Introduction

Using the APIs from [OpenAI](https://openai.com/) and the [langchain](https://github.com/hwchase17/langchain) project, it is quite easy to implement a bot that is fed with your documentation and other product information. The answers of the bot are then specific to the trained knowledge domain. Background on the techniques used can be found [here](https://www.mlq.ai/fine-tuning-gpt-3-question-answer-bot/); this article describes the required Python code. In a nutshell:

To create a question-answering bot at a high level, we need to:

* Prepare and upload a training dataset
* Find the most similar document embeddings to the question embedding
* Add the most relevant document sections to the query prompt
* Answer the user's question based on additional context

The first step should be done only once (or once with every doc update), as calculating the embeddings has a price tag attached. The language of choice for such projects is Python, with a rich field of 3rd party tools. langchain is especially useful as it offers almost everything the DocBot heart desires.

Embedding documents

Langchain offers different loaders, e.g., for PDF, TXT, or sitemaps. The web pages are then automatically scraped and de-HTMLized. This is what the embedding code looks like for our use case at [combit](https://www.combit.com/).

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.vectorstores import Chroma

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain.llms import OpenAI

from langchain.chains import RetrievalQA

from langchain.chat\_models import ChatOpenAI

from langchain.document\_loaders.sitemap import SitemapLoader

from langchain.document\_loaders import UnstructuredPDFLoader

from langchain.document\_loaders import TextLoader

# This adds documents from a langchain loader to the database. The customized splitters serve to be able to break at sentence level if required.

def add\_documents(loader, instance):

documents = loader.load()

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=100, separators= ["\n\n", "\n", ".", ";", ",", " ", ""])

texts = text\_splitter.split\_documents(documents)

instance.add\_documents(texts)

# Create embeddings instance

embeddings = OpenAIEmbeddings(openai\_api\_key="...")

# Create Chroma instance

instance = Chroma(embedding\_function=embeddings, persist\_directory="C:\\DocBot")

# add Knowledgebase Dump (CSV file)

loader = TextLoader("[C:\\DocBot\\Input\\en-kb@forum](mailto:C:\\DocBot\\Input\\en-kb@forum)-combit-net-2023-04-25.dcqresult.csv")

add\_documents(loader, instance)

# add EN sitemap

loader = SitemapLoader(web\_path='<https://www.combit.com/page-sitemap.xml>')

add\_documents(loader, instance)

# add EN Blog sitemap, only use English blog posts

loader = SitemapLoader(web\_path='<https://www.combit.blog/XMLSitemap.xml>', filter\_urls=["<https://www.combit.blog/en/>"])

add\_documents(loader, instance)

# add documentation PDFs

pdf\_files = ["C:\\DocBot\\Input\\Ad-hoc Designer-Manual.pdf",

"C:\\DocBot\\Input\\Designer-Manual.pdf",

"C:\\DocBot\\Input\\Programmers-Manual.pdf",

"C:\\DocBot\\Input\\ServicePack.pdf",

"C:\\DocBot\\Input\\ReportServer.pdf"]

for file\_name in pdf\_files:

loader = UnstructuredPDFLoader(file\_name)

add\_documents(loader, instance)

instance.persist()

instance = None

Python

Copy

The costs are within tolerable limits - you'd rarely exceed a few Dollars, even for extensive documentation. Each query also costs another mini-amount, using two calls (embedding the question and then completing the answer, see below).

Q&A query with GPT 3.5 turbo

For this purpose, Flask can provide a WebAPI that works via langchain with the persistent vector database created in the first step. For the actual completion, gpt-3.5-turbo is used so the bot is personalized with a custom prompt template. This enables us to set the tone for the answers and make sure the model is not hallucinating answers. To make CORS calls work (for testing on a local web page), an access-control-allow-origin header is added here.

from flask import Flask, request,make\_response

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.vectorstores import Chroma

from langchain.text\_splitter import CharacterTextSplitter

from langchain.llms import OpenAI

from langchain.chains import RetrievalQA

from langchain.chat\_models import ChatOpenAI

from langchain.prompts import PromptTemplate

app = Flask(\_\_name\_\_)

embeddings = OpenAIEmbeddings(openai\_api\_key="...")

instance = Chroma(persist\_directory="C:\\DocBot", embedding\_function=embeddings)

tech\_template = """As a combit support bot, your goal is to provide accurate

and helpful technical information about List & Label, a powerful reporting tool used for

building various applications. You should answer user inquiries based on the

context provided and avoid making up answers. If you don't know the answer,

simply state that you don't know. Provide concrete examples like code snippets

or function prototypes wherever possible. Remember to provide relevant information

about List & Label's features, benefits, and API to assist the user in

understanding how to best use it for application development.

{context}

Q: {question}

A: """

PROMPT = PromptTemplate(

template=tech\_template, input\_variables=["context", "question"]

)

qa = RetrievalQA.from\_chain\_type(llm=ChatOpenAI(model\_name="gpt-3.5-turbo", temperature=0,

openai\_api\_key="..."),

chain\_type="stuff",

retriever=instance.as\_retriever(),

chain\_type\_kwargs={"prompt": PROMPT})

@app.route('/api')

def my\_api():

query = request.args.get('query')

# process the input string here

output\_string = qa.run(query)

response = make\_response(output\_string, 200)

response.mimetype = "text/plain"

response.headers.add('Access-Control-Allow-Origin', '\*')

return response

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

Python

Copy

Usage in a Web Page

The API can then be used, e.g., via fetch from JavaScript. Like this, here with localhost as a source if you're testing the bot locally. The HTML code was written by ChatGPT, of course :).

<html>

<head>

<title>combit DocBot</title>

<link rel="stylesheet" href="<https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css>">

<style>

#spinner {

display: none;

}

#result {

margin-top: 20px;

}

</style>

</head>

<body>

<div class="container my-5">

<div class="row justify-content-center">

<div class="col-md-6">

<div class="card">

<div class="card-header bg-primary text-white">

<h4 class="mb-0">combit DocBot</h4>

</div>

<div class="card-body">

<form>

<div class="mb-3">

<label for="prompt" class="form-label">Enter a prompt:</label>

<input type="text" id="prompt" name="prompt" class="form-control">

</div>

<div class="d-grid gap-2">

<button type="submit" class="btn btn-primary">

Query

<span class="spinner-border spinner-border-sm ms-2" id="spinner" role="status" aria-hidden="true"></span>

</button>

</div>

</form>

</div>

<div class="card-footer">

<div class="result" id="result"></div>

</div>

</div>

</div>

</div>

</div>

<script src="<https://cdn.jsdelivr.net/npm/marked/marked.min.js>"></script>

<script>

const form = document.querySelector('form');

const resultDiv = document.querySelector('#result');

spinner.style.display = 'none';

form.addEventListener('submit', async (event) => {

event.preventDefault();

spinner.style.display = 'inline-block';

resultDiv.textContent = '';

const prompt = document.querySelector('#prompt').value;

fetch(`<http://127.0.0.1:5000/api?query=>${encodeURIComponent(prompt)}`)

.then((response) => response.text())

.then((text) => {

spinner.style.display = 'none'; resultDiv.innerHTML = marked.parse(text);});

});

</script>

</body>

</html>