Build a ChatGPT for PDFs with Langchain

BEGINNER CHATBOT CHATGPT PYTHON

Introduction

In just six months, OpenAI's ChatGPT has become an integral part of our lives. It's not just limited to tech anymore; people of all ages and professions, from students to writers, are using it extensively. These chat models excel in accuracy, speed, and human-like conversation. They are poised to play a significant role in various fields, not just technology.

Open-source tools like AutoGPTs, BabyAGI, and Langchain have emerged, harnessing the power of language models. Automate programming tasks with prompts, connect language models to data sources, and create AI applications faster than ever before. Langchain is a ChatGPT-enabled Q&A tool for PDFs, making it a one-stop shop for building AI applications.

> ChatGPT: Unlocking the Potential of Artificial Intelligence for Human-Like Conversation

Learning Objectives

- Build a chatbot interface using Gradio
- Extract texts from pdfs and create embeddings
- Store embeddings in the Chroma vector database
- Send query to the backend (Langchain chain)
- Perform semantic search over texts to find relevant sources of data
- Send data to LLM (ChatGPT) and receive answers on the chatbot

The Langchain makes it easy to perform all these steps in a few lines of code. It has wrappers for multiple services, including embedding models, chat models, and vector databases.

This article was published as a part of the Data Science Blogathon.

Table of contents

What is Langchain?

Langchain is an open-source tool written in Python that helps connect external data to Large Language Models. It makes the chat models like GPT-4 or GPT-3.5 more agentic and data-aware. So, in a way, Langchain provides a way for feeding LLMs with new data that it has not been trained on. Langchain provides many chains which abstract away complexities in interacting with language models. We also need several other tools, like Models for creating vector embeddings and vector databases to store vectors. Before proceeding further, let's have a quick look at text embeddings. What are these and why it is important?

Text Embeddings

Text embeddings are the heart and soul of Large Language Operations. Technically, we can work with language models with natural language but storing and retrieving natural language is highly inefficient. For example, in this project, we will need to perform high-speed search operations over large chunks of data. It is impossible to perform such operations on natural language data.

To make it more efficient, we need to transform text data into vector forms. There are dedicated ML models for creating embeddings from texts. The texts are converted into multidimensional vectors. Once embedded, we can group, sort, search, and more over these data. We can calculate the distance between two sentences to know how closely they are related. And the best part of it is these operations are not just limited to keywords like the traditional database searches but rather capture the semantic closeness of two sentences. This makes it a lot more powerful, thanks to Machine Learning.

Langchain Tools

Langchain has wrappers for all major vector databases like Chroma, Redis, Pinecone, Alpine db, and more. And same is true for LLMs, along with OpeanAI models, it also supports Cohere's models, GPT4ALL- an open-source alternative for GPT models. For embeddings, it provides wrappers for OpeanAI, Cohere, and HuggingFace embeddings. You can also use your custom embedding models as well.

So, in short, Langchain is a meta-tool that abstracts away a lot of complications of interacting with underlying technologies, which makes it easier for anyone to build AI applications quickly.

In this article, we will use the **OpeanAI embeddings** model for creating embeddings. If you want to deploy an AI app for end users, consider using any Opensource models, such as Huggingface models or Google's Universal sentence encoder.

To store vectors, we will use **Chroma DB**, an open-source vector store database. Feel free to explore other databases like Alpine, Pinecone, and Redis. Langchain has wrappers for all of these vector stores.

To create a Langchain chain, we will use **ConversationalRetrievalChain**(), ideal for conversation with chat models with history (to keep the context of the conversation). Do check out their official documentation regarding different LLM chains.

Set-up Dev Environment

There are quite a few libraries we will use. So, install them beforehand. To create a seamless, clutter-free development environment, use virtual environments or Docker.

```
gradio = "^3.27.0" openai = "^0.27.4" langchain = "^0.0.148" chromadb = "^0.3.21" tiktoken = "^0.3.3" pypdf = "^3.8.1" pymupdf = "^1.22.2"
```

Now, import these libraries

```
import gradio as gr from langchain.embeddings.openai import OpenAIEmbeddings from langchain.text_splitter import CharacterTextSplitter from langchain.vectorstores import Chroma from langchain.chains import ConversationalRetrievalChain from langchain.chat_models import ChatOpenAI from langchain.document_loaders import PyPDFLoader import os import fitz from PIL import Image
```

Build Chat Interface

The interface of the application will have two major functionalities, one is a chat interface, and the other renders the relevant page of the PDF as an image. Apart from this, a text box for accepting OpenAI API keys from end users. I would highly recommend going through the article for building a GPT chatbot with Gradio from scratch. The article discusses the fundamental aspects of Gradio. We will borrow a lot of things from this article.

Gradio Blocks class allows us to build a web app. The Row and Columns classes allow for aligning multiple components on the web app. We will use them to customize the web interface.

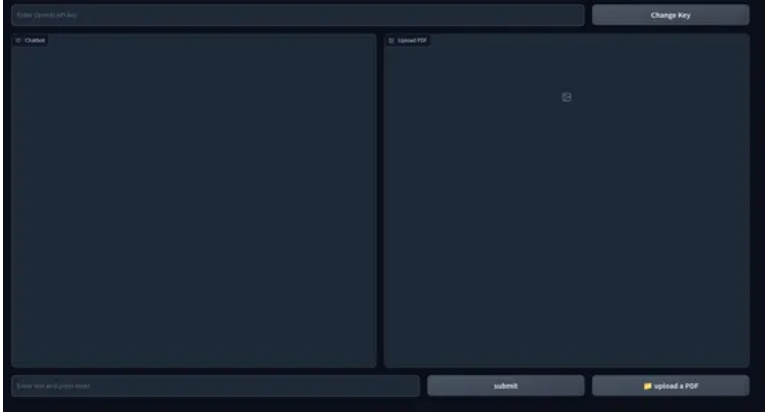
```
with gr.Blocks() as demo: # Create a Gradio block with gr.Column(): with gr.Row(): with gr.Column(scale=0.8): api_key = gr.Textbox( placeholder='Enter OpenAI API key', show_label=False, interactive=True ).style(container=False) with gr.Column(scale=0.2): change_api_key = gr.Button('Change Key') with gr.Row(): chatbot = gr.Chatbot(value=[], elem_id='chatbot').style(height=650) show_img = gr.Image(label='Upload PDF', tool='select').style(height=680) with gr.Row(): with gr.Column(scale=0.70): txt = gr.Textbox( show_label=False, placeholder="Enter text and press enter" ).style(container=False) with gr.Column(scale=0.15): submit_btn = gr.Button('Submit') with gr.Column(scale=0.15): btn = gr.UploadButton("📁 Upload a PDF", file_types=[".pdf"]).style()
```

The interface is simple with a few components.

It has:

- A chat interface to communicate with the PDF.
- A component for rendering relevant PDF pages.
- A text box for accepting the API key and a change key button.
- A text box for asking questions and a submit button.
- A button for uploading files.

Here is a snapshot of the web UI.



The frontend part of our application is complete. Let's hop on to the backend.

Backend

First, let's outline the processes we will be dealing with.

- Handle uploaded PDF and OpenAI API key
- Extract texts from PDF and create text embeddings out of it using OpenAI embeddings.
- Store vector embeddings in the ChromaDB vector store.
- Create a Conversational Retrieval chain with Langchain.
- Create embeddings of queried text and perform a similarity search over embedded documents.
- Send relevant documents to the OpenAI chat model (gpt-3.5-turbo).
- Fetch the answer and stream it on chat UI.
- Render relevant PDF page on Web UI.

These are the overview of our application. Let's start building it.

Gradio Events

When a specific action on the web UI is performed, these events are triggered. So, the events make the web app interactive and dynamic. Gradio allows us to define events with Python codes.

Gradio Events use component variables that we defined earlier to communicate with the backend. We will define a few Events that we need for our application. These are

- **Submit API key event**: Pressing enter after pasting the API key will trigger this event.
- **Change Key**: This will allow you to provide a new API key
- **Enter Queries**: Submit text queries to the chatbot
- **Upload File**: This will allow the end user to upload a PDF file

```
with gr.Blocks() as demo: # Create a Gradio block with gr.Column(): with gr.Row(): with gr.Column(scale=0.8): api_key = gr.Textbox( placeholder='Enter OpenAI API key', show_label=False, interactive=True ).style(container=False) with gr.Column(scale=0.2): change_api_key = gr.Button('Change Key') with gr.Row(): chatbot = gr.Chatbot(value=[], elem_id='chatbot').style(height=650) show_img = gr.Image(label='Upload PDF', tool='select').style(height=680) with gr.Row(): with gr.Column(scale=0.70): txt = gr.Textbox( show_label=False, placeholder="Enter text and press enter" ).style(container=False) with gr.Column(scale=0.15): submit_btn = gr.Button('Submit') with gr.Column(scale=0.15): btn = gr.UploadButton("
```

```
Upload a PDF", file_types=[".pdf"]).style() # Set up event handlers # Event handler for submitting the OpenAI API key api_key.submit(fn=set_apikey, inputs=[api_key], outputs=[api_key]) # Event handler for changing the API key change_api_key.click(fn=enable_api_box, outputs=[api_key]) # Event handler for uploading a PDF btn.upload(fn=render_first, inputs=[btn], outputs=[show_img]) # Event handler for submitting text and generating response submit_btn.click( fn=add_text, inputs=[chatbot, txt], outputs=[chatbot], queue=False ).success( fn=generate_response, inputs=[chatbot, txt, btn], outputs=[chatbot, txt] ).success( fn=render_file, inputs=[btn], outputs=[show_img] )
```

So far we have not defined our functions called inside above event handlers. Next, we will define all these functions to make a functional web app.

Handle API Keys

Handling the API keys of a user is important as the entire thing runs on the BYOK(Bring Your Own Key) principle. Whenever a user submits a key, the textbox must become immutable with a prompt suggesting the key is set. And when the “Change Key” event is triggered the box must be able to take inputs.

To do this, define two global variables.

```
enable_box = gr.Textbox.update(value=None,placeholder= 'Upload your OpenAI API key', interactive=True) disable_box = gr.Textbox.update(value = 'OpenAI API key is Set',interactive=False)
```

Define functions

```
def set_apikey(api_key): os.environ['OPENAI_API_KEY'] = api_key return disable_box def enable_api_box(): return enable_box
```

The set_apikey function takes a string input and returns the disable_box variable, which makes the textbox immutable after execution. In the Gradio Events section, we defined the api_key Submit Event, which calls the set_apikey function. We set the API key as an environment variable using the OS library.

Clicking the Change API key button returns the enable_box variable, which enables the mutability of the textbox again.

Create Chain

This is the most important step. This step involves extracting texts and creating embeddings and storing them in vector stores. Thanks to Langchain, which provides wrappers for multiple services making things easier. So, let’s define the function.

```
def process_file(file): # raise an error if API key is not provided if 'OPENAI_API_KEY' not in os.environ: raise gr.Error('Upload your OpenAI API key') # Load the PDF file using PyPDFLoader loader = PyPDFLoader(file.name) documents = loader.load() # Initialize OpenAIEmbeddings for text embeddings embeddings = OpenAIEmbeddings() # Create a ConversationalRetrievalChain with ChatOpenAI language model # and PDF search retriever pdfsearch = Chroma.from_documents(documents, embeddings,) chain = ConversationalRetrievalChain.from_llm(ChatOpenAI(temperature=0.3), retriever= pdfsearch.as_retriever(search_kwargs={"k": 1}), return_source_documents=True,) return chain
```

- Created a check if the API key is set or not. This will raise an error on the front end if the Key is not set.
- Load PDF file using PyPDFLoader

- Defined embeddings function with OpenAIEmbeddings.
- Created a vector store from the list of texts from the PDF using the embedding function.
- Defined a chain with the chatOpenAI(by default ChatOpenAI uses gpt-3.5-turbo), a base retriever (uses a similarity search).

Generate Response

Once the chain is created, we will call the chain and send our queries. Send a chat history along with the queries to keep the context of conversations and stream responses to the chat interface. Let's define the function.

```
def generate_response(history, query, btn): global COUNT, N, chat_history # Check if a PDF file is uploaded if not btn: raise gr.Error(message='Upload a PDF') # Initialize the conversation chain only once if COUNT == 0: chain = process_file(btn) COUNT += 1 # Generate a response using the conversation chain result = chain({"question": query, 'chat_history':chat_history}, return_only_outputs=True) # Update the chat history with the query and its corresponding answer chat_history += [(query, result["answer"])] # Retrieve the page number from the source document N = list(result['source_documents'][0])[1][1]['page'] # Append each character of the answer to the last message in the history for char in result['answer']: history[-1][-1] += char # Yield the updated history and an empty string yield history, ''
```

- Raises an error, if there is no PDF uploaded.
- Calls process_file function only once.
- Sends queries and chat history to the chain
- Retrieves the page number of the most relevant answer.
- Yield responses to the front end.

Render Image of A PDF File

The final step is to render the image of the PDF file with the most relevant answer. We can use the PyMuPdf and PIL libraries to render the images of the document.

```
def render_file(file): global N # Open the PDF document using fitz doc = fitz.open(file.name) # Get the specific page to render page = doc[N] # Render the page as a PNG image with a resolution of 300 DPI pix = page.get_pixmap(matrix=fitz.Matrix(300/72, 300/72)) # Create an Image object from the rendered pixel data image = Image.frombytes('RGB', [pix.width, pix.height], pix.samples) # Return the rendered image return image
```

- Open the file with PyMuPdf's Fitz.
- Get the relevant page.
- Get pix map for the page.
- Create the image from PIL's Image class.

This is everything we need to do for a functional web app for chatting with any PDF.

Putting everything together

```
#import csv import gradio as gr from langchain.embeddings.openai import OpenAIEmbeddings from langchain.text_splitter import CharacterTextSplitter from langchain.vectorstores import Chroma from langchain.chains import ConversationalRetrievalChain from langchain.chat_models import ChatOpenAI from
```

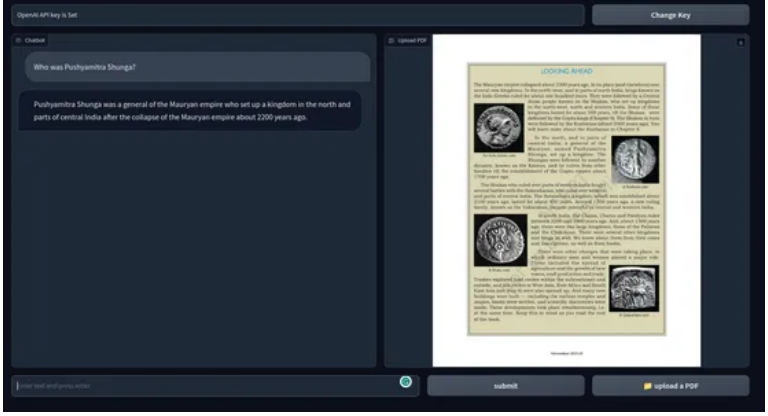
```
langchain.document_loaders import PyPDFLoader import os import fitz from PIL import Image # Global variables COUNT, N = 0, 0 chat_history = [] chain = '' enable_box = gr.Textbox.update(value=None, placeholder='Upload your OpenAI API key', interactive=True) disable_box = gr.Textbox.update(value='OpenAI API key is Set', interactive=False) # Function to set the OpenAI API key def set_apikey(api_key): os.environ['OPENAI_API_KEY'] = api_key return disable_box # Function to enable the API key input box def enable_api_box(): return enable_box # Function to add text to the chat history def add_text(history, text): if not text: raise gr.Error('Enter text') history = history + [(text, '')] return history # Function to process the PDF file and create a conversation chain def process_file(file): if 'OPENAI_API_KEY' not in os.environ: raise gr.Error('Upload your OpenAI API key') loader = PyPDFLoader(file.name) documents = loader.load() embeddings = OpenAIEmbeddings() pdfsearch = Chroma.from_documents(documents, embeddings) chain = ConversationalRetrievalChain.from_llm(ChatOpenAI(temperature=0.3), retriever=pdfsearch.as_retriever(search_kwargs={"k": 1}), return_source_documents=True) return chain # Function to generate a response based on the chat history and query def generate_response(history, query, btn): global COUNT, N, chat_history, chain if not btn: raise gr.Error(message='Upload a PDF') if COUNT == 0: chain = process_file(btn) COUNT += 1 result = chain({"question": query, 'chat_history': chat_history}, return_only_outputs=True) chat_history += [(query, result["answer"])] N = list(result['source_documents'][0])[1][1]['page'] for char in result['answer']: history[-1][-1] += char yield history, '' # Function to render a specific page of a PDF file as an image def render_file(file): global N doc = fitz.open(file.name) page = doc[N] # Render the page as a PNG image with a resolution of 300 DPI pix = page.get_pixmap(matrix=fitz.Matrix(300/72, 300/72)) image = Image.frombytes('RGB', [pix.width, pix.height], pix.samples) return image # Gradio application setup with gr.Blocks() as demo: # Create a Gradio block with gr.Column(): with gr.Row(): with gr.Column(scale=0.8): api_key = gr.Textbox( placeholder='Enter OpenAI API key', show_label=False, interactive=True ).style(container=False) with gr.Column(scale=0.2): change_api_key = gr.Button('Change Key') with gr.Row(): chatbot = gr.Chatbot(value=[], elem_id='chatbot').style(height=650) show_img = gr.Image(label='Upload PDF', tool='select').style(height=680) with gr.Row(): with gr.Column(scale=0.70): txt = gr.Textbox( show_label=False, placeholder="Enter text and press enter" ).style(container=False) with gr.Column(scale=0.15): submit_btn = gr.Button('Submit') with gr.Column(scale=0.15): btn = gr.UploadButton("□ Upload a PDF", file_types=[".pdf"]).style() # Set up event handlers # Event handler for submitting the OpenAI API key api_key.submit(fn=set_apikey, inputs=[api_key], outputs=[api_key]) # Event handler for changing the API key change_api_key.click(fn=enable_api_box, outputs=[api_key]) # Event handler for uploading a PDF btn.upload(fn=render_first, inputs=[btn], outputs=[show_img]) # Event handler for submitting text and generating response submit_btn.click( fn=add_text, inputs=[chatbot, txt], outputs=[chatbot], queue=False ).success( fn=generate_response, inputs=[chatbot, txt, btn], outputs=[chatbot, txt] ).success( fn=render_file, inputs=[btn], outputs=[show_img] ) demo.queue() if __name__ == "__main__": demo.launch()
```

Now that we have configured everything, let's launch our application.

You can launch the application in debug mode with the following command

gradio app.py

Otherwise, you can also simply run the application with the Python command. Below is a snapshot of the end product. GitHub repository of the codes.



Possible Improvements

The current application works great. But there are a few things you can do to make it better.

- This uses OpenAI embeddings which might be expensive in the long run. For a production-ready app, any offline embedding models might be more suitable.
- Gradio for prototyping is fine, but for the real world, an app with a modern javascript framework like Next Js or Svelte would be much better in terms of performance and aesthetics.
- We used cosine similarity for finding relevant texts. In some conditions, a KNN approach might be better.
- For PDFs with dense text content, creating smaller chunks of text might be better.
- Better the model, the better the performance. Experiment with other LLMs and compare the outcomes.

Practical Use Cases

Use the tools across multiple fields from Education to Law to Academia or any field you can imagine that requires the person to go through huge texts. Some of the practical use cases of ChatGPT for PDFs are

- **Educational Institutions**: Students can upload their textbooks, study materials, and assignments, and the tool can answer queries and explain particular sections. This can make the overall learning process less strenuous for students.
- **Legal**: Law firms have to deal with numerous amount of legal documents in PDF formats. This tool can be employed to extract relevant information from case documents, legal contracts, and statutes conveniently. It can help lawyers find clauses, precedents, and other information faster.
- **Academia**: Research scholars often deal with Research papers and technical documentation. A tool that can summarize the literature, analyze and provide answers from documents can go a long way saving overall time and improving productivity.
- **Administration:** Govt. offices and other administrative departments deal with copious amounts of forms, applications, and reports daily. Employing a chatbot that answers documents can streamline the administration process, thus saving everyone’s time and money.

- **Finance**: Analysing financial reports and revisiting them again and again is tedious. This can be made easier by employing a chatbot. Essentially an Intern.
- **Media**: Journalists and Analysts can use a chatGPT-enabled PDF question-answering tool to query large text corpus to find answers quickly.

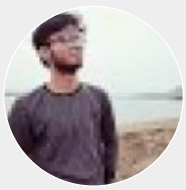
A chatGPT-enabled PDF Q&A tool can gather information faster from heaps of PDF text. It is like a search engine for text data. Not just PDFs, but we can also extend this tool to anything with text data with a little code manipulation.

Conclusion

So, this was all about building a chatbot to converse with any PDF file with ChatGPT. Thanks to Langchain, building AI applications has become far easier. Some of the key takeaways from the article are:

- Gradio is an open-source tool for prototyping AI applications. We created the front end of the application with Gradio.
- Langchain is another open-source tool that allows us to build AI applications. It has wrappers for popular LLMs and vector data stores, which allow us to interact easily with underlying services.
- We used Langchain for building the backend systems of our application.
- OpenAI models were overall crucial for our app. We used the OpenAI embeddings and GPT 3.5 engine to chat with PDFs.
- A ChatGPT-enabled Q&A tool for PDFs and other text data can go a long way in streamlining knowledge tasks.

Article Url - https://www.analyticsvidhya.com/blog/2023/05/build-a-chatgpt-for-pdfs-with-langchain/



Sunil Kumar Dash