The script is designed to:

1. Convert a PDF document into Markdown chunks.

2. Identify the top relevant chunks of text based on a user-provided prompt using AI-based semantic similarity.

3. Generate a concise, precise answer by using an AI language model (GPT-2) based on the relevant chunks.

4. Output the top relevant chunks along with the generated precise answer.

Step-by-Step Process

1. Mount Google Drive

- Code:

from google.colab import drive

drive.mount('/content/drive')

- Explanation:

This step mounts your Google Drive to the Colab environment, allowing the script to access files stored in your Drive. The PDF document is expected to be in your Google Drive, which the script will later process.

2. Convert PDF to Markdown Chunks

- Code:

```python

def pdf\_to\_markdown\_chunks(pdf\_path, chunk\_size=500):

```

- Explanation:

This function takes the PDF file and converts its text into Markdown format.

- PDF Reading: The `PyPDF2` library is used to extract text from each page of the PDF.

- Markdown Conversion: The extracted text is converted to Markdown using the `markdownify` library.

- Chunking: The Markdown text is divided into smaller chunks, each containing approximately 500 characters. This chunking is necessary because AI models often work better with smaller, contextually related pieces of text rather than long, continuous text.

3. Find Top Relevant Chunks

- Code:

```python

def find\_top\_relevant\_chunks(prompt, chunk\_texts, top\_n=3):

...

```

- Explanation:

This function identifies the top `n` relevant text chunks based on a given prompt.

- Model: The `SentenceTransformer` model (`all-MiniLM-L6-v2`) is used to encode both the user’s prompt and the text chunks into numerical embeddings (vectors).

- Similarity Calculation: The cosine similarity between the prompt's embedding and each chunk's embedding is calculated using `torch`. The chunks with the highest similarity scores are considered the most relevant.

- Output: The function returns the top `n` most relevant chunks and their similarity scores.

4. Generate a Precise Answer

- Code:

```python

def generate\_precise\_answer(prompt, top\_chunks):

...

```

- Explanation:

This function uses the `GPT-2` model to generate a precise answer based on the top relevant chunks.

- Tokenization: The prompt and the top chunks are concatenated and tokenized using `GPT2Tokenizer`.

- Model Inference: The tokenized input is passed to the `GPT2LMHeadModel` to generate a textual response. The `max\_length` and `no\_repeat\_ngram\_size` parameters are set to control the length and repetition in the generated text.

- Post-Processing: The generated text is further cleaned to remove excessive repetitions.

5. Process the Question and Provide Output

- Code:

```python

def process\_question(pdf\_path, prompt):

...

```

- Explanation:

This function orchestrates the entire process.

- Text Chunking: It first converts the PDF to Markdown and divides it into chunks.

- Top Chunks Identification: It then finds the top 3 most relevant chunks based on the user's prompt.

- Answer Generation: The precise answer is generated using the GPT-2 model.

- Combining Output: The function combines the precise answer with the top relevant chunks and prepares the final output.

6. Example Usage

- Code:

```python

pdf\_path = '/content/drive/MyDrive/ConvAI/intellectual\_property\_policy.pdf'

prompt = "Explain IP revenue distribution after death"

precise\_answer = process\_question(pdf\_path, prompt)

print(precise\_answer)

```

- Explanation:

In this example, the script is run with a specific PDF and a prompt. It processes the PDF to extract and analyze relevant information, generates a precise answer, and then prints the answer along with the relevant chunks that contributed to it.

How It Works in Detail

1. Data Storage:

- The PDF file is stored in your Google Drive, which is accessed through the Colab environment.

- The processed Markdown chunks are stored temporarily in memory during the script execution.

2. Processing Flow:

- The PDF is read, and the text is extracted and converted to Markdown.

- The text is then split into chunks to handle better by the AI models.

- A semantic similarity search is conducted to find the most relevant chunks of text in response to the prompt.

- These chunks are then used to form a coherent and precise response with the help of GPT-2.

3. Generating the Response:

- The response generation is AI-driven, where the prompt guides the AI model to focus on specific aspects of the text.

- By focusing on the most relevant chunks, the model produces a more accurate and concise response.

Considerations & Limitations

- Text Format Handling:

- The script is designed primarily for text and may struggle with complex formatting such as tables, images, or diagrams in the PDF. The conversion from PDF to Markdown might not accurately capture non-text elements, leading to less reliable results in those areas.

- Model Capabilities:

- While the GPT-2 model is powerful, it's not perfect. It might introduce repetitions or produce less coherent answers in certain cases. Post-processing helps, but the results can still vary.

- Scalability:

- This script works well for relatively small PDFs and chunks of text. For larger documents or more complex queries, performance might be an issue, requiring more powerful models or optimized chunking strategies.

**How to Install Streamlit Locally Using PowerShell:**

1. **Open PowerShell:**
   * Search for PowerShell in your Start Menu and open it.
2. **Ensure Python is Installed:**
   * You can check if Python is installed by running:

bash

Copy code

python --version

* + If Python is not installed, download and install it from the [official website](https://www.python.org/downloads/).

1. **Create a Virtual Environment (Optional but Recommended):**
   * Navigate to your project directory:

bash

Copy code

cd path\_to\_your\_project\_directory

* + Create a virtual environment:

bash

Copy code

python -m venv myenv

* + Activate the virtual environment:

bash

Copy code

.\myenv\Scripts\Activate

1. **Install Streamlit:**
   * Install Streamlit using pip:

bash

Copy code

pip install streamlit

1. **Run a Streamlit App:**
   * Create a .py file with your Streamlit code or use a sample Streamlit file.
   * Run the app with:

bash

Copy code

streamlit run your\_script.py

1. **Access the App:**
   * After running the command, Streamlit will start a local server. Open the provided local URL (usually http://localhost:8501) in your web browser to view the app.

This setup will give you a persistent, reliable development environment for working with Streamlit.

**1. Activate Your Virtual Environment:**

First, activate the virtual environment where you installed Streamlit.

* On **Windows:**

cmd

Copy code

.\venv\Scripts\activate

* On **macOS/Linux:**

bash

Copy code

source venv/bin/activate

Replace venv with the name of your virtual environment if it's different.

**2. Create a Python File for Your Streamlit App:**

In your virtual environment, navigate to the directory where you want to store your Streamlit application. Create a new Python file, for example, app.py.

* On **Windows Command Prompt:**

cmd

Copy code

notepad app.py

* On **macOS/Linux (or using a text editor in the terminal):**

bash

Copy code

nano app.py

Alternatively, you can use any code editor (like VS Code, PyCharm, etc.) to create and edit this file.

**3. Copy Your Code into the Python File:**

Paste your Streamlit code into app.py. For example:

python

Copy code

import streamlit as st

st.title('Hello, Streamlit!')

st.write('This is a simple Streamlit app.')

Save the file after pasting the code.

**4. Run Your Streamlit Application:**

With your virtual environment activated, navigate to the directory containing app.py, and run the Streamlit application using the following command:

bash

Copy code

streamlit run app.py

This command will start a local web server, and Streamlit will display the URL where your app is running (e.g., http://localhost:8501). Open this URL in your web browser to view your Streamlit app.

**5. Deploy Your App (Optional):**

If you want to deploy your Streamlit app so that others can access it online, you can use platforms like **Streamlit Community Cloud**, **Heroku**, or **Azure**. Here's a brief overview of how to deploy on Heroku:

**5.1. Prepare Your App for Heroku:**

* Create a requirements.txt file to list your app's dependencies. You can generate it using:

bash

Copy code

pip freeze > requirements.txt

* Create a Procfile with the following content:

bash

Copy code

web: streamlit run app.py

**5.2. Deploy on Heroku:**

* Install the Heroku CLI if you haven't already.
* Log in to Heroku:

bash

Copy code

heroku login

* Create a new Heroku app:

bash

Copy code

heroku create your-app-name

* Push your code to Heroku:

bash

Copy code

git init

git add .

git commit -m "Initial commit"

heroku git:remote -a your-app-name

git push heroku master

* Open your app in the browser:

bash

Copy code

heroku open

**Note:** Deploying on Heroku isn't mandatory for running the app locally. If you only need local execution, steps 1-4 are sufficient.

**Summary:**

* Activate your virtual environment.
* Create and edit the app.py file with your Streamlit code.
* Run the app locally using streamlit run app.py.
* (Optional) Deploy on a platform like Heroku for remote access.