Extracting Intelligent Insights With AI-Based Systems

TEAM - 592177

21BCE9221 Praveen Sai Krishna Paleti

21BCE9264 Bhavesh Saluru

21BCE9116 Partha Aakash Cheepurupalli

21BCE9306 Abhilesh Killari

AI-Based Intelligent Insight Extractor

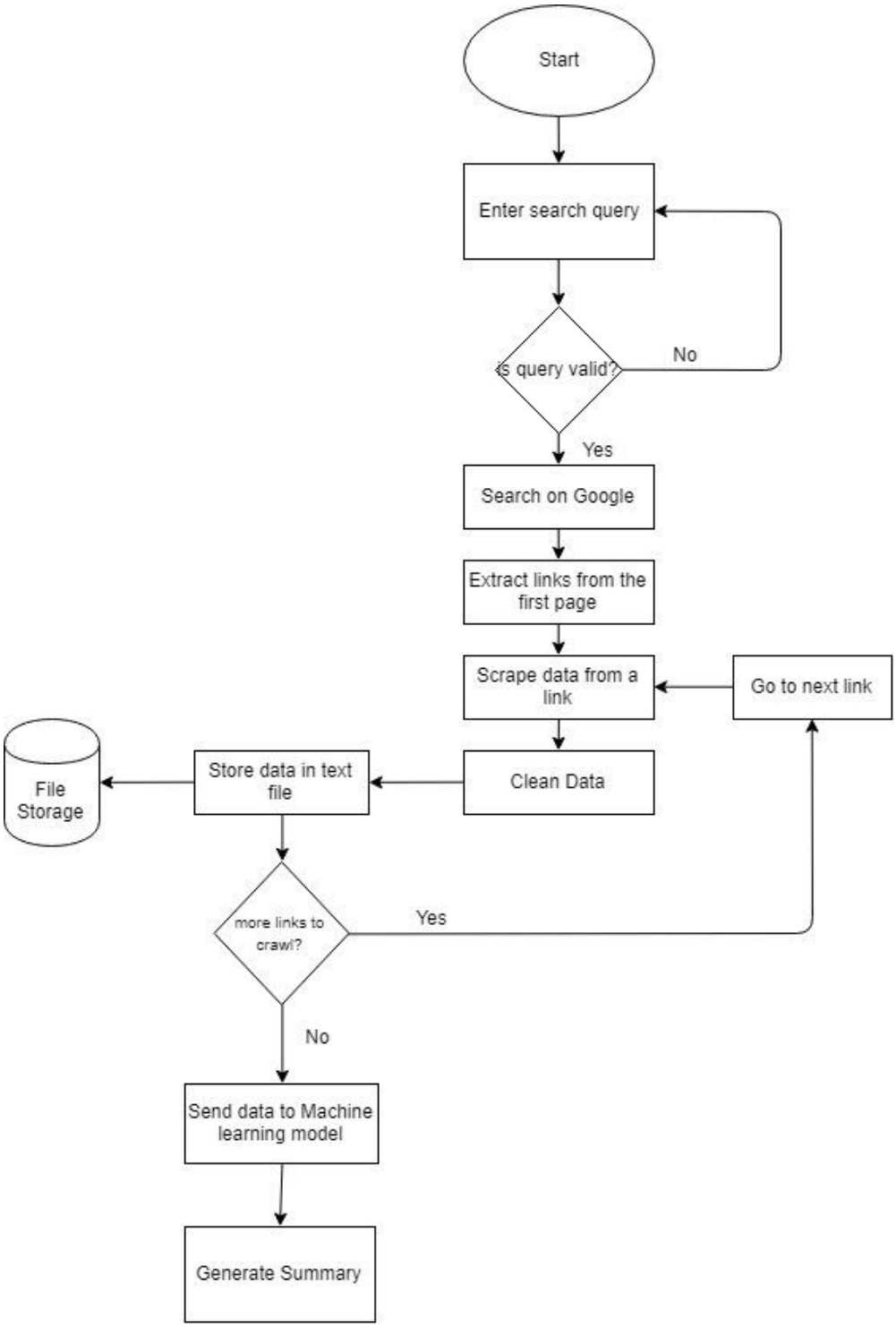
Introduction:

"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.

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 cutting-edge natural language processing techniques, the AI model transforms verbose content into concise and coherent summaries, facilitating efficient comprehension and decision-making.

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.

Technical Architecture :



Prerequisites:

To complete this project, you must require the following software's, concepts, and packages Anaconda Navigator is a free and open-source distribution of the Python and R programming languages for data science and machine learning related applications. It can be installed on Windows, Linux, and macOS.Conda is an open-source, cross-platform, package management system. Anaconda comes with so very nice tools like JupyterLab, Jupyter Notebook.

Also, you can **Google colab** which provides different runtime type where we build our model using given ram and gpu

Python packages:

- Type "pip install Flask" and click enter.
- Type "pip install transformers" and click enter.

Prior Knowledge:

You must have prior knowledge of the following topics to complete this project.

- NLP Concepts –pipeline techniques like tokenization to convert text to numbers and decoding after getting the summary from the model
- Transformers-Pegasus-xsum model which performs abstractive text summarization
- Flask Basics- Web framework used for building Web applications
- Html-For building the structure of the website
- CSS-For presentation of the website
- BeatifulSoup-For scraping the data from the url
- Bootstrap-Framework to implement CSS

Project Objectives:

By the end of this project, you will:

- Know fundamental concepts and techniques used for NLP.
- Gain a broad understanding of Transformers.
- Gain knowledge on pre-processing the text data.

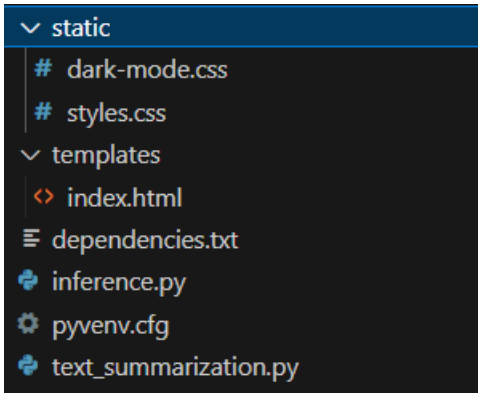
Project Flow:

- The user interacts with the UI to enter the input.
- Entered input is analyzed by the model which is integrated.
- Once the model analyses the input the summary is showcased on the UI
- To accomplish this, we have to complete all the activities listed below,
- **Import Required libraries**
    - Read Dataset
- **Text Pre-Processing**
    - Accessing data from the URL
    - Splitting our data into sentences
    - Combining sentences into chunks
- **Model Building**
    - Setting up environment for initializing the model
    - Initialising the model
    - Tokenization
    - Get the summary tokens from the model
    - Decoding the summary tokens
    - Summary

- **Application Building**
    - Building Html Pages
    - Build Python code
    - Run the application

Project Structure:

Create the Project folder which contains files as shown below



- We are building a flask application that needs HTML pages stored in the templates folder CSS pages stored in static folder and a python script inference.py and text_summarization.py for scripting.

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.

```
ARTICLE
```

'What is a Transformer? Maxime Follow Inside Machine learning -- 28 Listen Share An Introduction to Transformers and Sequence-to-Sequence Learning for Machine Learning New deep learning models are introduced at an increasing rate and sometimes it's hard to keep track of all the novelties. That said, one particular neural network model has proven to be especially effective for common natural language processing tasks. The model is called a Transformer and it makes use of several methods and mechanisms that I'll introduce here. The papers I refer to in the post offer a more detailed and quantitative description. Part 1: Sequence to Sequence Learning and Attention The paper 'Attention Is All You Need' describes transformers and what is called a sequence-to-sequence architecture. Sequence-to-Sequence (or Seq2Seq) is a neural net that transforms a given sequence of elements, such as the sequence of words in a sentence, into another sequence. (Well, this might not surprise you considering the name.) Seq2Seq models are particularly good at translation, where the sequence of words from one language is transformed into a sequence of different words in another language. A popular choice for this type of model is Long-Short-Term-Memory (LSTM)-based models. With sequence-dependent data, the LSTM modules can give meaning to the sequence while remembering (or forgetting) the parts it finds important (or unimportant). Sentences, for example, are sequence-dependent since the order of the words is crucial for understanding the sentence. LSTM are a natural choice for this type of data. Seq2Seq models consist of an Encoder and a Decoder. The Encoder takes the input sequence and maps it into a higher dimensional space (n-dimensional vector). That abstract vector is fed into the Decoder which turns it into an output sequence. The output sequence can be in another language, symbols, a copy of the input, etc. Imagine the Encoder and Decoder as human translators who can speak only two languages. Their first language is their mother tongue, which differs between both of them (e.g. German and French) and their second language an imaginary one they have in common. To translate German into French, the Encoder converts the German sentence into the other language it knows, namely the imaginary language. Since the Decoder is able to read that imaginary language, it can now translates from that language into French. Together, the model (consisting of Encoder and Decoder) can translate German into French! Suppose that, initially, neither the Encoder or the Decoder is very fluent in the imaginary language. To learn it, we train them (the model) on a lot of examples. A very basic choice for the Encoder and the Decoder of the Seq2Seq model is a single LSTM for each of them. You're wondering when the Transformer will finally come into play, aren't you? We need one more technical detail to make Transformers easier to understand: Attention. The attention-mechanism looks at an input sequence and decides at each step which other parts of the sequence are important. It sounds abstract, but let me clarify with an easy example: When reading this text, you always focus on the word you read but at the same time your mind still holds the important keywords of the text in memory in order to provide context. An attention-mechanism works similarly for a given sequence. For our example with the human Encoder and Decoder, imagine that instead of only writing down the translation of the sentence in the imaginary language, the Encoder also writes down keywords that are important to the semantics of the sentence, and gives them to the Decoder in addition to the regular translation. Those new keywords make the translation much easier for the Decoder because it knows what parts of the sentence are important and which key terms give the sentence context. In other words, for each input that the LSTM (Encoder) reads, the attention-mechanism takes into account several other inputs at the same time and decides which ones are important by attributing different weights to those inputs. The Decoder will then take as input the encoded sentence and the weights provided by the attention-mechanism. To learn more about attention, see this article. And for a more scientific approach than the one provided, read about different attention-based approaches for Sequence-to-Sequence models in this great paper called 'Effective Approaches to Attention-based Neural Machine Translation'. Part 2: The Transformer The paper 'Attention Is All You Need' introduces a novel architecture called Transformer. As the title indicates, it uses the attention-mechanism we saw earlier. Like LSTM, Transformer is an architecture for transforming one sequence into another one with the help of two parts (Encoder and Decoder), but it differs from the previously described/existing sequence-to-sequence models because it does not imply any Recurrent Networks (GRU, LSTM, etc.). Recurrent Networks were, until now, one of the best ways to capture the timely dependencies in sequences. However, the team presenting the paper proved that an architecture with only attention-mechanisms without any RNN (Recurrent Neural Networks) can improve on the results in translation task and other tasks! One improvement on Natural Language Tasks is presented by a team introducing BERT: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. So, what exactly is a Transformer? An image is worth thousand words, so we will start with that! The Encoder is on the left and the Decoder is on the right. Both Encoder and Decoder are composed of modules that can be stacked on top of each other multiple times, which is described by Nx in the figure. We see that the modules consist mainly of Multi-Head Attention and Feed Forward layers. The inputs and outputs (target sentences) are first embedded into an n-dimensional space since we cannot use strings directly. One slight but important part of the model is the positional encoding of the different words. Since we have no recurrent networks that can remember how sequences are fed into a model, we need to somehow give every word/part in our sequence a relative position since a sequence depends on the order of its elements. These positions are added to the embedded representation (n-dimensional vector) of each word. Let's have a closer look at these Multi-Head Attention bricks in the model: Let's start with the left description of the attention-mechanism. It's not very complicated and can be described by the following equation: Q is a matrix that contains the query (vector representation of one w

Activity-2: Splitting our data into sentences

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

```
[8]  ARTICLE = ARTICLE.replace('.', '.<eos>')
     ARTICLE = ARTICLE.replace('?', '?<eos>')
     ARTICLE = ARTICLE.replace('!', '!<eos>')

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

```
sentences
```

```
[' Part 3: Use-Case 'Transformer for Time-Series' We have seen the Transformer architecture and we know from literature and the 'Attention is All you Need' authors that the model does extremely well in language tasks.',
 ' Let's now test the Transformer in a use case.',
 ' Instead of a translation task, let's implement a time-series forecast for the hourly flow of electrical power in Texas, provided by the Electric Reliability Council of Texas (ERCOT).',
 ' You can find the hourly data here.',
 ' A great detailed explanation of the Transformer and its implementation is provided by harvardnlp.',
 ' If you want to dig deeper into the architecture, I recommend going through that implementation.',
 ' Since we can use LSTM-based sequence-to-sequence models to make multi-step forecast predictions, let's have a look at the Transformer and its power to make those predictions.',
 ' However, we first need to make a few changes to the architecture since we are not working with sequences of words but with values.',
 ' Additionally, we are doing an auto-regression and not a classification of words/characters.',
 ' The available data gives us hourly load for the entire ERCOT control area.',
 ' I used the data from the years 2003 to 2015 as a training set and the year 2016 as test set.',
 ' Having only the load value and the timestamp of the load, I expanded the timestamp to other features.',
 ' From the timestamp, I extracted the weekday to which it corresponds and one-hot encoded it.',
 ' Additionally, I used the year (2003, 2004, …, 2015) and the corresponding hour (1, 2, 3, …, 24) as the value itself.',
 ' This gives me 11 features in total for each hour of the day.',
 ' For convergence purposes, I also normalized the ERCOT load by dividing it by 1000.',
 ' To predict a given sequence, we need a sequence from the past.',
 ' The size of those windows can vary from use-case to use-case but here in our example I used the hourly data from the previous 24 hours to predict the next 12 hours.',
 ' It helps that we can adjust the size of those windows depending on our needs.',
 ' For example, we can change that to daily data instead of hourly data.',
 ' As a first step, we need to remove the embeddings, since we already have numerical values in our input.',
 ' An embedding usually maps a given integer into an n-dimensional space.',
 ' Here instead of using the embedding, I simply used a linear transformation to transform the 11-dimensional data into an n-dimensional space.',
 ' This is similar to the embedding with words.',
 ' We also need to remove the SoftMax layer from the output of the Transformer because our output nodes are not probabilities but real values.',
 ' After those minor changes, the training can begin!',
 ' As mentioned, I used teacher forcing for the training.',
 ' This means that the encoder gets a window of 24 data points as input and the decoder input is a window of 12 data points where the first one is a 'start-of-sequence' value and the following data points are simply the target sequence.',
 ' Having introduced a 'start-of-sequence' value at the beginning, I shifted the decoder input by one position with regard to the target sequence.',
```

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])
```

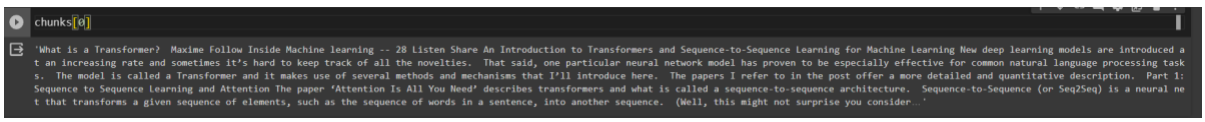
This code iterates through each chunk in the list of chunks. For each chunk, it uses the **join** method to concatenate the individual words within the chunk into a single string. The resulting string is then assigned back to the corresponding index in the **chunks** list. Essentially, it transforms 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


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

```
383
```

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

```
len(chunks)
```

```
8
```

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

```
chunks
```

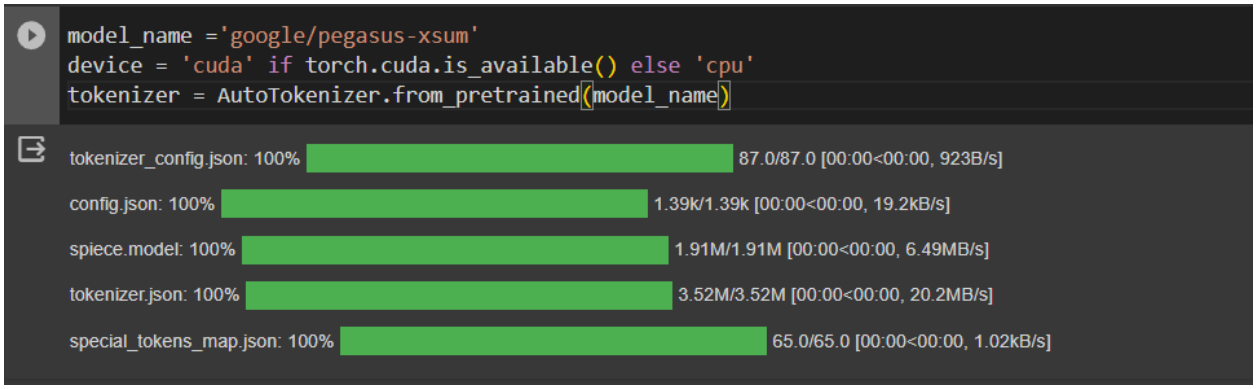
```
['What is a Transformer?  Maxime Follow Inside Machine learning -- 28 Listen Share An Introduction to Transformers and Sequence-to-Sequence Learning for Machine Learning New deep learning models are
introduced at an increasing rate and sometimes it’s hard to keep track of all the novelties.  That said, one particular neural network model has proven to be especially effective for common natural language
processing tasks.  The model is called a Transformer and it makes use of several methods and mechanisms that I’ll introduce here.  The papers I refer to in the post offer a more detailed and quantitative
description.  Part 1: Sequence to Sequence Learning and Attention The paper ‘Attention Is All You Need’ describes transformers and what is called a sequence-to-sequence architecture.  Sequence-to-Sequence (
Seq2Seq) is a neural net that transforms a given sequence of elements, such as the sequence of words in a sentence, into another sequence.  (Well, this might not surprise you considering the name. ) Seq2Seq
models are particularly good at translation, where the sequence of words from one language is transformed into a sequence of different words in another language.  A popular choice for this type of model is
Long-Short-Term-Memory (LSTM)-based models.  With sequence-dependent data, the LSTM modules can give meaning to the sequence while remembering (or forgetting) the parts it finds important (or unimportant).
Sentences, for example, are sequence-dependent since the order of the words is crucial for understanding the sentence.  LSTM are a natural choice for this type of data.  Seq2Seq models consist of an Encoder
and a Decoder.  The Encoder takes the input sequence and maps it into a higher dimensional space (n-dimensional vector).  That abstract vector is fed into the Decoder which turns it into an output sequence.
The output sequence can be in another language, symbols, a copy of the input, etc.  Imagine the Encoder and Decoder as human translators who can speak only two languages.  Their first language is their moth
tongue, which differs between both of them (e. g.  German and French) and their second language an imaginary one they have in common.  To translate German into French, the Encoder converts the German senten
into the other language it knows, namely the imaginary language.',
 ' Since the Decoder is able to read that imaginary language, it can now translates from that language into French.  Together, the model (consisting of Encoder and Decoder) can translate German into French!
Suppose that, initially, neither the Encoder or the Decoder is very fluent in the imaginary language.  To learn it, we train them (the model) on a lot of examples.  A very basic choice for the Encoder and t
Decoder of the Seq2Seq model is a single LSTM for each of them.  You’re wondering when the Transformer will finally come into play, aren’t you?  We need one more technical detail to make Transformers easier
to understand: Attention.  The attention-mechanism looks at an input sequence and decides at each step which other parts of the sequence are important.  It sounds abstract, but let me clarify with an easy
example: When reading this text, you always focus on the word you read but at the same time your mind still holds the important keywords of the text in memory in order to provide context.  An attention-
mechanism works similarly for a given sequence.  For our example with the human Encoder and Decoder, imagine that instead of only writing down the translation of the sentence in the imaginary language, the
Encoder also writes down keywords that are important to the semantics of the sentence, and gives them to the Decoder in addition to the regular translation.  Those new keywords make the translation much
easier for the Decoder because it knows what parts of the sentence are important and which key terms give the sentence context.  In other words, for each input that the LSTM (Encoder) reads, the attention-
mechanism takes into account several other inputs at the same time and decides which ones are important by attributing different weights to those inputs.  The Decoder will then take as input the encoded
sentence and the weights provided by the attention-mechanism.  To learn more about attention, see this article.  And for a more scientific approach than the one provided, read about different attention-base
approaches for Sequence-to-Sequence models in this great paper called ‘Effective Approaches to Attention-based Neural Machine Translation’.  Part 2: The Transformer The paper ‘Attention Is All You Need’
introduces a novel architecture called Transformer.  As the title indicates, it uses the attention-mechanism we saw earlier.',
 ' Like LSTM, Transformer is an architecture for transforming one sequence into another one with the help of two parts (Encoder and Decoder), but it differs from the previously described/existing sequence-t
sequence models because it does not imply any Recurrent Networks (GRU, LSTM, etc. ).  Recurrent Networks were, until now, one of the best ways to capture the timely dependencies in sequences.  However, the
team presenting the paper proved that an architecture with only attention-mechanisms without any RNN (Recurrent Neural Networks) can improve on the results in translation task and other tasks!  One
improvement on Natural Language Tasks is presented by a team introducing BERT: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.  So, what exactly is a Transformer?  An image
is worth thousand words, so we will start with that!  The Encoder is on the left and the Decoder is on the right.  Both Encoder and Decoder are composed of modules that can be stacked on top of each other
multiple times, which is described by Nx in the figure.  We see that the modules consist mainly of Multi-Head Attention and Feed Forward layers.  The inputs and outputs (target sentences) are first embedded
into an n-dimensional space since we cannot use strings directly.  One slight but important part of the model is the positional encoding of the different words.  Since we have no recurrent networks that can
remember how sequences are fed into a model, we need to somehow give every word/part in our sequence a relative position since a sequence depends on the order of its elements.  These positions are added to
the embedded representation (n-dimensional vector) of each word.  Let’s have a closer look at these Multi-Head Attention bricks in the model: Let’s start with the left description of the attention-mechanism
It’s not very complicated and can be described by the following equation: Q is a matrix that contains the query (vector representation of one word in the sequence), K are all the keys (vector representation
of all the words in the sequence) and V are the values, which are again the vector representations of all the words in the sequence.  For the encoder and decoder, multi-head attention modules, V consists of
```

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 human-like 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

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

pytorch_model.bin: 100% 2.28G/2.28G [00:35<00:00, 87.2MB/s]
Some weights of PegasusForConditionalGeneration were not initialized from the model checkpoint at
You should probably TRAIN this model on a down-stream task to be able to use it for predictions a
generation_config.json: 100% 259/259 [00:00<00:00, 11.8kB/s]
```

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 CUDA-compatible 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**.

```
batch
```

```
{'input_ids': tensor([[  463,   117,   114, 51979,   152, 81398,  6044,  7395,  3838,   761,
          1315,  2482, 11015,  6274,   983, 12621,   112, 38979,   111, 38641,
           121,   497,   121,   283, 52495, 14191,  4473,   118,  3838,  4473,
           351,  1355,   761,  1581,   127,  2454,   134,   142,  2186,   872,
           111,  1254,   126,   123,   116,   514,   112,   376,  1103,   113,
           149,   109, 70669,   107,   485,   243,   108,   156,   970, 14849,
           952,   861,   148,  3288,   112,   129,   704,   957,   118,   830,
           710,  1261,  2196,  2722,   107,   139,   861,   117,   568,   114,
         51979,   111,   126,   493,   207,   113,   500,  1625,   111,  7869,
           120,   125,   123,   267,  4094,   264,   107,   139,  3392,   125,
          3984,   112,   115,   109,   450,   369,   114,   154,  2067,   111,
         13350,  3180,   107,  3643,  8403, 38641,   112, 38641,  4473,   111,
         26476,   139,   800,   402, 41666,   125,   116,   436,   226,  5380,
           123,  5002, 40749,   111,   180,   117,   568,   114,  5936,   121,
           497,   121, 69987,  3105,   107, 38641,   121,   497,   121,   283,
         52495, 14191,   143,   490,   110, 77727,   522, 77727,   158,   117,
           114, 14849,  2677,   120, 17474,   114,   634,  5936,   113,  1811,
           108,   253,   130,   109,  5936,   113,   989,   115,   114,  5577,
           108,   190,   372,  5936,   107,   143,  3371,   108,   136,   382,
           146,  2989,   119,  2635,   109,   442,   107,   110,   158,   110,
         77727,   522, 77727,  1581,   127,  1533,   234,   134,  5256,   108,
           241,   109,  5936,   113,   989,   135,   156,  1261,   117,  7267,
           190,   114,  5936,   113,   291,   989,   115,   372,  1261,   107,
           202,   785,   814,   118,   136,   619,   113,   861,   117,  2859,
           121, 17648,   121, 24953,   121, 46118,   143, 15417,  2259,   158,
           121,   936,  1581,   107,   441,  5936,   121, 21048,   335,   108,
           109, 15833,  2259,  6364,   137,   361,  2050,   112,   109,  5936,
           277, 13358,   143,   490, 18553,   158,   109,   972,   126,  5258,
```

Activity 4: Get the summary tokens from the model

```
translated = model.generate(**batch)
```

- **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 human-readable 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.

```
tgt_text
```

```
['In this post, I'm going to introduce you to a new type of deep learning model.',
 'We've already shown that a machine translation model called Seq2Seq can read an imaginary language.',
 'In this paper, we present a new model for multi-to-sequence translation called Transformer.',
 'For the attention module that is taking into account the encoder and the decoder sequences, V is different from the sequence represented by Q.',
 'In this blog, I'm going to show you how to shift the decoder input sequence during training.',
 'In the second part of my series on sequence-to-sequence models, I am going to show you how to train and implement the Transformer.',
 'In this example, I used teacher forcing to train the Transformer to predict the next 12 hours.',
 'In this blog, I'm going to show you how to train a model to predict the next 12 hours of a time-series.']
```

Activity 6: Summary

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

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.

```
[33] text
```

```
'In this post, I'm going to introduce you to a new type of deep learning model. We've already shown that a machine translation model called Seq2Seq can read an imaginary language. In this paper, we present a ne
w model for multi-to-sequence translation called Transformer. For the attention module that is taking into account the encoder and the decoder sequences, V is different from the sequence represented by Q. In th
is blog, I'm going to show you how to shift the decoder input sequence during training. In the second part of my series on sequence-to-sequence models, I am going to show you how to train and implement the Tran
sformer. In this example, I used teacher forcing to train the Transformer to predict the next 12 hours. In this blog, I'm going to show you how to train a model to predict the next 12 hours of a time-series.'
```

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

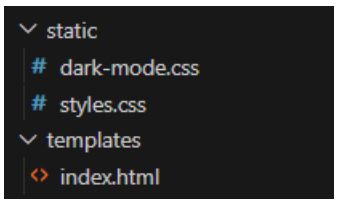
Activity-1: Building Html Pages

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



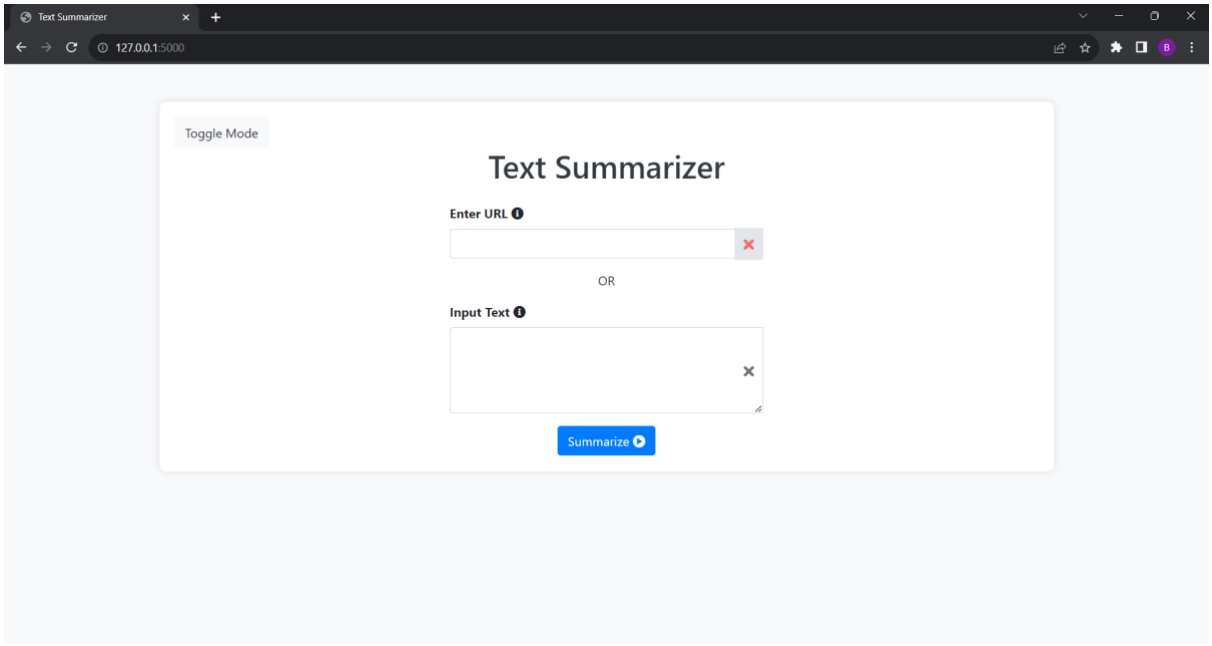
The main focus of the app is to keep it simple, clean and functional.

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

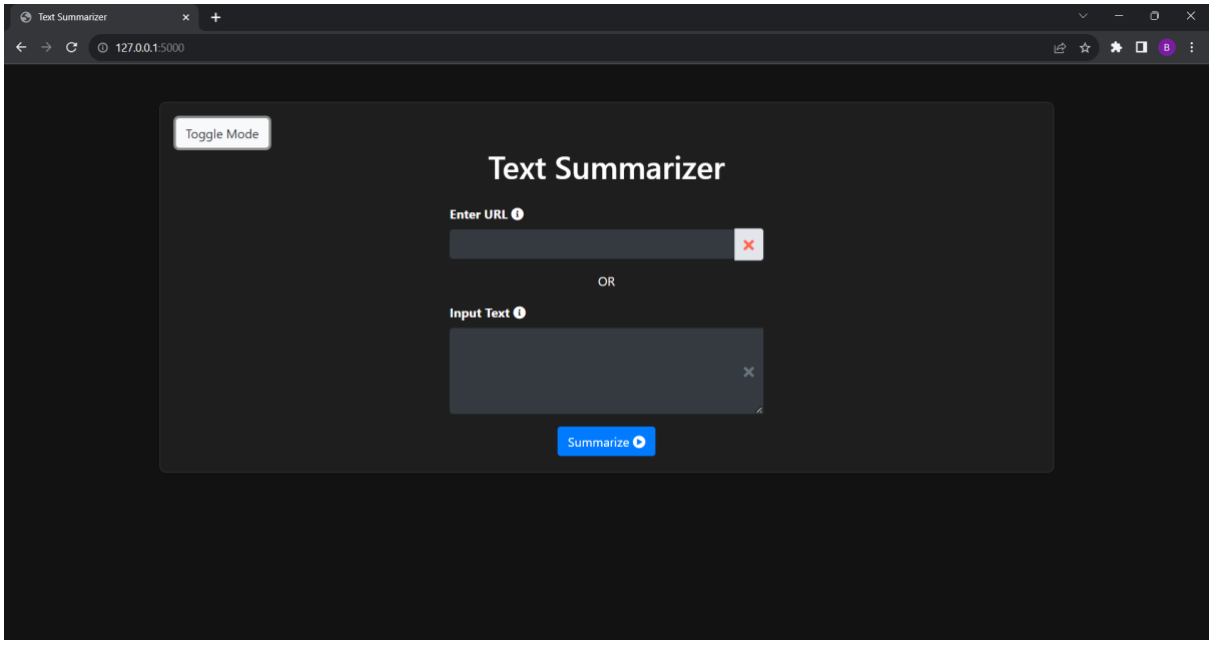
The UI is kept as simple as possible to avoid any confusions and efforts were made to enhance the UI by providing hints so as to make it easy and understandable to use.

Let’s see how our web page looks like:

Light Mode:



Dark Mode:

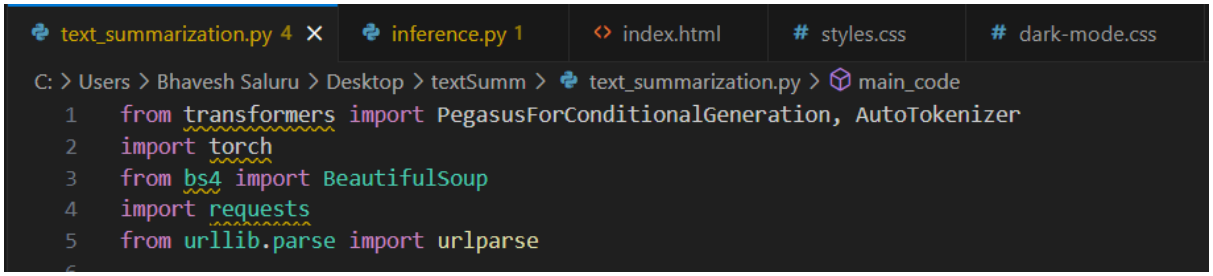


Activity 2: Build Python code

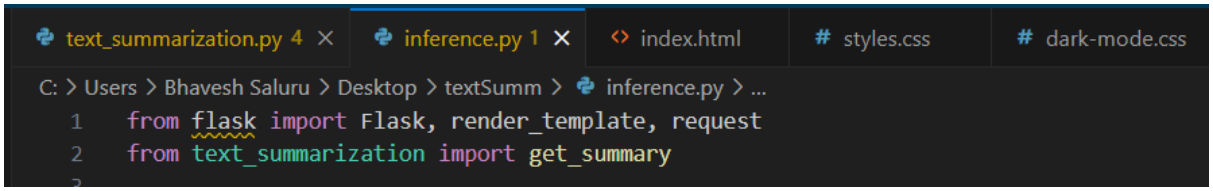
Import the libraries

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.

```
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.

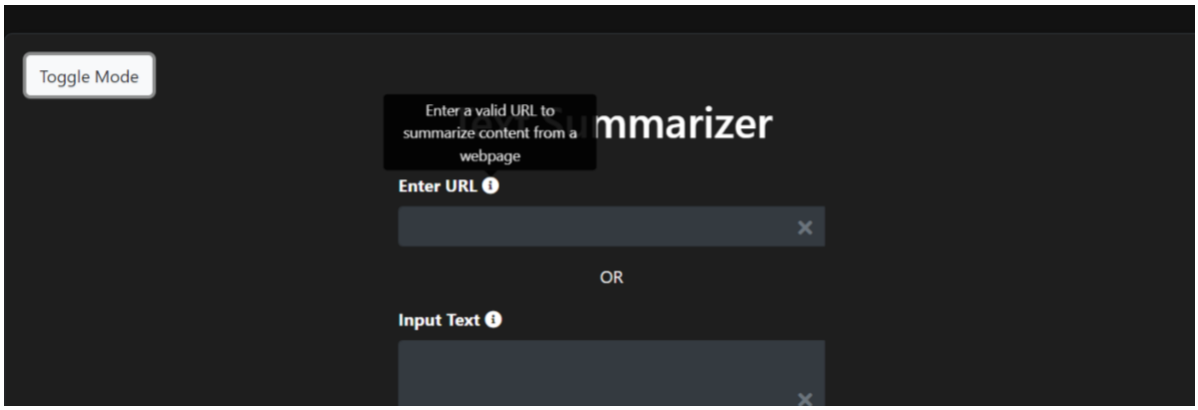
Output:

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:

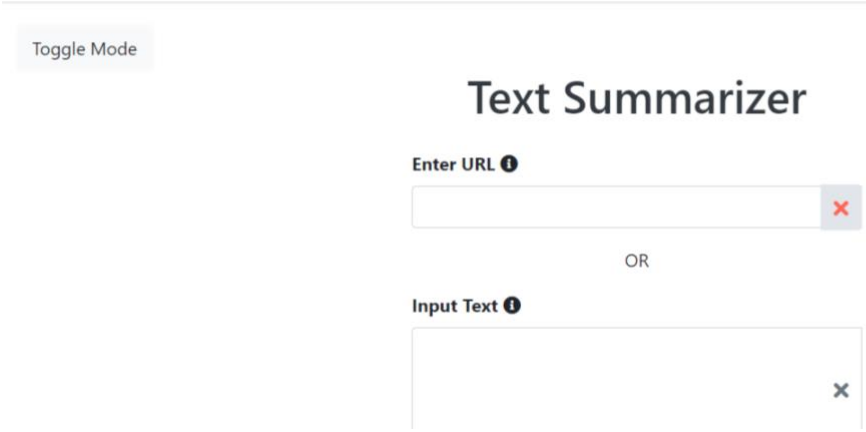
In the first field, user is required to enter/provide the url, an about button is provided next to the text field, to provide more details about the component.



Option two:

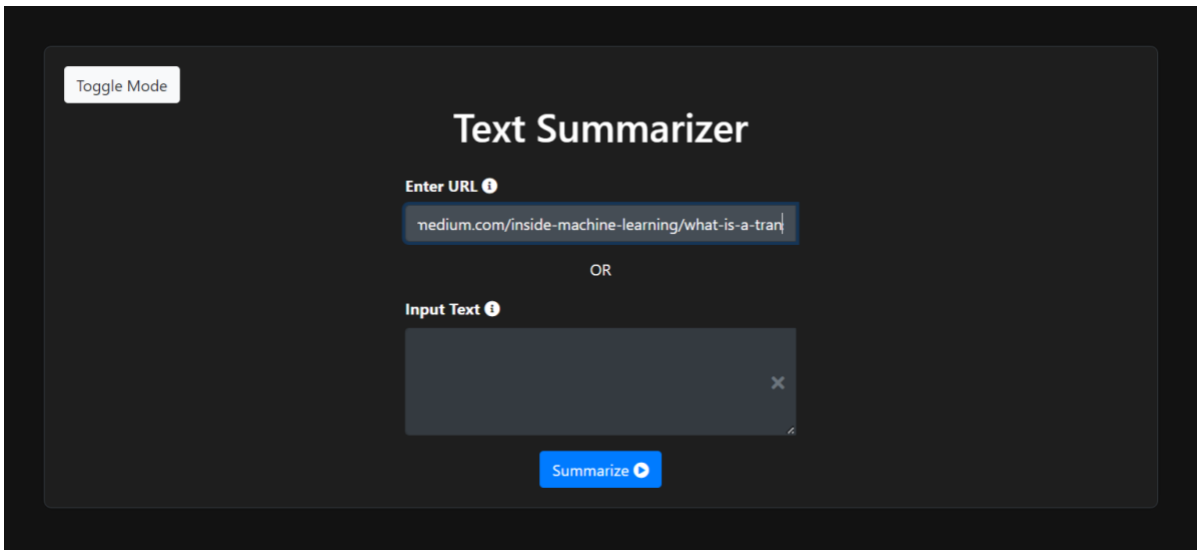
In the second field, user is required to enter/provide the text, an about button is provided next to the text field, to provide more details about the component.

If the user intends to clear the text in the field, one can click on the cancel button next to the input text field

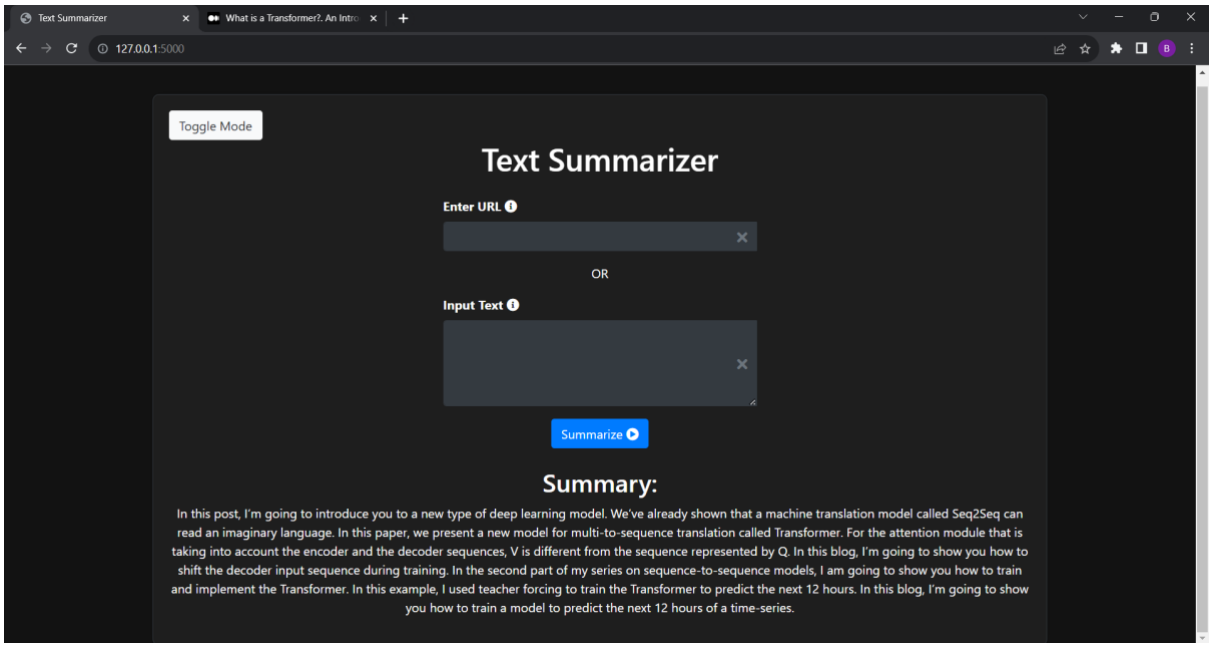


Once the text is entered, one can click summarize and wait for the summary of the text.

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

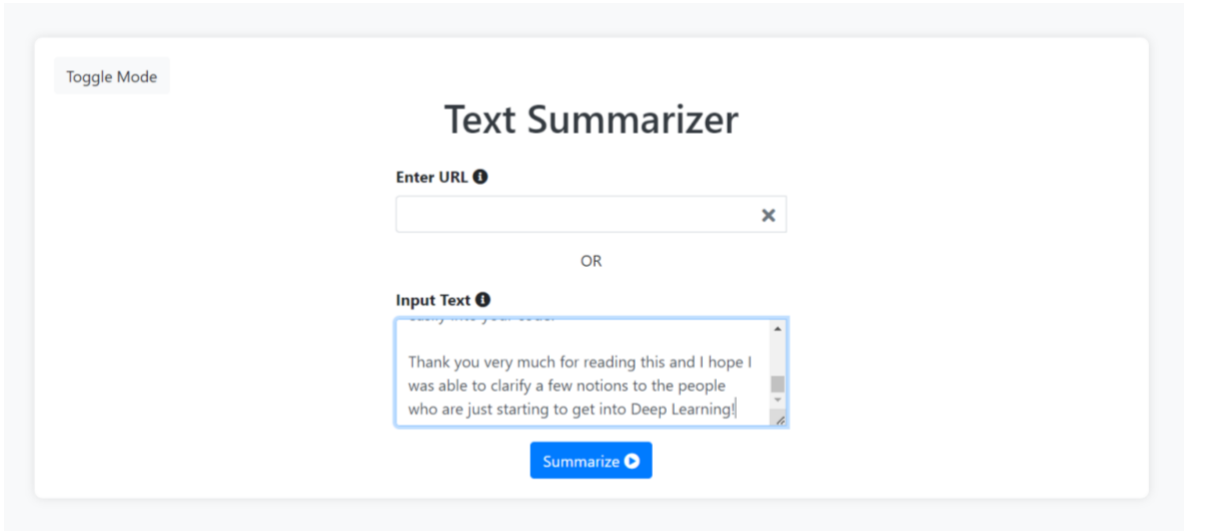


The output/summary looks as follows:

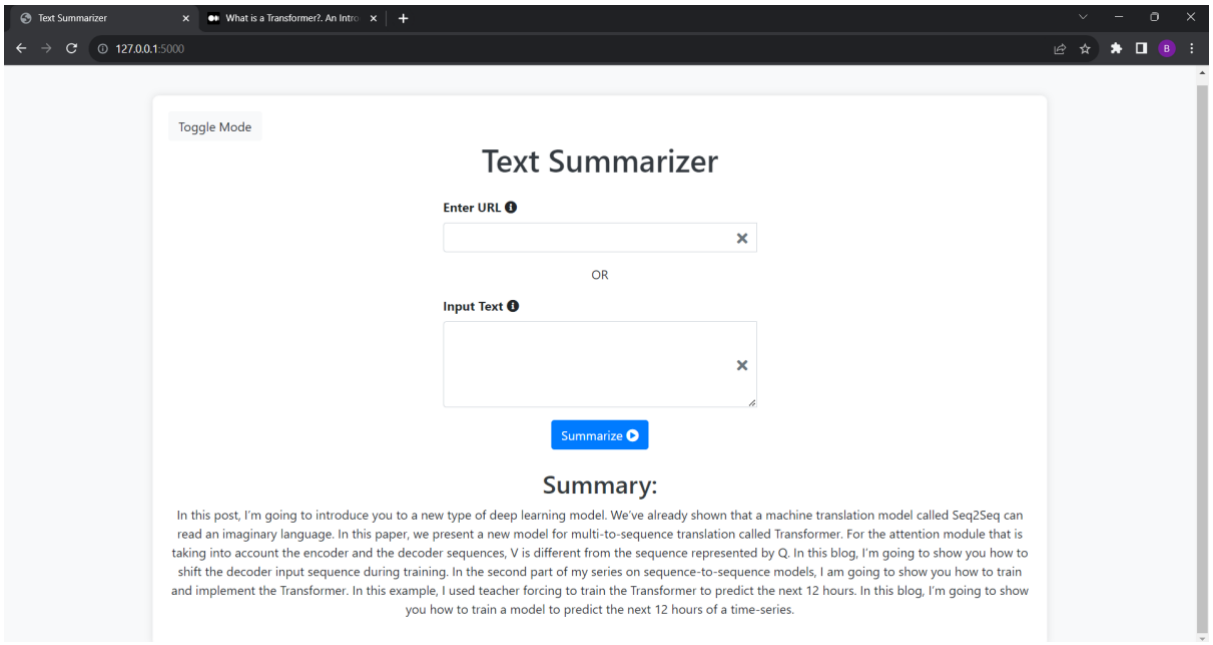


The same can be done on direct text as well, which generates the same output, testing the same in light mode this time instead of dark to show the proper working of the UI

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.