



Home



My Network



Jobs



Messaging



Notifications



Me



For Business

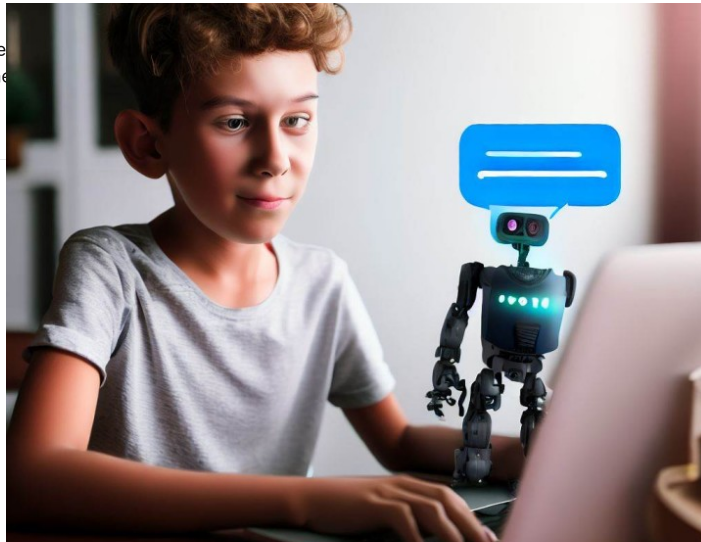


Try

Create your own newsletter

Start your own discussion with a newsletter
build your thought leadership with every ne

[Try it out](#)



DALL-E

How to Use the Streamlit App to Build a Chatbot that Can Respond to Questions from PDF Files?



Indrajit S.

SAP Build LCNC Certified | Proven Record
of Developing Automated Workflows...

1 article

✓ Following

August 9, 2023

[Open Immersive Reader](#)

Hello, everybody! In this article, I'll demonstrate how to use the Streamlit app to build a chatbot that can respond to queries from PDF files using Python, Langchain and the OpenAI API. This is a fun and practical project that can assist you in interactively exploring a variety of subjects and documents.

What You Will Need

To follow along with this tutorial, you will need the following:

- A computer with python 3 installed. You can download python from [here](#).

- A text editor or an IDE of your choice. I recommend using Visual Studio Code, which you can download from [here](#).
- A terminal or a command prompt to run the commands and the app.
- An OpenAI API key, click [here](#) to know more about it. You will need to create an account and generate a secret key. You will also get \$5 of free credit to use the OpenAI API (only if you are Signup with a phone number which is never been used before)
- A .env file, which is a file that stores environment variables. You will need to create this file in the same folder as your python code and store your OpenAI API key in it as follows:

```
OPENAI_API_KEY=sk-  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Replace the xs with your actual secret key. Do not share this file or your key with anyone else.

What You Will Learn

In this tutorial, you will learn how to:

- Install and import the required python libraries for this project.
- Read and process PDF files using PyPDF2.
- Vectorize and store text documents using OpenAIEmbeddings and FAISS from the langchain library.
- Create a chatbot that can answer questions from PDF files using ChatOpenAI and ConversationalRetrievalChain from the langchain library.
- Create a streamlit app with a chat window that allows users to interact with the chatbot and choose different PDF files.

How to Install and Import the Required Python Libraries

The first step is to install and import the required python libraries for this project. We will use the following libraries:

- langchain: A library that allows you to create and manipulate natural language using blockchain technology.
- openai: A library that provides access to powerful artificial intelligence models and tools from OpenAI.
- PyPDF2: A library that allows you to read and write PDF files in python.
- python-dotenv: A library that allows you to load environment variables from a .env file.
- streamlit: A library that allows you to create interactive web applications for data science and machine learning.
- faiss-cpu: A library that provides efficient similarity search and clustering of dense vectors.
- streamlit-extras: A library that provides some extra components and utilities for streamlit applications.

To install these libraries, you can use the pip command in your terminal or command prompt as follows:

```
pip install langchain openai PyPDF2==3.0.1  
python-dotenv==1.0.0 streamlit==1.24.0 faiss-  
cpu==1.7.4 streamlit-extras
```

This will install all the libraries and their dependencies in your default python environment.

You can also create a requirements txt file and store all the packages name and their version as per your need and save this in your project directory (like below) and use the pip command in your terminal or command prompt as follows:

```
langchain  
PyPDF2==3.0.1  
python-dotenv==1.0.0  
streamlit==1.24.0  
faiss-cpu==1.7.4  
streamlit-extras  
openai
```

PIP command to install packages from requirements file

```
pip install -r requirements.txt
```

Alternatively, you can create a virtual environment for this project using the venv module in python as follows (replace **env** with the name you want to name your environment):

```
python -m venv env
```

This will create a folder called env in your current directory, which will contain a copy of python and pip.

To activate the virtual environment, you can use the following command depending on your operating system:

- On Windows, run:

```
env\Scripts\activate
```

- On Linux or macOS, run:

```
source env/bin/activate
```

You should see (env) at the beginning of your terminal or command prompt line, indicating that you are in the virtual environment.

To install the libraries in the virtual environment, you can use the same pip command as before.

To deactivate the virtual environment, you can use the following command:

```
deactivate
```

You should see (env) disappear from your terminal or command prompt line, indicating that you are out of the virtual environment.

To import these libraries in your python code, you can use the following statements:

```
import streamlit as st
from PyPDF2 import PdfReader
from dotenv import load_dotenv
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains import ConversationalRetrievalChain
from langchain.chat_models import ChatOpenAI
from langchain.vectorstores import FAISS
import os
import pickle
```

How to Read and Process PDF Files Using PyPDF2

The next step is to read and process PDF files using PyPDF2. We will use some sample PDF files that contain different topics, such as health insurance benefits, tax regime, reinforcement learning, and GPT-4 training. You can use any pdf file and save them in a folder called **PDFs** in your current directory. You can download the tax regime PDF [here](#), you can download the health insurance PDF [here](#).

To read the content of a PDF file, we will define a function called **read_pdf** that takes a file path as an argument and returns a string of text extracted from the PDF file. We will use the PdfReader class from PyPDF2 to open the file and iterate over its pages. We will use the **extract_text** method from each page to get the text and append it to a variable called text. We will return the text variable at the end of the function.

The code for the **read_pdf** function is as follows:

```
# Function to read PDF content
def read_pdf(file_path):
    pdf_reader = PdfReader(file_path)
    text = ""
    for page in pdf_reader.pages:
        text += page.extract_text()
    return text
```

To map the PDF files to their names, we will create a dictionary called **pdf_mapping** that contains the names as keys and the file paths as values. For example, we will map the name '**Health Insurance Benefits**' to the file path '**pdfs/TaxBenefits_HealthInsurance.pdf**'. You can add more mappings as needed.

The code for the **pdf_mapping** dictionary is as follows:

```
# Mapping of PDFs
pdf_mapping = {
    'Health Insurance Benefits':
        'pdfs/TaxBenefits_HealthInsurance.pdf',
    'Tax Regime': 'pdfs/New-vs-Old-Tax.pdf',
    'Reinforcement Learning':
        'pdfs/SuttonBartoIPRLBook2ndEd.pdf',
    'GPT-4 All Training':
        'pdfs/2023_GPT4All_Technical_Report.pdf',
    # Add more mappings as needed
}
```

How to Vectorize and Store Text Documents Using OpenAIEmbeddings and FAISS

The next step is to vectorize and store text documents using OpenAIEmbeddings and FAISS from the langchain library. We will use these libraries to create numerical representations of the text documents and store them in a **vectorstore** that can be used for similarity search and retrieval.

To vectorize the text documents, we will use the OpenAIEmbeddings class from the langchain library. This class allows us to use the OpenAI API to generate embeddings for any text using different AI models, such as GPT-3 or GPT-4. We will need to load our environment variables from the **.env** file using the **load_dotenv** function from the **python-dotenv** library. This will allow us to access our **OpenAI API key** using the **os.environ** dictionary.

The code for loading the environment variables and creating an instance of OpenAIEmbeddings is as follows:

```
# Load environment variables
load_dotenv()
```

```
# Create an instance of OpenAIEmbeddings
embeddings = OpenAIEmbeddings()
```

To store the vectorized documents, we will use the FAISS class from the langchain library. This class allows us to create a vectorstore using FAISS, which is a library that provides efficient similarity search and clustering of dense vectors. We will use the **from_texts** class method to create a **vectorstore** from a list of texts using an embedding instance. This method will return an instance of FAISS that contains the vectorized texts and their ids.

The code for creating an instance of FAISS from a list of texts is as follows:

```
# Create an instance of FAISS from a list of
texts
vectorstore = FAISS.from_texts(texts,
embedding=embeddings)
```

However, before we can create a list of texts from our PDF files, we need to split them into smaller chunks that can be processed by the OpenAI API. To do this, we will use the **RecursiveCharacterTextSplitter** class from the **langchain library**. This class allows us to split a long text into smaller chunks based on a chunk size, a chunk overlap, and a length function. We will use a **chunk size of 1000 characters**, a chunk **overlap of 150 characters**, and the len function as parameters.

The code for creating an instance of **RecursiveCharacterTextSplitter** is as follows:

```
# Create an instance of
RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=150,
    length_function=len
)
```

Now, we can use this instance to split any text into smaller chunks using the **split_text** method. This method takes a text argument and returns a list of chunks. We can use this method on each PDF file that we have read using

our **read_pdf** function. We can then append all the chunks into one list called **documents**.

The code for splitting each PDF file into chunks and creating a list of documents is as follows:

```
# Process the PDF text and create the documents
list
documents = []
for file_path in pdf_mapping.values():
    text = read_pdf(file_path)
    chunks = text_splitter.split_text(text=text)
    documents.extend(chunks)
```

Now, we can use our **documents list** to create our **vectorstore** using our embeddings instance as follows:

```
# Vectorize the documents and create vectorstore
vectorstore = FAISS.from_texts(documents,
                                embedding=embeddings)
```

We can also save our **vectorstore** using **pickle**, which is a module that allows us to serialize and deserialize python objects. This will allow us to reuse our **vectorstore** without having to recreate it every time we run our app. We will create a folder called **pickle** in our current directory and save our **vectorstore** as a pickle file with the same name as the PDF file. We will use the **open** function to open a file in write binary mode and the **dump** function from **pickle** to write our vectorstore object to the file.

The code for saving our vectorstore using pickle is as follows:

```
# Save vectorstore using pickle
pickle_folder = "pickle"
if not os.path.exists(pickle_folder):
    os.mkdir(pickle_folder)

for name, file_path in pdf_mapping.items():
    pickle_file_path =
os.path.join(pickle_folder, f"{name}.pkl")
    if not os.path.exists(pickle_file_path):
        with open(pickle_file_path, "wb") as f:
            pickle.dump(vectorstore, f)
```


How to Create a Chatbot that Can Answer Questions from PDF Files Using ChatOpenAI and ConversationalRetrievalChain

The next step is to create a chatbot that can answer questions from PDF files using **ChatOpenAI** and **ConversationalRetrievalChain** from the langchain library. We will use these classes to create a chatbot that can generate natural language responses based on the OpenAI API and retrieve relevant documents from our vectorstore.

To create a chatbot that can generate natural language responses, we will use the ChatOpenAI class from the langchain library. This class allows us to use the OpenAI API to create a chatbot that can converse with users using different AI models, such as GPT-3 or GPT-4. We will need to provide some parameters for this class, such as **temperature**, **max_tokens**, and **model_name**. Temperature controls the randomness of the generated text, max_tokens controls the maximum number of tokens in the generated text, and model_name controls the name of the AI model to use. We will use a temperature of 0, which means no randomness, a max_tokens of 1000, which means up to 1000 tokens in the generated text, and a model_name of gpt-3.5-turbo, which is a custom model that combines GPT-3 and GPT-4.

The code for creating an **instance of ChatOpenAI** is as follows:

```
# Create an instance of ChatOpenAI
llm = ChatOpenAI(temperature=0, max_tokens=1000,
model_name="gpt-3.5-turbo")
```

To create a chatbot that can retrieve relevant documents from our vectorstore, we will use the **ConversationalRetrievalChain** class from the langchain library. This class allows us to create a chatbot that can answer questions from a retriever object using a language model object. We will need to provide two arguments for this class: **llm** and **retriever**. Llm is an instance of ChatOpenAI

that we have created before, and retriever is an instance of FAISS that we have created before.

The code for creating an instance of **ConversationalRetrievalChain** is as follows:

```
# Create an instance of
ConversationalRetrievalChain
qa = ConversationalRetrievalChain.from_llm(llm,
retriever=vectorstore.as_retriever())
```

Now, we have created our chatbot that can answer questions from PDF files using natural language processing and vector similarity. We can use this chatbot by calling its call method with a dictionary argument that contains two keys: **question** and **chat_history**. Question is a string that contains the user's question, and chat_history is a list of tuples that contains the previous messages between the user and the chatbot. Each tuple contains two elements: role and content. Role is either "user" or "assistant", and content is a string that contains the message. The call method will return a dictionary that contains one key: answer. Answer is a string that contains the chatbot's response.

The code for using our chatbot with an example question and chat history is as follows:

P.S - this response is coming from the document I had train/feed to the code, in your case you may get different response based on which document you train/feed

```
# Use our chatbot with an example question and
chat history
question = "What are the benefits of health
insurance?"
chat_history = [
    ("user", "Hi, I want to know more about
health insurance."),
    ("assistant", "Hello, welcome to the PDF
Chat App. I can answer your questions from
different PDF files. Please choose a PDF file
from the sidebar.")
]
result = qa({"question": question,
"chat_history": chat_history})
answer = result["answer"]
print(answer)
```

The output of this code is as follows:

The benefits of health insurance include:

1. **Financial Protection:** Health insurance helps protect you from high medical expenses by covering a significant portion of your healthcare costs. It can help you avoid financial strain and ensure that you receive the necessary medical treatment without worrying about the cost.

2. **Access to Quality Healthcare:** With health insurance, you have access to a network of healthcare providers and hospitals. This allows you to receive timely and quality medical care when needed.

3. **Preventive Care:** Many health insurance plans cover preventive services such as vaccinations, screenings, and annual check-ups. These services help detect and prevent potential health issues before they become more serious and costly to treat.

4. **Coverage for Hospitalization:** Health insurance provides coverage for hospitalization expenses, including room charges, doctor's fees, surgical procedures, and medication. This can significantly reduce the financial burden of a hospital stay.

5. **Coverage for Medications:** Health insurance often includes coverage for prescription medications, making them more affordable and accessible.

6. **Mental Health Support:** Many health insurance plans now include coverage for mental health services, including therapy and counseling. This ensures that individuals can receive the necessary support for their mental well-being.

7. **Additional Benefits:** Some health insurance plans offer additional benefits such as maternity coverage, dental and vision care, alternative therapies, and wellness programs.

It's important to note that the specific benefits and coverage may vary depending on the insurance plan and provider. It's advisable to carefully review the terms and conditions of your health insurance policy to understand the exact benefits you are entitled to.

How to Create a Streamlit App with a Chat Window that Allows Users to Interact with the Chatbot and Choose Different PDF Files

The final step is to create a **streamlit app** with a chat window that allows users to interact with the chatbot and choose different PDF files. We will use the streamlit library to create a web app that displays a title, a sidebar, and a chat window. We will use the **streamlit-extras** library to create a chat window that supports markdown rendering and chat input.

To create a streamlit app, we will need to define a main function that contains all the streamlit code. We will also need to check if the name variable is equal to "main" and call the main function if it is. This will ensure that our app runs only when we execute our python file directly, and not when we import it as a module.

The code for defining the main function and checking the name variable is as follows:

```
# Main Streamlit app
def main():
    # Streamlit code goes here

if __name__ == "__main__":
    main()
```

To display a title for our app, we will use the `st.title` function from streamlit. This function takes a string argument and displays it as a large heading on the app. We will use the string "Query your PDF" as our title.

The code for displaying a title for our app is as follows:

```
# Display a title for our app
st.title("Query your PDF")
```

To display a sidebar for our app, we will use the `st.sidebar` container from streamlit. This container allows us to create a sidebar on the left side of the app that can contain different widgets and elements. We will use this container to display some information about our app, such as its name, description, and purpose. We will also use this container to display a selectbox widget that allows users to choose different PDF files from our `pdf_mapping` dictionary. We will use the `st.title`, `st.markdown`, and `st.selectbox` functions from streamlit to create these elements.

The code for displaying a sidebar for our app is as follows:

```
# Display a sidebar for our app
with st.sidebar:
    # Display the name of our app
    st.title('🗨️ PDF Chat App')
    # Display some information about our app
    st.markdown('''
    ## About
    This app allows you to query different PDF
    files using natural language and get answers
    from a chatbot powered by OpenAI API.
    ''')
    # Display a selectbox widget that allows
    users to choose different PDF files
    custom_names = list(pdf_mapping.keys())
    selected_custom_name = st.selectbox('Choose
    your PDF', ['', *custom_names])
```

To display a chat window for our app, we will use the `st.chat_message` and `st.chat_input` functions from `streamlit-extras`. These functions allow us to create a chat window that supports markdown rendering and chat input. We will use these functions to display the previous messages between the user and the chatbot, and to allow the user to type their questions. We will also use these functions to display the chatbot's responses based on our `qa` instance and our `vectorstore` instance. We will use the `st.session_state` object from `streamlit` to store and access the messages and the processed data across different runs of our app.

The code for displaying a chat window for our app is as follows:

```
# Display a chat window for our app
# Initialize Streamlit chat UI
if "messages" not in st.session_state:
    st.session_state.messages = []

for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

if prompt := st.chat_input("Ask your questions
from PDF "f'{selected_custom_name}'"?"):
    st.session_state.messages.append({"role":
    "user", "content": prompt})
    with st.chat_message("user"):
        st.markdown(prompt)

# Load vectorstore using pickle
pickle_folder = "pickle"
```

```

pickle_file_path =
os.path.join(pickle_folder, f"
{selected_custom_name}.pkl")
if os.path.exists(pickle_file_path):
    with open(pickle_file_path, "rb") as f:
        vectorstore = pickle.load(f)
        retriever = vectorstore.as_retriever()
        qa.retriever = retriever

result = qa({"question": prompt,
"chat_history": [(message["role"],
message["content"]) for message in
st.session_state.messages]})

with st.chat_message("assistant"):
    message_placeholder = st.empty()
    full_response = result["answer"]

message_placeholder.markdown(full_response +
"|")
message_placeholder.markdown(full_response)

st.session_state.messages.append({"role":
"assistant", "content": full_response})

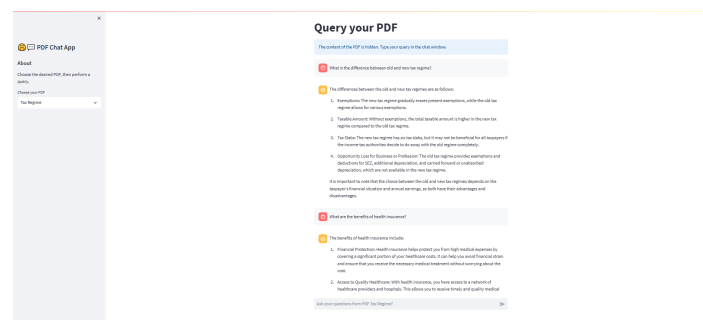
```

How to Run and Test the Streamlit App

To run and test the streamlit app, we will need to use the streamlit run command in our terminal or command prompt. This command will launch a local web server and open a browser window that displays our app. We will need to provide the name of our python file as an argument for this command. For example, if our python file is called app.py, we will use the following command:

```
streamlit run app.py
```

This will open a browser window that looks something like this:



Screenshot from my app

We can test our app by choosing different PDF files from the sidebar and typing different questions in the chat window.

We can see how the chatbot responds to our questions and provides relevant information from the PDF files. We can also see how the chatbot uses natural language processing and vector similarity to generate and retrieve the answers.

Here are some examples of questions and answers that we can try with our app:

Question:

What are the benefits of health insurance?

- **Answer:** The benefits of health insurance include:

1. **Financial Protection:** Health insurance helps protect you from high medical expenses by covering a significant portion of your healthcare costs. It can help you avoid financial strain and ensure that you receive the necessary medical treatment without worrying about the cost.
2. **Access to Quality Healthcare:** With health insurance, you have access to a network of healthcare providers and hospitals. This allows you to receive timely and quality medical care when needed.
3. **Preventive Care:** Many health insurance plans cover preventive services such as vaccinations, screenings, and annual check-ups. These services help detect and prevent potential health issues before they become more serious and costly to treat.
4. **Coverage for Hospitalization:** Health insurance provides coverage for hospitalization expenses, including room charges, doctor's fees, surgical procedures, and medication. This can significantly reduce the financial burden of a hospital stay.
5. **Coverage for Medications:** Health insurance often includes coverage for prescription medications, making them more affordable and accessible.
6. **Mental Health Support:** Many health insurance plans now include coverage for mental health services, including therapy and counseling. This ensures that individuals can receive the necessary support for their mental well-being.
7. **Additional Benefits:** Some health insurance plans offer additional benefits such as maternity coverage, dental and

vision care, alternative therapies, and wellness programs.

- It's important to note that the specific benefits and coverage may vary depending on the insurance plan and provider. It's advisable to carefully review the terms and conditions of your health insurance policy to understand the exact benefits you are entitled to.

Question: What is the difference between old and new tax regime?

Answer: The differences between the old and new tax regimes are as follows:

1 Exemptions: The new tax regime gradually erases present exemptions, while the old tax regime allows for various exemptions.

2 Taxable Amount: Without exemptions, the total taxable amount is higher in the new tax regime compared to the old tax regime.

3 Tax Slabs: The new tax regime has six tax slabs, but it may not be beneficial for all taxpayers if the income-tax authorities decide to do away with the old regime completely.

4 Opportunity Loss for Business or Profession: The old tax regime provides exemptions and deductions for SEZ, additional depreciation, and carried forward or unabsorbed depreciation, which are not available in the new tax regime.

- It is important to note that the choice between the old and new tax regimes depends on the taxpayer's financial situation and annual earnings, as both have their advantages and disadvantages.

Here is the full code - Feel free to use and make changes as you fit for your practice.

```
import os
import streamlit as st
from PyPDF2 import PdfReader
from dotenv import load_dotenv
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
import pickle
from langchain.chains import
```



```

ConversationalRetrievalChain
from langchain.chat_models import ChatOpenAI
from langchain.vectorstores import FAISS

# Function to read PDF content
def read_pdf(file_path):
    pdf_reader = PdfReader(file_path)
    text = ""
    for page in pdf_reader.pages:
        text += page.extract_text()
    return text

# Mapping of PDFs
pdf_mapping = {
    'HealthInsurance Benefits':
    'TaxBenefits_HealthInsurance.pdf',
    'Tax Regime': 'New-vs-Old-Tax.pdf',
    'Reinforcement Learning':
    'SuttonBartoIPRLBook2ndEd.pdf',
    'GPT4 All Training':
    '2023_GPT4All_Technical_Report.pdf',
    # Add more mappings as needed
}

# Load environment variables
load_dotenv()

# Main Streamlit app
def main():
    st.title("Query your PDF")
    with st.sidebar:
        st.title('🗨️ PDF Chat App')
        st.markdown('')
        ## About
        Choose the desired PDF, then perform a
        query.
    '''

    custom_names = list(pdf_mapping.keys())

    selected_custom_name =
    st.sidebar.selectbox('Choose your PDF', ['',
    *custom_names])

    selected_actual_name =
    pdf_mapping.get(selected_custom_name)

    if selected_actual_name:
        pdf_folder = "pdfs"
        file_path = os.path.join(pdf_folder,
        selected_actual_name)

        try:
            text = read_pdf(file_path)
            st.info("The content of the PDF is
            hidden. Type your query in the chat window.")
        except FileNotFoundError:
            st.error(f"File not found:
            {file_path}")
        return

```

```

except Exception as e:
    st.error(f"Error occurred while
reading the PDF: {e}")
    return

text_splitter =
RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=150,
    length_function=len
)

# Process the PDF text and create the
documents list
documents =
text_splitter.split_text(text=text)

# Vectorize the documents and create
vectorstore
embeddings = OpenAIEmbeddings()
vectorstore =
FAISS.from_texts(documents,
embedding=embeddings)

st.session_state.processed_data = {
    "document_chunks": documents,
    "vectorstore": vectorstore,
}

# Save vectorstore using pickle
pickle_folder = "Pickle"
if not os.path.exists(pickle_folder):
    os.mkdir(pickle_folder)

pickle_file_path =
os.path.join(pickle_folder, f"
{selected_custom_name}.pkl")

if not os.path.exists(pickle_file_path):
    with open(pickle_file_path, "wb") as
f:
        pickle.dump(vectorstore, f)

# Load the Langchain chatbot
llm = ChatOpenAI(temperature=0,
max_tokens=1000, model_name="gpt-3.5-turbo")
qa =
ConversationalRetrievalChain.from_llm(llm,
vectorstore.as_retriever())

# Initialize Streamlit chat UI
if "messages" not in st.session_state:
    st.session_state.messages = []

for message in
st.session_state.messages:
    with
st.chat_message(message["role"]):
        st.markdown(message["content"])

if prompt := st.chat_input("Ask your
questions from PDF
"f'{selected_custom_name}'"?"):
    st.session_state.messages.append({"role":

```

```

"user", "content": prompt})
    with st.chat_message("user"):
        st.markdown(prompt)

    result = qa({"question": prompt,
"chat_history": [(message["role"],
message["content"]) for message in
st.session_state.messages]})
    print(prompt)

    with st.chat_message("assistant"):
        message_placeholder = st.empty()
        full_response = result["answer"]

message_placeholder.markdown(full_response +
"|")

message_placeholder.markdown(full_response)
print(full_response)

st.session_state.messages.append({"role":
"assistant", "content": full_response})

if __name__ == "__main__":
    main()

```

Conclusion

In this article, we have learned:

- How to create a streamlit app with a chatbot that can answer questions from PDF files using python and the OpenAI API.
- Install and import the required python libraries, such as langchain, openai, PyPDF2, python-dotenv, streamlit, faiss-cpu, and streamlit-extras.
- Read and process PDF files using PyPDF2 and split them into smaller chunks using RecursiveCharacterTextSplitter from the langchain library.
- Vectorize and store text documents using OpenAIEmbeddings and FAISS from the langchain library and save them using pickle.
- Create a chatbot that can generate natural language responses using ChatOpenAI from the langchain library and retrieve relevant documents using ConversationalRetrievalChain from the langchain library.
- Create a streamlit app with a title, a sidebar, and a chat window using streamlit and streamlit-extras.
- Run and test the streamlit app using the streamlit run command and interact with the chatbot and choose


different PDF files.

I hope you enjoyed this article and found it useful. If you have any questions or feedback, please feel free to leave a connect. Thank you for reading! 😊

#Python #Langchain #OpenAI #PDF #streamlit
#DataScience, #MachineLearning, #NLP #AI

Report this

Published by



Indrajit S.
SAP Build LCNC Certified | Proven Record of Developing Automated Work...
Published • 6mo

1 article









✓ Following


Hello, everyone! I am excited to share with you my latest attempt on creating: a streamlit app with a chatbot that can answer questions from PDF files using python and the OpenAI API. This is a fun and useful project that can help you explore different topics and documents in an interactive way. In this article, I will show you how I created this app step by step, and how you can try it yourself. I hope you enjoy reading it and find it useful. Please feel free to leave your comments and feedback below. Thank you for your support! 😊

#Streamlit #OpenAI #PDF #Python #Langchain #AI #DocumentChatbot

Like Comment Share 16 1 comment

Reactions






+4

1 Comment

Most relevant ▾



Enrica D'souza • 3rd+
Marketing Associate at Engati
6mo ...

Explore a variety of chatbot applications in our concise guide. Click for insights- <https://bit.ly/3Yr7sqN>

Like 2 Reply



Indrajit S.
SAP Build LCNC Certified | Proven Record of Developing Automated Workflows using SAP Build Process Automation (formerly SAP iRPA) v2 & v3 | JS | UiPath | VB.NET | GenAI

✓ Following

