AI-Powered PDF Document Summarization and Question–Answer Generation System

SUBMITTED BY,

CHINMAYA A S

AI-Powered PDF Document Summarization and Question–Answer Generation System Using Transformer Models

Introduction

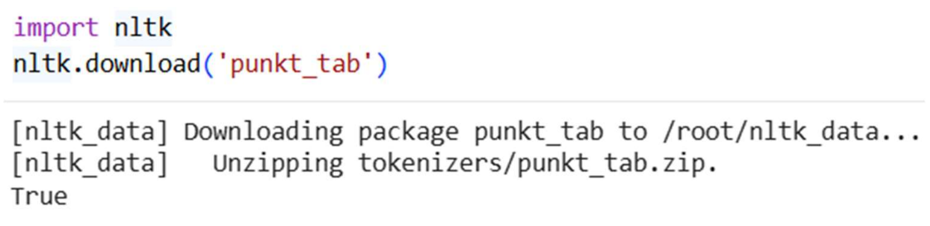
This project presents an end-to-end Natural Language Processing (NLP) system that transforms a static PDF document into an intelligent, interactive content experience. The workflow begins by extracting raw text from a PDF file using pdfplumber, converting unstructured document data into machine-readable format. The extracted text is then processed through multiple transformer-based models to generate meaningful insights and interactive outputs.

First, the document is summarized using the "t5-small" model through the Hugging Face pipeline("summarization"), enabling concise understanding of lengthy content. After summarization, the text is segmented into structured passages using nltk sentence tokenization to ensure efficient downstream processing.

Next, automated question generation is performed using the "valhalla/t5-base-qg-hl" model with a "text2text-generation" pipeline. This converts document passages into context-aware questions. Finally, a transformer-based question-answering model, "deepset/roberta-base-squad2", is used to extract precise answers directly from the original context while avoiding duplicate responses through set-based logic.

Overall, this project demonstrates how transformer models can be integrated into a unified pipeline to convert static documents into summarized, question-driven, and answer-aware AI-interactive systems.

1.NLTK Resource Initialization for Tokenization Support



```
import nltk
nltk.download('punkt_tab')
```

```
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
True
```

This step is responsible for preparing the Natural Language Toolkit (NLTK) environment so that text tokenization can work properly in later stages of your project.

When import nltk runs, Python loads the nltk library into memory. This makes all of NLTK’s functions, modules, and utilities available for use in your script. At this point, no processing happens yet — it simply makes the library accessible.

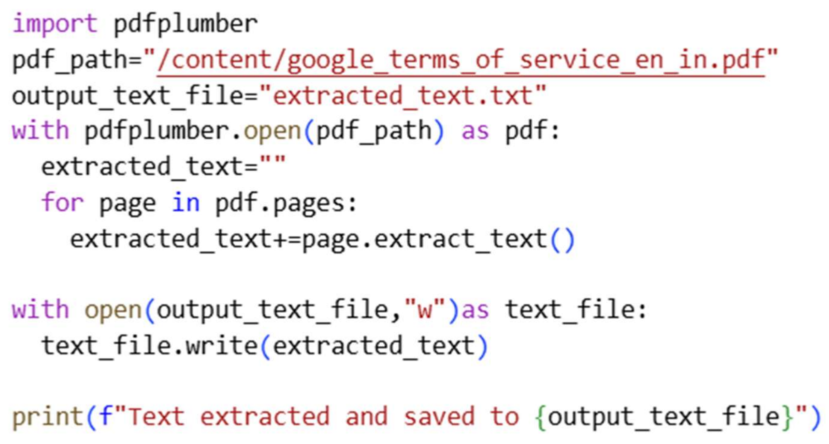
When nltk.download('punkt_tab') executes, NLTK checks whether the resource 'punkt_tab' is already available in the local NLTK data directory. If it is not present, NLTK downloads it from its official data repository and stores it locally. This resource is related to sentence tokenization support and is required for breaking text into sentences correctly.

In simple terms, this step:

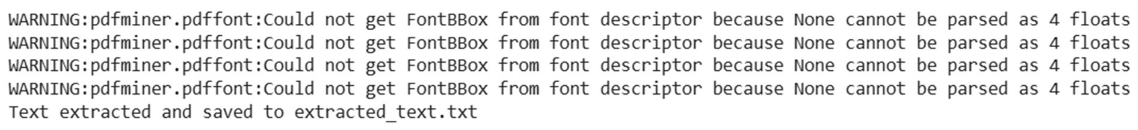
- Loads the nltk library
- Downloads the required tokenization resource
- Ensures sentence splitting functions work without errors
- Prevents runtime errors related to missing tokenizer data

This initialization step is important because many NLP operations (like sentence tokenization in question generation or preprocessing) depend on these resources being available.

2.PDF Text Extraction and Saving using pdfplumber



```
import pdfplumber
pdf_path="/content/google_terms_of_service_en_in.pdf"
output_text_file="extracted_text.txt"
with pdfplumber.open(pdf_path) as pdf:
  extracted_text=""
  for page in pdf.pages:
    extracted_text+=page.extract_text()

with open(output_text_file,"w")as text_file:
  text_file.write(extracted_text)

print(f"Text extracted and saved to {output_text_file}")
```



```
WARNING:pdfminer.pdffont:Could not get FontBBox from font descriptor because None cannot be parsed as 4 floats
WARNING:pdfminer.pdffont:Could not get FontBBox from font descriptor because None cannot be parsed as 4 floats
WARNING:pdfminer.pdffont:Could not get FontBBox from font descriptor because None cannot be parsed as 4 floats
WARNING:pdfminer.pdffont:Could not get FontBBox from font descriptor because None cannot be parsed as 4 floats
Text extracted and saved to extracted_text.txt
```

This code is used to extract text from a PDF file located at pdf_path, store the extracted content inside the variable extracted_text, save it into a file named output_text_file, and finally print a confirmation message. Below is a detailed, line-by-line explanation using the exact words from your code.

The line import pdfplumber loads the pdfplumber library into the program. This allows the script to use functions like pdfplumber.open() to read and extract text from a PDF file. Without this import statement, the program would not recognize pdfplumber.

Next, pdf_path="/content/google_terms_of_service_en_in.pdf" creates a variable named pdf_path and assigns it the string path of the PDF file. This tells the program where the PDF is located. The file /content/google_terms_of_service_en_in.pdf is the source from which text will be extracted.

Then, output_text_file="extracted_text.txt" creates another variable named output_text_file and assigns it the name of the text file where the extracted content will be saved. This means all extracted text will be written into extracted_text.txt.

The statement with pdfplumber.open(pdf_path) as pdf: opens the PDF file specified in pdf_path.

- pdfplumber.open(pdf_path) loads the PDF.
- as pdf assigns the opened PDF to the variable pdf.
- The with statement ensures the PDF file is automatically closed after the block finishes executing.

Inside this block, extracted_text="" initializes an empty string variable named extracted_text. This variable will store all the text extracted from every page of the PDF.

The loop for page in pdf.pages: iterates through each page in pdf.pages. Here:

- pdf.pages contains all pages of the PDF.
- page represents one page at a time during each iteration.

Inside the loop, extracted_text+=page.extract_text() extracts text from the current page using page.extract_text() and appends it to the existing extracted_text.

- page.extract_text() reads text from that specific page.
- += adds the new text to the previously stored text.
- Over multiple iterations, this combines text from all pages into one single string.

After all pages are processed, the next block with open(output_text_file,"w")as text_file: creates and opens a file named extracted_text.txt in write mode "w".

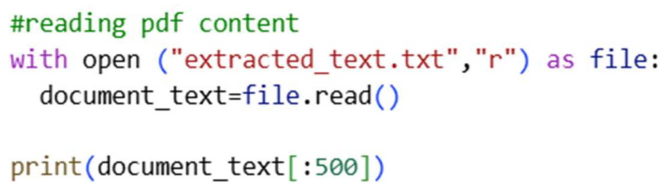
- If the file does not exist, it is created.
- If it already exists, its content is overwritten.
- as text_file assigns the file object to text_file.

Inside this block, text_file.write(extracted_text) writes the complete content stored in extracted_text into extracted_text.txt. This saves all the extracted PDF text into the output file.

Finally, print(f"Text extracted and saved to {output_text_file}") prints a confirmation message.

- The f indicates an f-string.
- {output_text_file} dynamically inserts the value "extracted_text.txt" into the message.
- This confirms that the text extraction and saving process was successful.

3.Reading Extracted PDF Text and Displaying Preview Content



```
#reading pdf content
with open ("extracted_text.txt","r") as file:
  document_text=file.read()

print(document_text[:500])
```

This code section is used to read the previously saved file extracted_text.txt, store its contents inside the variable document_text, and display the first 500 characters as a preview. Below is a detailed explanation of what happens when each line runs.

The line with open ("extracted_text.txt","r") as file: opens the file named "extracted_text.txt" in read mode "r".

- open("extracted_text.txt","r") tells Python to access the file for reading.
- "r" means the file will only be read, not modified.
- as file assigns the opened file object to the variable file.
- The with statement ensures that the file is automatically closed after the block finishes executing, which prevents memory leaks and resource issues.

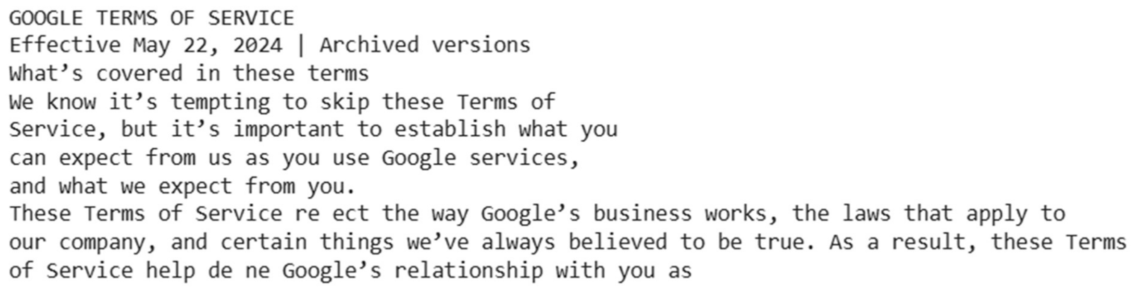
Inside this block, document_text=file.read() reads the entire content of file.

- file.read() retrieves all text stored inside "extracted_text.txt".
- The content is stored in the variable document_text.
- At this point, document_text contains the complete extracted PDF text as one large string.

After the file has been read and automatically closed, the line print(document_text[:500]) displays only the first 500 characters of document_text.

- document_text[:500] uses string slicing.
- : starts from the beginning of the string.
- 500 means it stops at character index 500.

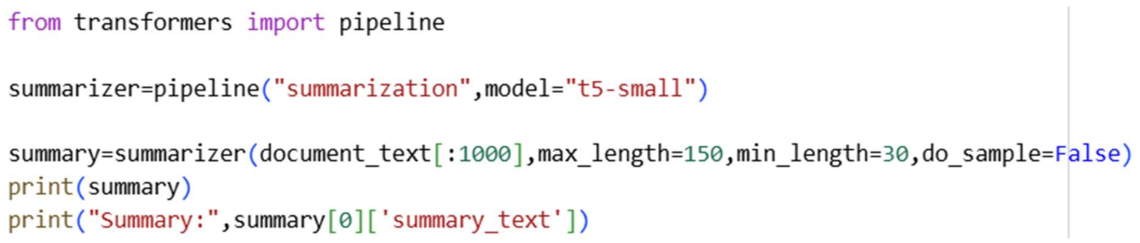
- This is done to preview the content without printing the entire document, which might be very large.



```
GOOGLE TERMS OF SERVICE
Effective May 22, 2024 | Archived versions
What’s covered in these terms
We know it’s tempting to skip these Terms of
Service, but it’s important to establish what you
can expect from us as you use Google services,
and what we expect from you.
These Terms of Service re ect the way Google’s business works, the laws that apply to
our company, and certain things we’ve always believed to be true. As a result, these Terms
of Service help de ne Google’s relationship with you as
```

4.Text Summarization using transformers Pipeline with t5-small

This code section performs automatic text summarization using the pipeline function from the transformers library. It takes a portion of document_text, generates a summary using the t5-small model, and prints the output. Below is a detailed explanation of what happens when each line runs.



```
from transformers import pipeline

summarizer=pipeline("summarization",model="t5-small")

summary=summarizer(document_text[:1000],max_length=150,min_length=30,do_sample=False)
print(summary)
print("Summary:",summary[0]['summary_text'])
```

The line from transformers import pipeline imports the pipeline function from the transformers library. The pipeline function provides an easy way to use pre-trained models for tasks like summarization, question answering, and translation without manually loading tokenizers and models.

Next, summarizer=pipeline("summarization",model="t5-small") creates a summarization pipeline and assigns it to the variable summarizer.

- "summarization" tells the pipeline that the task is text summarization.
- model="t5-small" specifies that the pre-trained t5-small model will be used for generating summaries.
- When this line runs, the t5-small model and its tokenizer are downloaded (if not already cached) and loaded into memory.
- The variable summarizer now becomes a ready-to-use summarization function.

Then, summary=summarizer(document_text[:1000],max_length=150,min_length=30,do_sample=False) generates the summary.

- document_text[:1000] takes only the first 1000 characters from document_text. This is done because transformer models like t5-small have input length limits.
- max_length=150 ensures that the generated summary will not exceed 150 tokens.
- min_length=30 ensures the summary is at least 30 tokens long.
- do_sample=False disables random sampling, meaning the model uses deterministic decoding (greedy decoding) to produce consistent output each time.
- The result is stored in the variable summary.
- The summary variable is actually a list containing a dictionary. That dictionary has a key 'summary_text' which holds the generated summary.

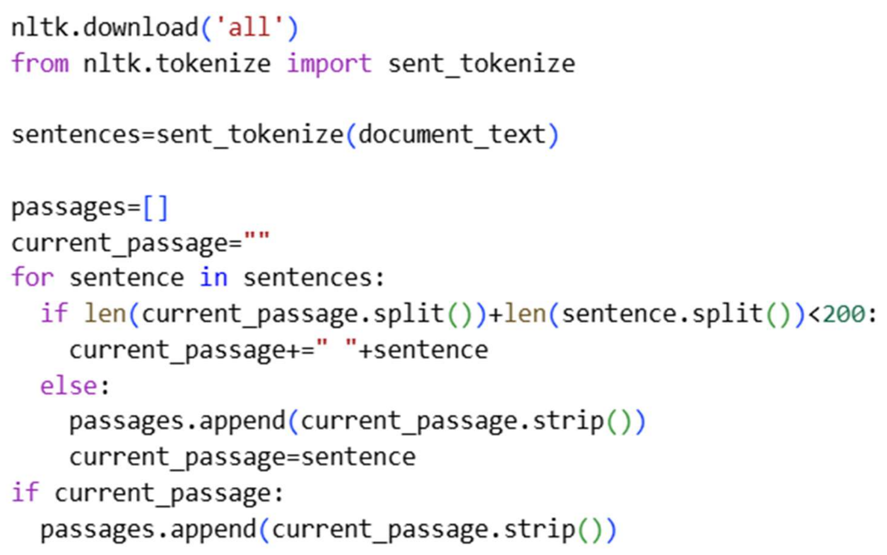
The line print(summary) prints the full raw output returned by the summarizer. This typically looks like a list containing a dictionary structure, such as:

- [{'summary_text': '...generated summary...'}]

Finally, print("Summary:",summary[0]['summary_text']) extracts and prints only the actual summarized text.

- summary[0] accesses the first element of the list.
- ['summary_text'] retrieves the summary string from the dictionary.
- "Summary:" is printed before the actual summarized text for clarity.

5.Sentence Tokenization and Passage Creation using nltk



```
nltk.download('all')
from nltk.tokenize import sent_tokenize

sentences=sent_tokenize(document_text)

passages=[]
current_passage=""
for sentence in sentences:
  if len(current_passage.split())+len(sentence.split())<200:
    current_passage+=" "+sentence
  else:
    passages.append(current_passage.strip())
    current_passage=sentence
if current_passage:
  passages.append(current_passage.strip())
```

This code section downloads required NLTK resources, splits document_text into sentences using sent_tokenize, and then groups those sentences into smaller passages with a word limit of approximately 200 words. Below is a detailed, line-by-line explanation of what happens when each line runs.

The line nltk.download('all') downloads all available datasets, tokenizers, corpora, and resources from the nltk library.

- This ensures that functions like sent_tokenize work properly.
- When this line runs for the first time, it downloads required files such as the Punkt tokenizer model used for sentence splitting.

Next, from nltk.tokenize import sent_tokenize imports the sent_tokenize function from nltk.tokenize.

- sent_tokenize is used to split a large block of text into individual sentences based on punctuation and language rules.

The line sentences=sent_tokenize(document_text) applies sentence tokenization to document_text.

- sent_tokenize(document_text) breaks the full document into a list of sentences.
- The result is stored in the variable sentences.
- Now, sentences contains each sentence as a separate string inside a list.

Then, passages=[] initializes an empty list called passages.

- This list will store grouped text segments (each around 200 words).

The line current_passage="" creates an empty string variable named current_passage.

- This variable temporarily collects sentences before adding them to passages.

The loop for sentence in sentences: starts iterating through each sentence in the sentences list.

- During each iteration, one sentence is processed at a time.

Inside the loop, the condition
if len(current_passage.split())+len(sentence.split())<200: checks whether adding the current sentence to current_passage would keep the total word count under 200 words.

- current_passage.split() splits current_passage into words.
- len(current_passage.split()) counts the number of words currently in current_passage.
- sentence.split() splits the current sentence into words.

- len(sentence.split()) counts the words in that sentence.
- The sum ensures the combined word count stays below 200.

If the condition is true,
current_passage+=" "+sentence appends the sentence to current_passage.

- " " ensures there is a space before adding the sentence.
- This gradually builds a passage until it nears 200 words.

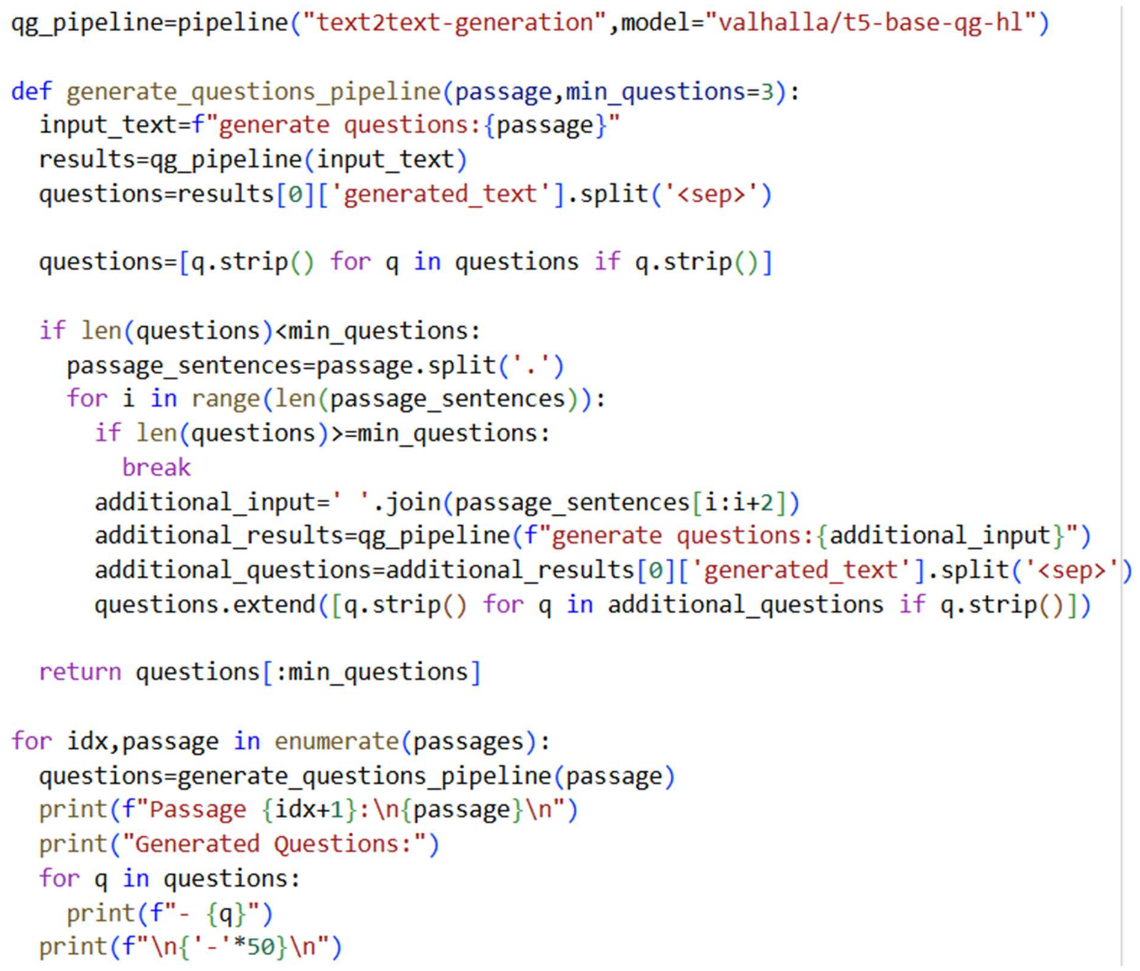
If the condition is false (meaning adding the sentence would exceed 200 words), the else: block executes:

- passages.append(current_passage.strip()) adds the current completed passage to the passages list.
- .strip() removes extra leading or trailing spaces.
- current_passage=sentence resets current_passage to start a new passage with the current sentence.

After the loop finishes processing all sentences, the condition if current_passage: checks whether there is any remaining text stored in current_passage.

- If it is not empty, passages.append(current_passage.strip()) adds the final passage to the passages list.
- This ensures no text is left out.

6.Question Generation Pipeline Using T5 (valhalla/t5-base-qg-hl)



```python
qg_pipeline=pipeline("text2text-generation",model="valhalla/t5-base-qg-hl")

def generate_questions_pipeline(passage,min_questions=3):
  input_text=f"generate questions:{passage}"
  results=qg_pipeline(input_text)
  questions=results[0]['generated_text'].split('<sep>')

  questions=[q.strip() for q in questions if q.strip()]

  if len(questions)<min_questions:
    passage_sentences=passage.split('.')
    for i in range(len(passage_sentences)):
      if len(questions)>=min_questions:
        break
      additional_input=' '.join(passage_sentences[i:i+2])
      additional_results=qg_pipeline(f"generate questions:{additional_input}")
      additional_questions=additional_results[0]['generated_text'].split('<sep>')
      questions.extend([q.strip() for q in additional_questions if q.strip()])

  return questions[:min_questions]

for idx,passage in enumerate(passages):
  questions=generate_questions_pipeline(passage)
  print(f"Passage {idx+1}:\n{passage}\n")
  print("Generated Questions:")
  for q in questions:
    print(f"- {q}")
  print(f"\n{'-'*50}\n")
```

This complete code builds an automated question generation workflow using a transformer-based model and then applies it to multiple passages. The execution begins by initializing a pretrained model, defining a function to generate questions from a given passage, ensuring a minimum number of questions are produced, and finally looping through all passages to display structured output. Below is a detailed explanation of what happens when each line runs, written in clear paragraph form while using the exact words from your code.

The execution starts when qg_pipeline = pipeline("text2text-generation", model="valhalla/t5-base-qg-hl") runs. At this moment, the pipeline function loads a transformer model configured for "text2text-generation". The model "valhalla/t5-base-qg-hl" is specifically trained for question generation tasks. When this line executes, the model weights are loaded into memory, tokenizer settings are initialized, and the object qg_pipeline becomes a ready-to-use inference pipeline. From this point forward, any text passed into qg_pipeline() will be processed by this model to generate questions. This line essentially activates the AI engine of the system.

Next, def generate_questions_pipeline(passage, min_questions=3): defines a function named generate_questions_pipeline. The parameter passage represents the input text from which questions will be generated, and min_questions=3 sets a default minimum number of questions required. When this function is defined, Python stores its logic but does not execute it yet. The function will only run when it is called later inside the loop.Inside the function, the line input_text = f"generate questions:{passage}" creates a formatted string. The phrase "generate questions:" acts as an instruction prompt to guide the model, and {passage} inserts the actual passage content. When this line runs, the variable input_text now contains a structured instruction telling the model to generate questions from the given passage. This step is crucial because T5 models rely heavily on task-specific prompts.

When results = qg_pipeline(input_text) executes, the input_text is passed into the model. The model tokenizes the input, processes it through multiple transformer layers, and generates output text. The result is returned as a list containing dictionaries. This output is stored in the variable results. At this point, the model has produced raw generated text, but it is not yet structured into individual questions.

The next line, questions = results[0]['generated_text'].split('<sep>'), extracts the generated text from the first dictionary inside results. The key 'generated_text' contains all generated questions combined into one string. The method .split('<sep>') separates the string wherever the <sep> token appears. The <sep> token is used by the model to separate multiple questions. After this line runs, questions becomes a list containing individual question strings.Immediately after that, questions = [q.strip() for q in questions if q.strip()] performs cleaning. This list comprehension iterates through each q in questions, applies q.strip() to remove extra whitespace, and keeps only those entries where q.strip() is not empty. When this line completes, the questions list contains only clean, properly formatted question strings without blank entries.

The condition if len(questions) < min_questions: checks whether the number of generated questions is less than the required min_questions. If this condition is true, the code attempts to generate additional questions. First, passage_sentences = passage.split('.') divides the passage into smaller sentence-level chunks. This creates a list called passage_sentences.

Then a for loop runs using for i in range(len(passage_sentences)):. This loop iterates through each sentence index. Inside the loop, the condition if len(questions) >= min_questions: checks whether enough questions have already been collected. If enough questions exist, break immediately stops further execution of the loop.

If more questions are still needed, additional_input = ' '.join(passage_sentences[i:i+2]) combines two consecutive sentences into one string. This smaller chunk is then passed again into the model using additional_results = qg_pipeline(f"generate questions:{additional_input}"). The newly generated output is extracted using additional_results[0]['generated_text'].split('<sep>'), and cleaned using another list comprehension. Finally, questions.extend(...) adds these new questions to the existing questions list. This entire block ensures that the function tries its best to reach at least min_questions.

Once the required number of questions is available, the line return questions[:min_questions] executes. This slices the questions list and returns only the first min_questions elements. Even if extra questions were generated, only the required number is returned. This guarantees consistent output size.

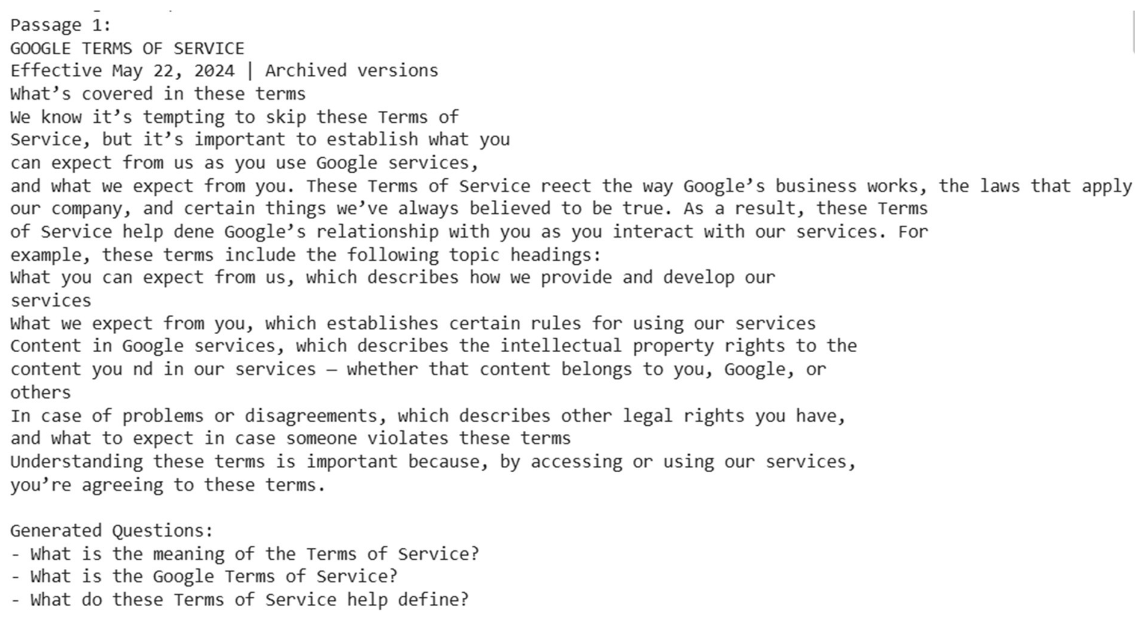
After the function definition, execution moves to the loop for idx, passage in enumerate(passages):. The enumerate(passages) function provides both the index idx and the actual passage content during each iteration. For every passage inside passages, the function generate_questions_pipeline(passage) is called, and the result is stored in questions.

The line print(f"Passage {idx+1}:\n{passage}\n") prints the passage number and the actual passage content. Then print("Generated Questions:") prints a heading. The loop for q in questions: iterates through each generated question, and print(f"- {q}") prints each question in bullet-style format. Finally, print(f"\n{'-'*50}\n") prints a separator line of 50 dashes, making the output visually structured and easy to read.

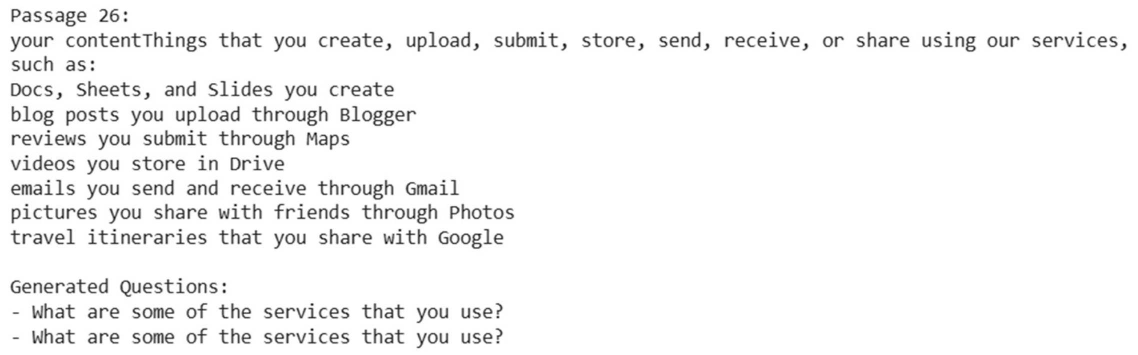
Overall Execution Flow

When the complete code runs:

- The pipeline loads "valhalla/t5-base-qg-hl".
- The function generate_questions_pipeline is defined.
- Each passage inside passages is processed.
- Questions are generated and cleaned.
- Additional questions are generated if needed.
- Exactly min_questions are returned.
- The passage and its generated questions are printed in a structured format.

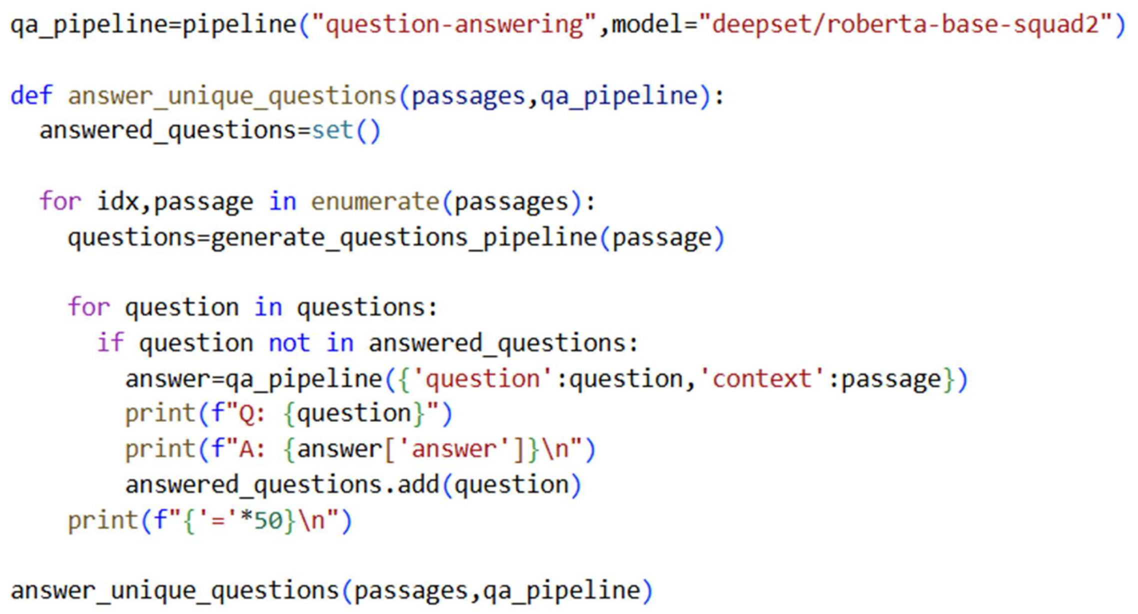


```
Passage 1:
GOOGLE TERMS OF SERVICE
Effective May 22, 2024 | Archived versions
What’s covered in these terms
We know it’s tempting to skip these Terms of
Service, but it’s important to establish what you
can expect from us as you use Google services,
and what we expect from you. These Terms of Service reect the way Google’s business works, the laws that apply
our company, and certain things we’ve always believed to be true. As a result, these Terms
of Service help dene Google’s relationship with you as you interact with our services. For
example, these terms include the following topic headings:
What you can expect from us, which describes how we provide and develop our
services
What we expect from you, which establishes certain rules for using our services
Content in Google services, which describes the intellectual property rights to the
content you nd in our services — whether that content belongs to you, Google, or
others
In case of problems or disagreements, which describes other legal rights you have,
and what to expect in case someone violates these terms
Understanding these terms is important because, by accessing or using our services,
you’re agreeing to these terms.

Generated Questions:
- What is the meaning of the Terms of Service?
- What is the Google Terms of Service?
- What do these Terms of Service help define?
```



```
Passage 26:
your contentThings that you create, upload, submit, store, send, receive, or share using our services,
such as:
Docs, Sheets, and Slides you create
blog posts you upload through Blogger
reviews you submit through Maps
videos you store in Drive
emails you send and receive through Gmail
pictures you share with friends through Photos
travel itineraries that you share with Google

Generated Questions:
- What are some of the services that you use?
- What are some of the services that you use?
```

7.Implementation of the Unique Question Answering Function Using RoBERTa-SQuAD2 Pipeline



```python
qa_pipeline=pipeline("question-answering",model="deepset/roberta-base-squad2")

def answer_unique_questions(passages,qa_pipeline):
  answered_questions=set()

  for idx,passage in enumerate(passages):
    questions=generate_questions_pipeline(passage)

    for question in questions:
      if question not in answered_questions:
        answer=qa_pipeline({'question':question,'context':passage})
        print(f"Q: {question}")
        print(f"A: {answer['answer']}\n")
        answered_questions.add(question)
    print(f"{'='*50}\n")

answer_unique_questions(passages,qa_pipeline)
```

This complete code builds a Question Answering system using a transformer model and ensures that only unique questions are answered across multiple passages. The workflow initializes a qa_pipeline, defines a function named answer_unique_questions, tracks already answered questions using a set, and finally processes each passage to generate and answer questions. Below is a detailed explanation of what happens when each line runs, written in paragraph format using the exact words from your code.

The execution begins with qa_pipeline = pipeline("question-answering", model="deepset/roberta-base-squad2"). When this line runs, the pipeline function loads a pretrained model configured for "question-answering". The model "deepset/roberta-base-squad2" is specifically fine-tuned on the SQuAD2 dataset, meaning it is trained to extract precise answers from a given context. At execution time, the tokenizer and model weights are loaded into memory, and the object qa_pipeline becomes a ready-to-use question-answering system. From this point onward, any dictionary containing 'question' and 'context' passed into qa_pipeline() will return an extracted answer.

Next, the function def answer_unique_questions(passages, qa_pipeline): is defined. This function takes two parameters: passages, which contains multiple text inputs, and qa_pipeline, which is the question-answering model. When this definition runs, Python stores the function logic but does not execute it yet. The function will only execute when it is called at the end.

Inside the function, the line answered_questions = set() initializes an empty set. When this line runs, a new set object is created in memory. The purpose of answered_questions is to store each question that has already been answered. Since a set does not allow duplicate values, it automatically ensures uniqueness. This is important for preventing repeated answers.

Then the loop for idx, passage in enumerate(passages): begins. When this loop runs, enumerate(passages) provides both the index idx and the actual passage content during each iteration. This means the function processes one passage at a time.

Inside this loop, the line questions = generate_questions_pipeline(passage) executes. Here, the previously defined generate_questions_pipeline function is called with the current passage. When this line runs, the question generation model creates multiple questions based on that passage. The resulting list of questions is stored in the variable questions.

Next, another loop begins with for question in questions:. This loop iterates through each question generated from the current passage. During each iteration, the code checks whether that specific question has already been answered before.

The condition if question not in answered_questions: runs for each question. When this condition is evaluated:

- If the question is not present in answered_questions, the code inside the block executes.
- If the question is already in the set, it is skipped automatically.

This ensures that duplicate questions across different passages are not answered again.

When a new question is detected, the line answer = qa_pipeline({'question': question, 'context': passage}) executes. At this point:

- A dictionary containing 'question': question and 'context': passage is passed into qa_pipeline.
- The model processes the context and searches for the most relevant answer span.
- The output is returned as a dictionary.
- This dictionary is stored in the variable answer.

Then, print(f"Q: {question}") prints the current question in a formatted style. Immediately after, print(f"A: {answer['answer']}\n") extracts the 'answer' value from the answer dictionary and prints it. The \n ensures a blank line after each answer for readability.

After printing, the line answered_questions.add(question) runs. This adds the current question to the answered_questions set. From this point forward, if the same question appears again in another passage, it will not be answered again because it already exists in the set.

Once all questions in the current passage are processed, the line print(f"{'='*50}\n") executes. This prints a separator line of 50 equal signs. This improves output formatting and visually separates results between passages.

Finally, outside the function definition, the line answer_unique_questions(passages, qa_pipeline) calls the function. When this line runs:

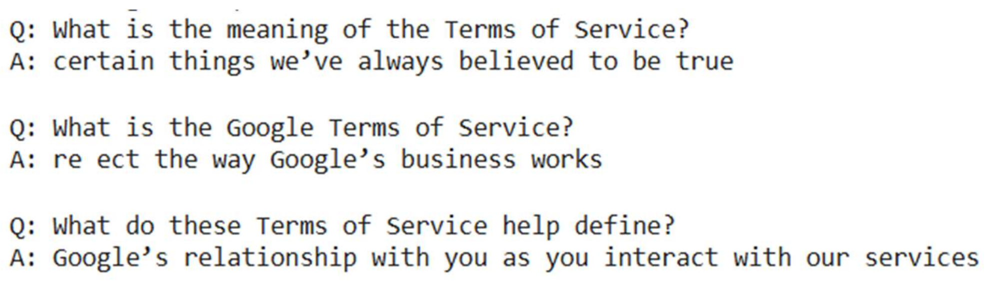
- The entire logic inside answer_unique_questions executes.
- Each passage is processed.
- Questions are generated.
- Unique questions are answered.
- Answers are printed.

- Duplicate questions are skipped.

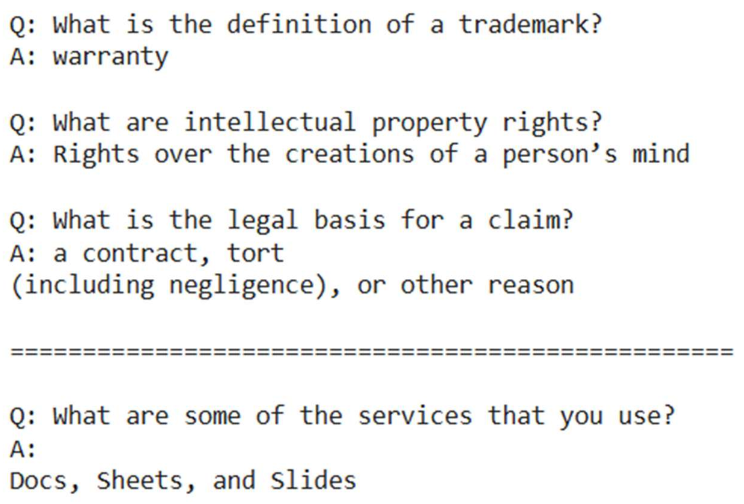
Overall Execution Flow

When the complete code runs:

- The pipeline loads "deepset/roberta-base-squad2" for "question-answering".
- The function answer_unique_questions is defined.
- An empty set named answered_questions is created.
- Each passage in passages is processed.
- Questions are generated using generate_questions_pipeline.
- Only unique questions are answered.
- Each question and its extracted answer are printed.
- A separator line organizes the output.



```
Q: What is the meaning of the Terms of Service?
A: certain things we've always believed to be true

Q: What is the Google Terms of Service?
A: re ect the way Google's business works

Q: What do these Terms of Service help define?
A: Google's relationship with you as you interact with our services
```



```
Q: What is the definition of a trademark?
A: warranty

Q: What are intellectual property rights?
A: Rights over the creations of a person's mind

Q: What is the legal basis for a claim?
A: a contract, tort
(including negligence), or other reason

==================================================

Q: What are some of the services that you use?
A:
Docs, Sheets, and Slides
```