for converting text into tokens that the model can process.

Activity 2: Initialising the model



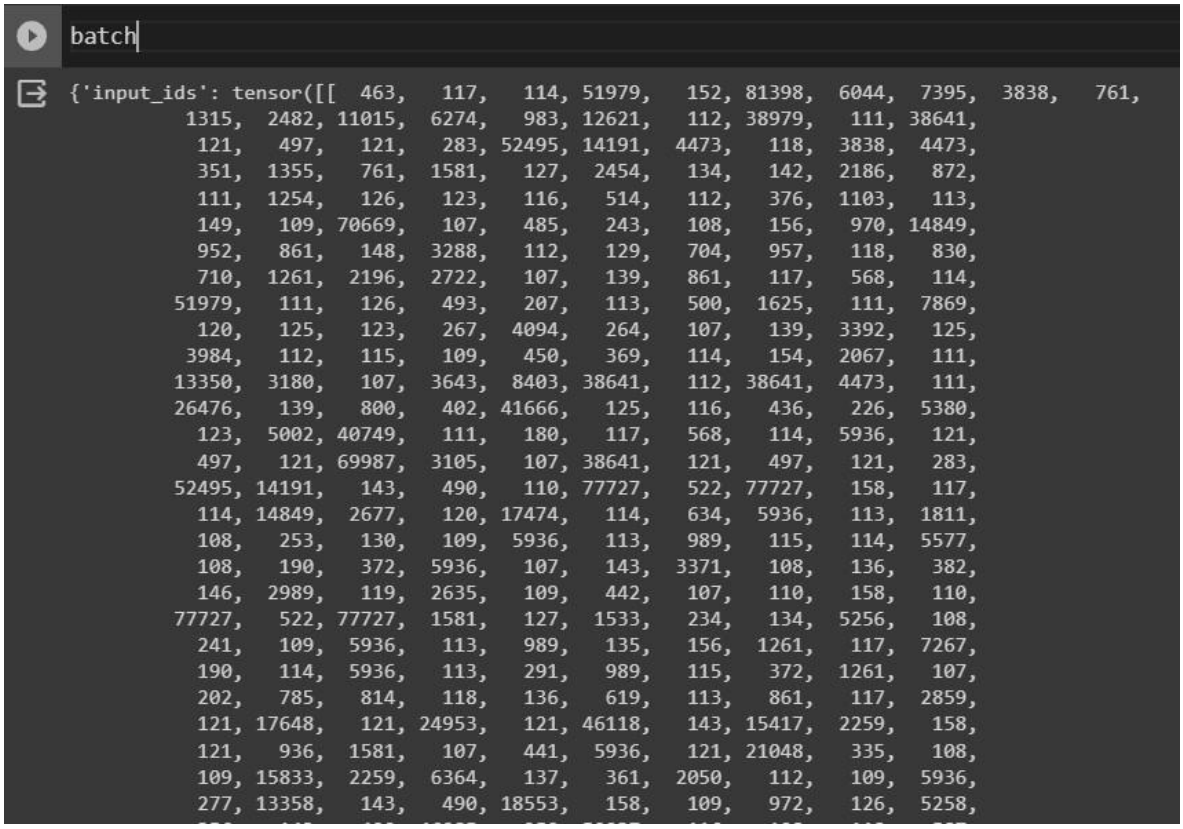
1. **model = PegasusForConditionalGeneration.from_pretrained(model_name)**: Initializes a Pegasus model for conditional text generation using the **PegasusForConditionalGeneration** class from the Hugging Face Transformers library. The **from_pretrained** method loads the pre-trained weights and architecture specified by the **model_name** ('google/pegasus-xsum').

2. **.to(device)**: Moves the model to the specified computing device. If a CUDAcompatible GPU is available, it will be moved to 'cuda' (GPU); otherwise, it will be moved to 'cpu' (CPU).

Activity 3: Tokenization

```
batch = tokenizer(chunks, truncation=True, padding='longest', return_tensors="pt").to(device)
```

this line tokenizes and processes the text in the chunk, applying truncation and padding, and then converts the result into PyTorch tensors, ensuring that the data is on the correct computing device. The processed batch is stored in the variable **batch**.



Activity 4: Get the summary tokens from the model

- **model.generate(**batch)**: Calls the **generate** method of the Pegasus model to generate text based on the input batch. The **batch** contains tokenized and processed input text. The **generate** method utilizes the model to produce the corresponding output, which, in this case, is the generated translation.

The resulting translated text is stored in the variable **translated**.

Output translated tokens

```
translated
```

```
tensor([[    0,   222,   136,   450,   108,   125,   123,   208,   313,   112,
          4094,   119,   112,   114,   177,   619,   113,  1355,   761,   861,
           107,     1,     0,     0,     0,     0,     0,     0,     0,     0,
             0],
        [    0,   184,   123,   261,   506,  1673,   120,   114,  1157,  5256,
           861,   568,   110, 77727,   522, 77727,   137,   516,   142, 23347,
          1261,   107,     1,     0,     0,     0,     0,     0,     0,     0,
             0],
        [    0,   222,   136,   800,   108,   145,   799,   114,   177,   861,
           118,  1546,   121,   497,   121, 69987,  5256,   568, 51979,   107,
             1,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0],
        [    0,   321,   109,  1090,  4352,   120,   117,   646,   190,   728,
           109, 40753,   111,   109, 44129, 14012,   108,  1328,   117,   291,
           135,   109,  5936,  4543,   141,  2706,   107,     1,     0,     0,
             0],
        [    0,   222,   136,   821,   108,   125,   123,   208,   313,   112,
           403,   119,   199,   112,  3460,   109, 44129,  3196,  5936,   333,
           569,   107,     1,     0,     0,     0,     0,     0,     0,     0,
             0],
        [    0,   222,   109,   453,   297,   113,   161,   679,   124,  5936,
           121,   497,   121, 69987,  1581,   108,   125,   346,   313,   112,
           403,   119,   199,   112,  1976,   111,  3395,   109, 51979,   107,
             1],
        [    0,   222,   136,   587,   108,   125,   263,  2118, 10596,   112,
          1976,   109, 51979,   112,  7582,   109,   352,   665,   539,   107,
             1,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0],
        [    0,   222,   136,   821,   108,   125,   123,   208,   313,   112,
           403,   119,   199,   112,  1976,   114,   861,   112,  7582,   109,
           352,   665,   539,   113,   114,   166,   121, 17774,   107,     1,
             0]])
```

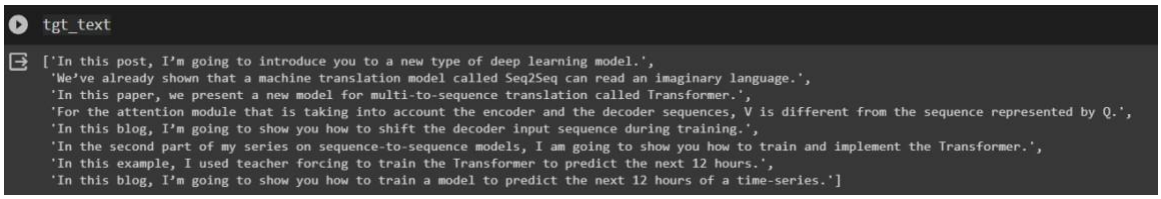
Activity 5: Decoding the summary tokens

```
tgt_text = tokenizer.batch_decode(translated, skip_special_tokens=True)
```

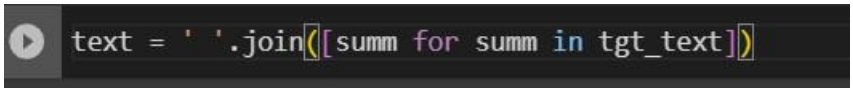
This line of code performs the decoding of the generated tokens back into humanreadable text:

- **tokenizer.batch_decode(translated, skip_special_tokens=True)**: Utilizes the tokenizer's **batch_decode** method to convert the generated tokens (**translated**) into a list of strings. The **skip_special_tokens=True** parameter instructs the tokenizer to skip any special tokens (e.g., padding tokens) during the decoding process.

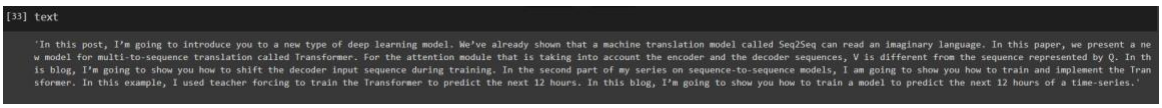
The resulting list of strings represents the decoded, human-readable text and is stored in the variable **tgt_text**. Each element in the list corresponds to the generated translation for a specific input sequence or chunk of text.



Activity 6: Summary



This line of code creates a single string (**text**) by joining together individual translations stored in the **tgt_text** list. Each translation is separated by a space in the concatenated string.



This is the final summary we got from the model.

Milestone 4: Application Building

In this section, we will be building a web application that is integrated into the model we built. A UI is provided for the uses where he/she has to enter the text for a summary. Then the summary is showcased on the UI.

This section has the following tasks

- Building HTML Pages
- Building server-side script

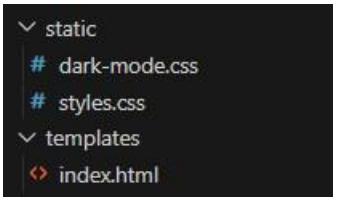
Activity-1: Building Html Pages

For this project create one HTML file and two CSS files namely

- index.html
- styles.css
- dark-mode.css

and save them in the templates folder and static folder respectively.

HTML files come under templates folder and CSS files comes under static folder



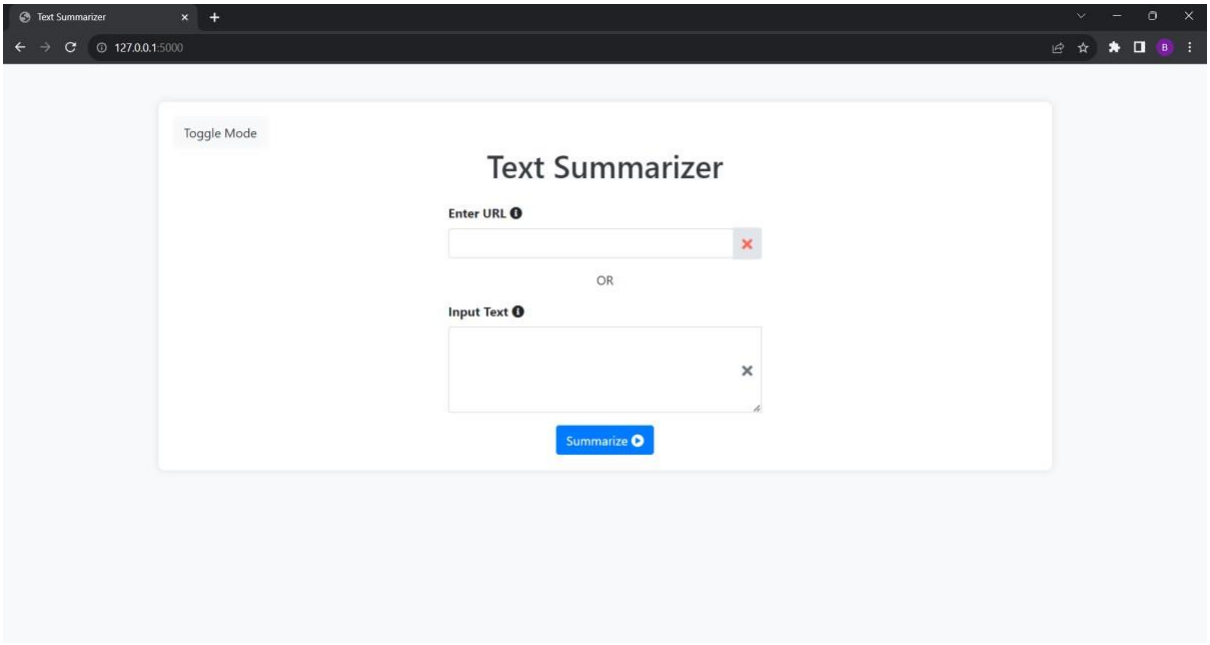
The main focus of the app is to keep it simple, clean and functional.

The Web app has two modes, namely, light mode and dark mode.

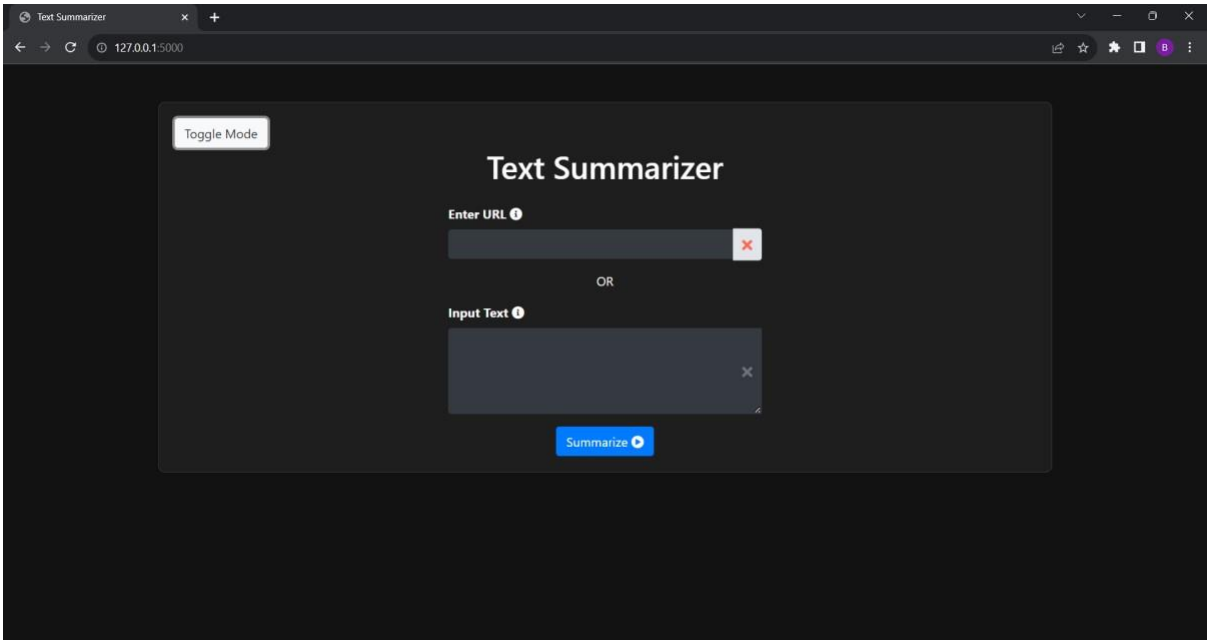
The UI is kept as simple as possible to avoid any confusions and efforts were made to enhance the UI by providing hints so as to make it easy and understandable to use.

Let’s see how our web page looks like:

Light Mode:



Dark Mode:

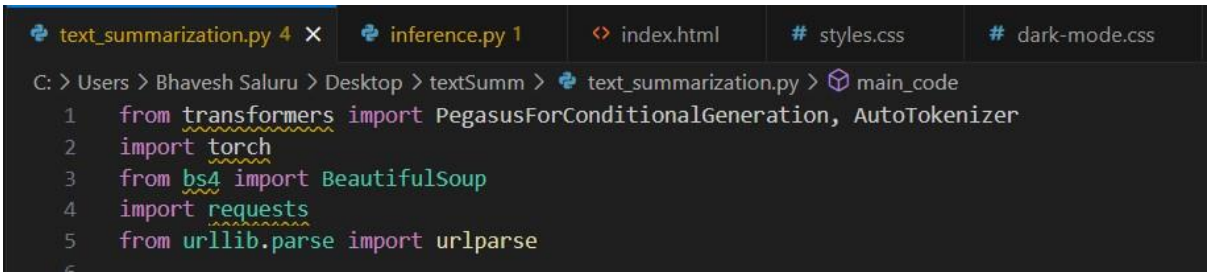


Activity 2: Build Python code

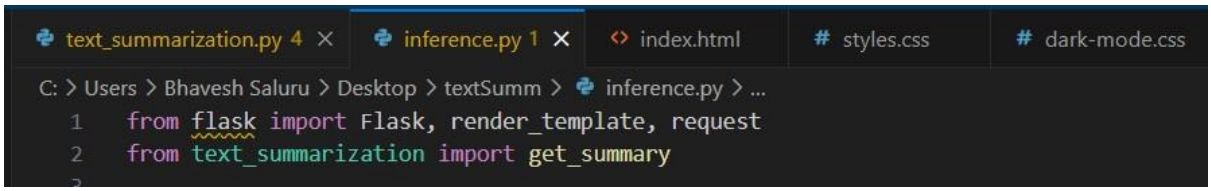
Import the libraries

We will be having two python files, one for the model itself and one for the inference part

In text_summarization.py, we import the following libraries:



In inference.py, we import the following libraries:



Importing the flask module into the project is mandatory. An object of the Flask class is our Web Application. Flask constructor takes the name of the current module (__name__) as an argument

```
app = Flask(__name__)
```

Render HTML page: Here we will be using the declared constructor to route to the HTML page that we have created earlier. In the above example, the '/' URL is bound with the index.html function. Hence, when the home page of the web server is opened in the browser, the HTML page will be rendered. Whenever you enter the values from the HTML page the values can be retrieved using the GET Method and whenever you enter the values to the HTML page the values can be sent using the POST Method.

In our case, we will be checking if the text retrieved from the html i.e., from user is from url text field or plain data text field and handle the data accordingly with the below index function.

```
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        url = request.form['url']
        text_input = request.form.get('text_input')
        if url:
            summary = get_summary(url)
        elif text_input:
            summary = get_summary(text_input)
        else:
            summary = None
        return render_template('index.html', summary=summary)
    return render_template('index.html', summary=None)

if __name__ == '__main__':
    app.run(debug=True)
```

Once we fetch the data, we will be calling the get_summary (data) function from our main tex_summarizatin.py and return and render the output received rom the model to the html file/user.

Here's the overview of what happens when the data is passed to the get_summary (data) function

First, the data is checked if it is url or plain text

```
def get_summary(url):
    if not is_valid_url(url):
        return main_code(url)

    URL = url

    r = requests.get(URL)

    soup = BeautifulSoup(r.text, 'html.parser')
    results = soup.find_all(['h1', 'p'])
    text = [result.text for result in results]
    data = ' '.join(text)

    return main_code(data)
```

The is_valid_url(url) function return whether or not the data is url

```
def is_valid_url(url):
    try:
        result = urlparse(url)
        return all([result.scheme, result.netloc])
    except ValueError:
        return False
```

Once the Boolean value is received, the get_summary() function processes and fetches the data if it is url and then pass it to main_code(data) function where the actual summarization is done, if not, the data is directly passed to main_code(data) function for the summary.

```python
def main_code(data):
    ARTICLE = data

    ARTICLE = ARTICLE.replace('.', '.<eos>')
    ARTICLE = ARTICLE.replace('?', '?<eos>')
    ARTICLE = ARTICLE.replace('!', '!<eos>')

    sentences = ARTICLE.split('<eos>')

    max_chunk = 400
    current_chunk = 0
    chunks = []
    for sentence in sentences:
        if len(chunks) == current_chunk + 1:
            if len(chunks[current_chunk]) + len(sentence.split(' ')) <= max_chunk:
                chunks[current_chunk].extend(sentence.split(' '))
            else:
                current_chunk += 1
                chunks.append(sentence.split(' '))
        else:
            chunks.append(sentence.split(' '))

    for chunk_id in range(len(chunks)):
        chunks[chunk_id] = ' '.join(chunks[chunk_id])

    model_name ='google/pegasus-xsum'
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = PegasusForConditionalGeneration.from_pretrained(model_name).to(device)
    batch = tokenizer(chunks, truncation=True, padding='longest', return_tensors="pt").to(device)
    translated = model.generate(**batch)
    tgt_text = tokenizer.batch_decode(translated, skip_special_tokens=True)

    text = ' '.join([summ for summ in tgt_text])

    return text
```

The main_code(data) function first pre-processes the data and then divide it into chunks of data of size 400 for better and faster results.

Once the chunks are ready, tokenization is done followed by the model summarizes each chunk and combines all the summaries produced into one combined and final summary.

In addition, the script uses any dedicated graphic card, if available.

Now this final summary is returned to the get_summary() function and the get_summary() functions returns the same to our inference.py and that indeed returns the text to the html, i.e., user with clean UI.

Activity 3: Run the application

- Open the anaconda/command prompt from the start menu
- Navigate to the folder where your python script is
- Now activate the virtual environment
- Now type the “python inference.py” command
- Navigate to the localhost to view the web app
- Enter the inputs(url/text), click on the summarize button and see the result/summary on the same web page.

8. PERFORMANCE TESTING

8.1 Performace Metrics

Model Performance Testing:

```
print(model)
```

```
PegasusForConditionalGeneration(
  (model): PegasusModel(
    (shared): Embedding(96103, 1024, padding_idx=0)
    (encoder): PegasusEncoder(
      (embed_tokens): Embedding(96103, 1024, padding_idx=0)
      (embed_positions): PegasusSinusoidalPositionalEmbedding(512, 1024)
      (layers): ModuleList(
        (0-15): 16 x PegasusEncoderLayer(
          (self_attn): PegasusAttention(
            (k_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (v_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (q_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (out_proj): Linear(in_features=1024, out_features=1024, bias=True)
          )
          (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
          (activation_fn): ReLU()
          (fc1): Linear(in_features=1024, out_features=4096, bias=True)
          (fc2): Linear(in_features=4096, out_features=1024, bias=True)
          (final_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
        )
      )
      (layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
    )
    (decoder): PegasusDecoder(
      (embed_tokens): Embedding(96103, 1024, padding_idx=0)
      (embed_positions): PegasusSinusoidalPositionalEmbedding(512, 1024)
      (layers): ModuleList(
        (0-15): 16 x PegasusDecoderLayer(
          (self_attn): PegasusAttention(
            (k_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (v_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (q_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (out_proj): Linear(in_features=1024, out_features=1024, bias=True)
          )
          (activation_fn): ReLU()
          (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
          (encoder_attn): PegasusAttention(
            (k_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (v_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (q_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (out_proj): Linear(in_features=1024, out_features=1024, bias=True)
          )
          (encoder_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
          (fc1): Linear(in_features=1024, out_features=4096, bias=True)
          (fc2): Linear(in_features=4096, out_features=1024, bias=True)
          (final_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
        )
      )
      (layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
    )
  )
  (lm_head): Linear(in_features=1024, out_features=96103, bias=False)
)
```

```
[29] total_params = sum(p.numel() for p in model.parameters())
     print(f"Total parameters: {total_params}")

     Total parameters: 569748480
```

```
rouge_names = ["rouge1", "rouge2", "rougeL", "rougeLsum"]

reference = dataset['train'][1]['document']

records = []

for model_name in summaries:
    rouge_metric.add(prediction = summaries[model_name], reference = reference )
    score = rouge_metric.compute()
    rouge_dict = dict((rn, score[rn].mid.fmeasure ) for rn in rouge_names )
    print('rouge_dict ', rouge_dict )
    records.append(rouge_dict)

pd.DataFrame.from_records(records, index = summaries.keys() )
```

```
rouge_dict  {'rouge1': 0.10526315789473684, 'rouge2': 0.011834319526627219, 'rougeL': 0.0584795321637427, 'rougeLsum': 0.10526315789473684}
```

	rouge1	rouge2	rougeL	rougeLsum
pegasus	0.105263	0.011834	0.05848	0.105263

9. **RESULTS**

9.1 Output Screenshots

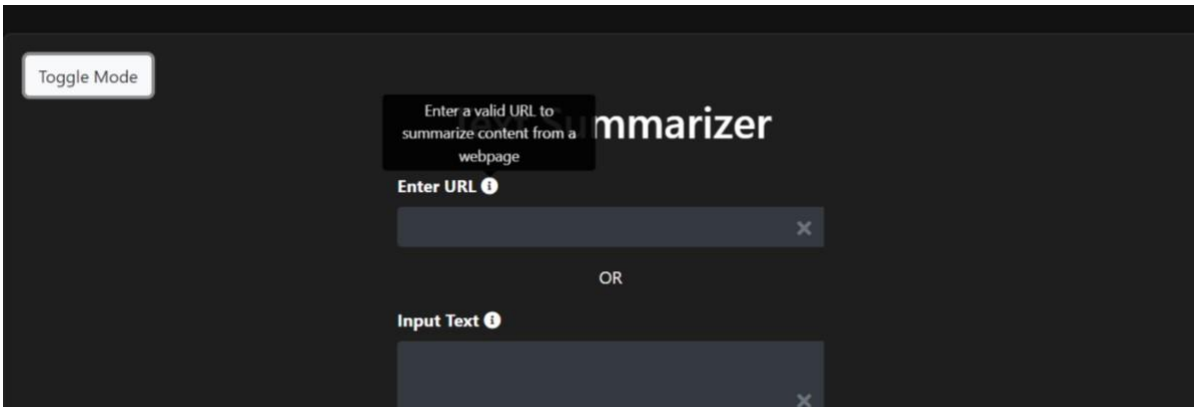
Output:

Here's the overview of how the web app looks and works:

User has two options to provide the text that needs to be summarized.

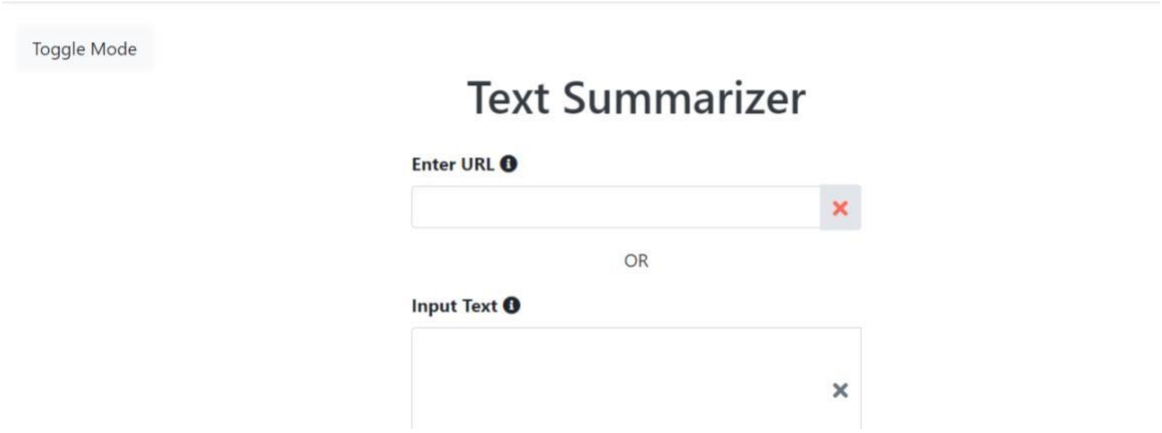
Option one:

In the first field, user is required to enter/provide the url, an about button is provided next to the text field, to provide more details about the component.



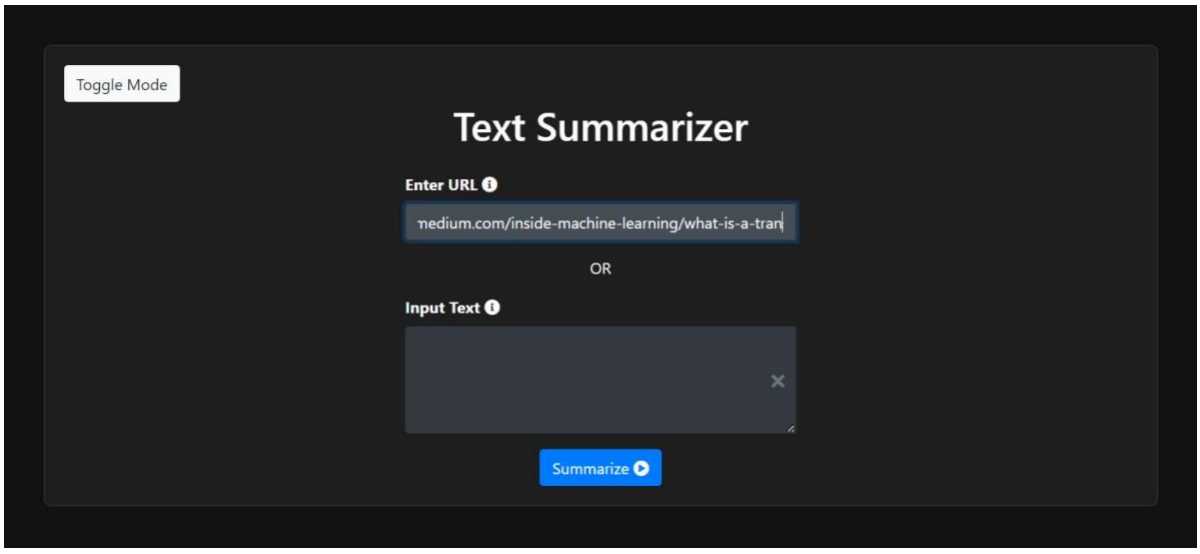
Option two:

In the second field, user is required to enter/provide the text, an about button is provided next to the text field, to provide more details about the component.
If the user intends to clear the text in the field, one can click on the cancel button next to the input text field

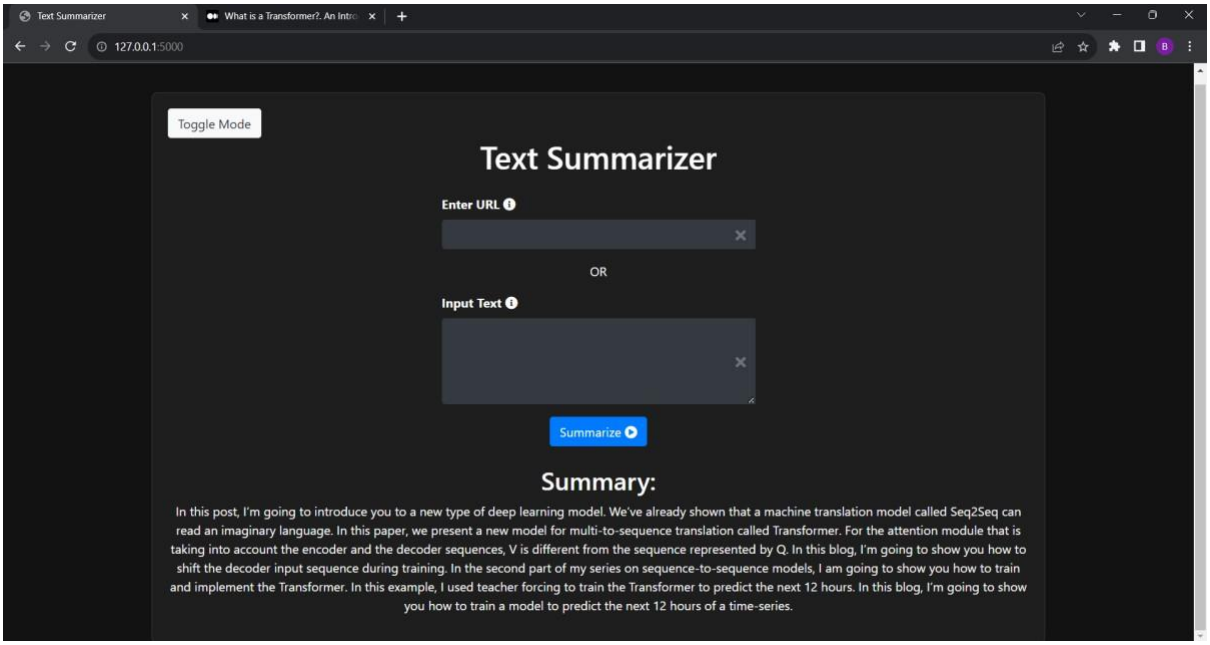


Once the text is entered, one can click summarize and wait for the summary of the text.

For example, when the url https://medium.com/inside-machine-learning/what-is-atransformer-d07dd1fbec04 is provided,

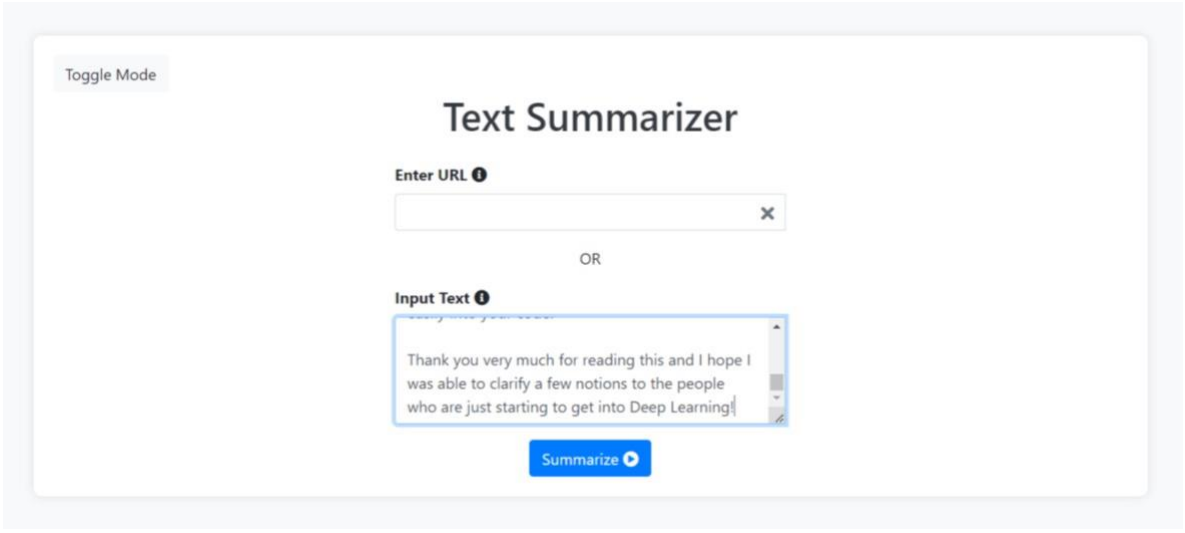


The output/summary looks as follows:

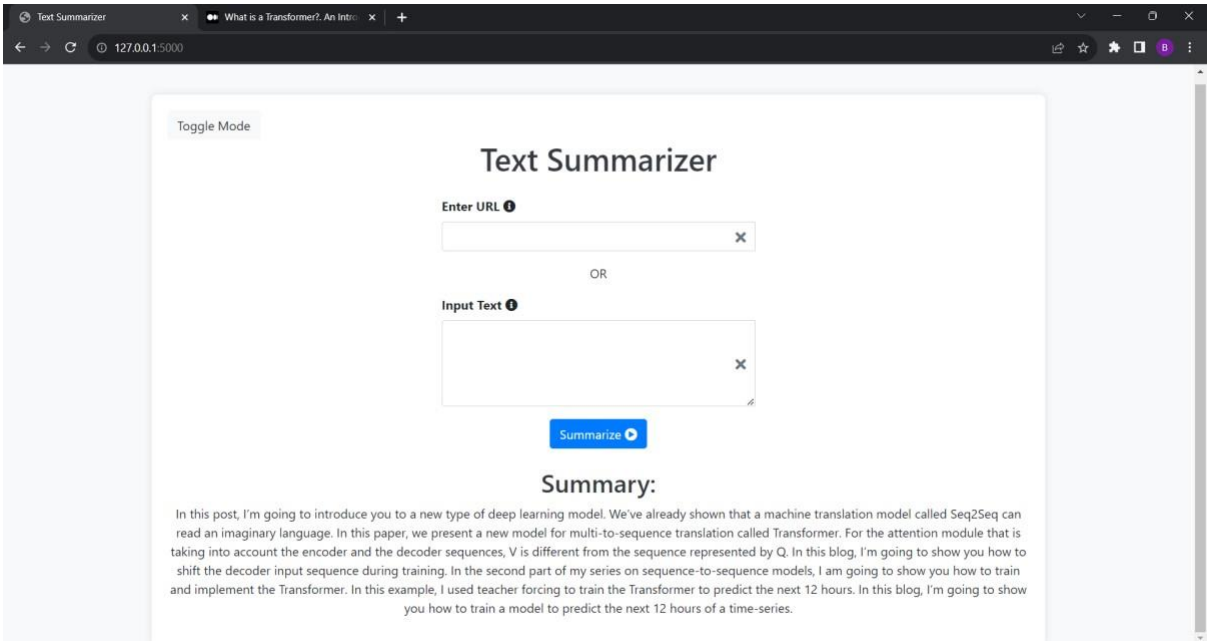


The same can be done on direct text as well, which generates the same output, testing the same in light mode this time instead of dark to show the proper working of the UI

Let's say, I give the same text from the same website directly to the app, it looks like:



The output/summary in this case looks like:



In both the cases, we can clearly see the working of the model and its efficiency, nowhere, the text summarized is included in the main text/input text. This shows the efficiency of the model and the app's proper functionality.

10. ADVANTAGES & DISADVANTAGES

Advantages:

1. Time Efficiency:

   - Enables quick extraction of key information from lengthy texts, saving time for users.

2. Information Management:

   - Helps manage information overload by condensing content into concise summaries.

3. Enhanced Productivity:

   - Facilitates quicker decision-making and research by providing crucial insights promptly.

4. Customization:

   - Allows users to customize summaries based on length, depth, or content focus.

5. Scalability:

- Can handle large volumes of text for summarization, scaling to accommodate varying loads.

6. Language Diversity:

- Supports summarization in multiple languages, catering to a diverse user base.

Disadvantages:

1. Loss of Context:

- May lose nuanced details or context crucial for comprehensive understanding.

2. Accuracy Challenges:

- Automated systems might miss certain key points, leading to inaccuracies in summaries.

3. Complex Content Handling:

- Struggles with complex or highly technical content, affecting the quality of summaries.

4. Security Concerns:

- Handling sensitive information might raise security and privacy issues.

5. Dependency on Input Quality:

- Quality of summaries heavily relies on the quality and structure of input text.

6. Ethical Considerations:

- Automatic summarization might raise ethical concerns, especially in content interpretation and bias.

11. CONCLUSION

In conclusion, a text summarization system in NLP presents a powerful solution to address the challenges of information overload and time-consuming manual summarization. While offering time efficiency, enhanced productivity, and customization options, it also grapples with accuracy limitations, loss of context, and potential security concerns.

Despite these challenges, the system's scalability, multilingual support, and ability to streamline information make it a valuable tool for various users, including researchers, professionals, and content consumers. Moving forward, addressing accuracy issues, ensuring robust security measures, and continuously improving algorithms will be crucial for advancing the effectiveness and reliability of text summarization systems in NLP.

12. FUTURE SCOPE

In the future, text summarization systems in NLP will likely improve accuracy, offer better customization, adapt in real-time, leverage AI advancements, address biases, handle diverse languages, integrate multimedia, and enable collaborative interactions for enhanced summarization experiences.

13. APPENDIX

Source Code

```
!pip install transformers
!pip install sentencepiece

from transformers import PegasusForConditionalGeneration, AutoTokenizer
import torch
from bs4 import BeautifulSoup
import requests

URL="https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04"

r = requests.get(URL)

soup = BeautifulSoup(r.text, 'html.parser')
results = soup.find_all(['h1', 'p'])
text = [result.text for result in results]
```

```
ARTICLE = ' '.join(text)

ARTICLE

ARTICLE = ARTICLE.replace('.', '.<eos>')
ARTICLE = ARTICLE.replace('?', '?<eos>')
ARTICLE = ARTICLE.replace('!', '!<eos>')

sentences = ARTICLE.split('<eos>')

sentences

max_chunk = 400
current_chunk = 0
chunks = []
for sentence in sentences:
    if len(chunks) == current_chunk + 1:
        if len(chunks[current_chunk]) + len(sentence.split(' ')) <= max_chunk:
            chunks[current_chunk].extend(sentence.split(' '))
        else:
            current_chunk += 1
            chunks.append(sentence.split(' '))
    else:
        print(current_chunk)
        chunks.append(sentence.split(' '))


chunks

for chunk_id in range(len(chunks)):
```

```
    chunks[chunk_id] = ' '.join(chunks[chunk_id])

chunks

len(chunks)

len(chunks[0].split(' '))

chunks[0]

chunks

model_name ='google/pegasus-xsum'
device = 'cuda' if torch.cuda.is_available() else 'cpu'
tokenizer = AutoTokenizer.from_pretrained(model_name)

model = PegasusForConditionalGeneration.from_pretrained(model_name).to(device)

print(model)

total_params = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total_params}")

batch = tokenizer(chunks, truncation=True, padding='longest', return_tensors="pt").to(device)

batch

translated = model.generate(**batch)

translated
```

```
tgt_text = tokenizer.batch_decode(translated, skip_special_tokens=True)

tgt_text

text = ' '.join([summ for summ in tgt_text])

text
```

GitHub & Project Demo Link

GitHub : https://github.com/smartinternz02/SI-GuidedProject-614965-1700737179

Project Demo Link :
https://drive.google.com/file/d/1ACYkxTKTn0_Whm1fSXUWgUgCZMVDIXtc/view?usp=share_link