

# ChatGPT Summarizer

Building A Simple LLM Application

# Overview of Today's Session

In today's session, we will be building a VERY simple LLM application; a PDF Summarizer application.

In this we will learn:

- How to load and prepare PDF documents for LLM applications.
- Initialize a ChatGPT model and write prompts.
- Use LangChain to set up a QA based LLM with text data.
- And build a Streamlit App to deploy the LLM application.

# What is an LLM?

- Large Language Models (LLMs) are deep learning models, often Transformer-based neural networks, that are trained on large amounts of text, include a very large number of parameters, and are designed to understand and generate human-like text.
- These models ARE NOT REASONING ENGINES as they often work by essentially predicting the next best word in a series rather than actually reasoning or “thinking” like a human.

# LLM Potential

- LLMs are used in a wide range of applications, including language translation, content generation, chatbots, sentiment analysis, and even medical research.
- LLMs have also enabled breakthroughs in machine translation, automated content creation, and have the potential to enhance human-computer interaction.

# LLM Applications

The use of LLMs has exploded in recent years:

- OpenAI, Microsoft, Google, and Meta have all been major players in the field recently with each releasing major LLM developments.
- Many companies are now offering LLM tools and resources for their development including OpenAI, LangChain, HuggingFace, Databricks, AWS, Anthropic, etc.
- LLMs and Generative AI are drastically changing the tech landscape and are already an increasingly in-demand area of expertise.

# Major Models In Play

ChatGPT 3.5 and ChatGPT 4 – OpenAI

LLaMa and LLaMa 2 – Meta

PaLM 2 – Google

Claude – Anthropic

BLOOM – HuggingFace

There are many other models out there that are better or worse for different purposes; below are some resources:

- [Top Open Source LLMs in 2024](#)
- [12 Best Models in 2024](#)

# OpenAI API

- Open AI has made LLMs fairly accessible by developing an API to interact with their AI models.
- To use the OpenAI API:
  - Sign up or log in to your OpenAI account.
  - Generate a new API Key.
  - Install the OpenAI API Python library.
    - `pip install --upgrade openai`
  - Load your key into your environment.
    - Recommended to use an environment (.env) file.
  - Test your OpenAI connection.

# **Building An LLM Application**



# The Main Function

This is the main function for our application. It works by:

- processing a PDF
- building a vector database for the text
- initializing the model
- using LangChain to pass a query and the vector db to the model
- defining the query
- running the model

```
def get_summary(pdf, model, openai_api_key):  
  
    # Loading The PDF Document  
    pdf_doc = load_pdf(pdf)  
  
    # Splitting The Documents For Processing  
    documents = split_text_documents(pdf_doc)  
  
    # Building a Vector DB For The Documents  
    vectordb = Chroma.from_documents(documents, embedding=OpenAIEmbeddings(openai_api_key = openai_api_key))  
  
    # Initializing the ChatGPT Model.  
    llm = ChatOpenAI(temperature=0.3, model_name=model, openai_api_key = openai_api_key)  
  
    # Creating a RetrievalQA model combining the language model and vector database.  
    pdf_qa = RetrievalQA.from_chain_type(  
        llm,  
        retriever=vectordb.as_retriever(search_kwargs={'k': 4}),  
        chain_type="stuff",  
    )  
  
    # Defining the Query.  
    query = """ Write a summary for the document passed to you. You are getting a PDF document and your job \\  
is to interpret the information in the document and to provide a summary. This summary should highlight \\  
key points and identify the main takeaways from the document. If there are important names, dates, events, \\  
facts, etc. please try to keep those in your summary. Do not make up information or facts. The returned \\  
summary should be a few paragraphs.  
    """  
  
    # Executing the RetrievalQA model with the defined query.  
    result = pdf_qa.run(query)  
  
    return result
```

# Loading The PDF

This function uses PdfReader from PyPDF2 to read the PDF file.

It iterates through each page and generates a large string of the text.

That string is then split so that it can be loaded into the vector DB.

```
# Define a function named load_pdf that takes a PDF file path (pdf) as a parameter
def load_pdf(pdf):
    # Create a PdfReader object to read the PDF file
    pdf_reader = PdfReader(pdf)

    # Initialize an empty string to store the extracted text from the PDF
    text = ""

    # Iterate through each page in the PDF
    for page in pdf_reader.pages:
        # Extract the text content from the current page and append it to the 'text' variable
        text += page.extract_text()

    # Use the text_to_doc_splitter function to split the extracted text into a document
    document = text_to_doc_splitter(text)

    # Return the resulting document
    return document
```

# Splitters

The first function is used in the PDF loading step to break apart the text into manageable documents.

The second function is used to split those documents into manageable chunks of text for the vector DB.

```
# Define a function named text_to_doc_splitter that takes a string (text) as a parameter
def text_to_doc_splitter(text: str):
    # Create an instance of the RecursiveCharacterTextSplitter class with specified parameters
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=10000, # The maximum size of each chunk of text
        chunk_overlap=0, # The overlap between consecutive chunks
        length_function=len, # A function to calculate the length of the text
        add_start_index=True, # Whether to add a start index to each chunk
    )

    # Create a document by splitting the input text using the splitter
    document = splitter.create_documents([text])

    # Return the resulting document
    return document

# Define a function named split_text_documents that takes a list of text documents (docs) as a parameter
def split_text_documents(docs: list):
    # Create an instance of the CharacterTextSplitter class with specified parameters
    text_splitter = CharacterTextSplitter(
        chunk_size=1000, # The maximum size of each chunk of text
        chunk_overlap=20 # The overlap between consecutive chunks
    )

    # Split the input list of text documents into smaller chunks using the text splitter
    documents = text_splitter.split_documents(docs)

    # Return the resulting list of split documents
    return documents
```

# Initializing The ChatGPT Model

```
# Initializing the ChatGPT Model.  
llm = ChatOpenAI(temperature=0.3, model_name=model, openai_api_key = openai_api_key)
```

The function ChatOpenAI comes from LangChain to initialize the ChatGPT model.

Here we can set the model that we will be using, we can send along our API key, and we can set the model temperature.

Temperature is a parameter that controls the randomness of predictions:

- higher temperature leads to more varied and unpredictable outputs
- lower temperature results in more conservative and expected responses.

# Setting Up The Vector DB

```
# Building a Vector DB For The Documents
vectordb = Chroma.from_documents(documents, embedding=OpenAIEmbeddings(openai_api_key = openai_api_key))
```

To pass the PDF document to the LLM model, we are going to create a vector database using Chroma, an open source database for embeddings.

Vectors, or embeddings, are numerical arrays representing complex data like text, images, audio, or video.

- A vector database organizes these arrays into a form that allows them to be used in machine learning models.
- In the context of text, vector representations capture semantic and syntactic relationships between words, enhancing the algorithm's ability to comprehend and process language.

For this code, we are using an OpenAI embedding model, but using an embedding model is a crucial part of building a vector DB.

More information can be found here: [Vector DB and Chroma Article](#)

# The QA Model

```
# Creating a RetrievalQA model combining the language model and vector database.
pdf_qa = RetrievalQA.from_chain_type(
    llm,
    retriever=vectordb.as_retriever(search_kwargs={'k': 4}),
    chain_type="stuff",
)
```

Here we are using a Question – Answering Chain from LangChain.

- A QA chain allows for questions to be asked about the documents passed to the model, and those questions can vary widely.
- With this in mind, the QA chain is being used as it has fairly wide applications.

We could have used a Summarizer Chain here instead but a QA chain may be more helpful for wider purposes. More on chain types can be found [here](#).

This chain works to combine the LLM model and the vector DB, as well as the retriever, which allows for us to ask questions about our documents.

For another example of this approach you can head [here](#).

# The Query And Prompt Engineering

```
# Defining the Query.  
query = """ Write a summary for the document passed to you. You are getting a PDF document and your job \ is to interpret the information in the document and to provide a summary. This summary should highlight \ key points and identify the main takeaways from the document. If there are important names, dates, events, \ facts, etc. please try to keep those in your summary. Do not make up information or facts. The returned \ summary should be a few paragraphs.  
      """
```

For this application, we are sending the above prompt.

When building a prompt, prompt engineering techniques should be kept in mind, like:

- Use clear and concise language.
- Be as specific as possible including the format, tone, or style of the response.
- Provide context.
- Provide examples, especially when asking for creative content like stories, poems, or code.
- Use role-playing to frame your prompt such as asking it to respond as a teacher, a scientist, or a historian.
- Avoid open-ended questions.
- Use keywords in your prompt.
- Set constraints like word count, time frame, or complexity.

For more information check out this article [here](#).

# Building The Streamlit App

Here we are setting up the Streamlit Application.

The first block of code just configures the web page settings.

The second block of code is introductory text for our application.

The third block of code is the form where the PDF document can be submitted by the user.

When the submit button is clicked, the PDF will be processed, passed to the model, and a summary will be generated.

```
# Setting Up Some Configuration Settings
st.set_page_config(
    page_title='PDF Summarizer',
    page_icon='-',
    layout='wide',
    initial_sidebar_state='collapsed',
)

# Setting Up The Application Landing Page
st.markdown("""
# PDF Document Summarizer

##### This is a simple web application that uses OpenAI's GPT-3.5 Turbo model to generate a summary of a PDF document.
##### To get started, please enter your OpenAI API Key and upload a PDF file.
##### When ready click the "Submit" button to generate your summary.
""")

def generate_response(doc, model, openai_api_key):
    # Use st.spinner to show a loading spinner while the model is working
    with st.spinner("Generating summary..."):
        output = get_summary(doc, model, openai_api_key)
        st.write(output)

with st.form('my_form'):
    if openai_api_key is None:
        openai_api_key = st.text_input('OpenAI API Key', 'Enter Your OpenAI API Key', type='password')

    files = st.file_uploader("Upload files", type=["pdf"], accept_multiple_files=False)

    submitted = st.form_submit_button('Submit')

    if submitted:
        generate_response(files, model, openai_api_key)
```



# Testing The App

## PDF Document Summarizer

This is a simple web application that uses OpenAI's GPT-3.5 Turbo model to generate a summary of a PDF document.

To get started, please enter your OpenAI API Key and upload a PDF file.

When ready click the "Submit" button to generate your summary.

Upload files



Drag and drop file here

Limit 200MB per file • PDF

Browse files

Submit

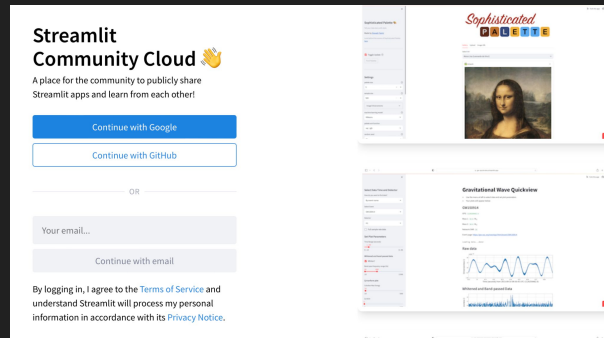
Once the application is built, we can deploy it locally to test its functionality.

If it is working locally, and everything checks out, then it can actually be deployed and hosted for others to use.

# Deploying The Streamlit App

1. Head [here](#) to set up a Streamlit account.
2. Ensure your code is in a GitHub Repo.
3. Select 'New App' and 'Use Existing Repo'
4. Fill out the form.
5. And deploy your app!

For more information on getting started with Streamlit Apps, you can head [here](#).



### Deploy an app

Repository [Paste GitHub URL](#)

Branch

Main file path

App URL (Optional)

[Advanced settings...](#)

# Making Modifications

This application was very simple, but it offers the basic building blocks for you to create a wide number of tools. You can take this and build all sorts of things!

Things to consider though:

- Building embedding layers to incorporate reference info.
- Fine-tuning the model to generate better responses.
- Building conversational applications such as a Chatbot.
- Employing approaches like RAG to have the model retrieve information.
- Using different models including those that are open source.


# Thanks!

Thank you for watching!

Hope you enjoyed!


Now go forth and develop your own  
LLM applications!

And please feel free to follow me on  
LinkedIn!



**I AM  
SPEAKING**  
AT DATAHOUR

▼  
▼  
▼  
▼

 **18 January 2024**

 **7:30 PM - 8:30 PM IST**

**TOPIC**  
Building a Simple LLM Application:  
ChatGPT Summarizer

 **Nielsen**

▼  
▼  
▼  
▼

**RSVP**

Jonathan Schlosser  
Senior Data Scientist