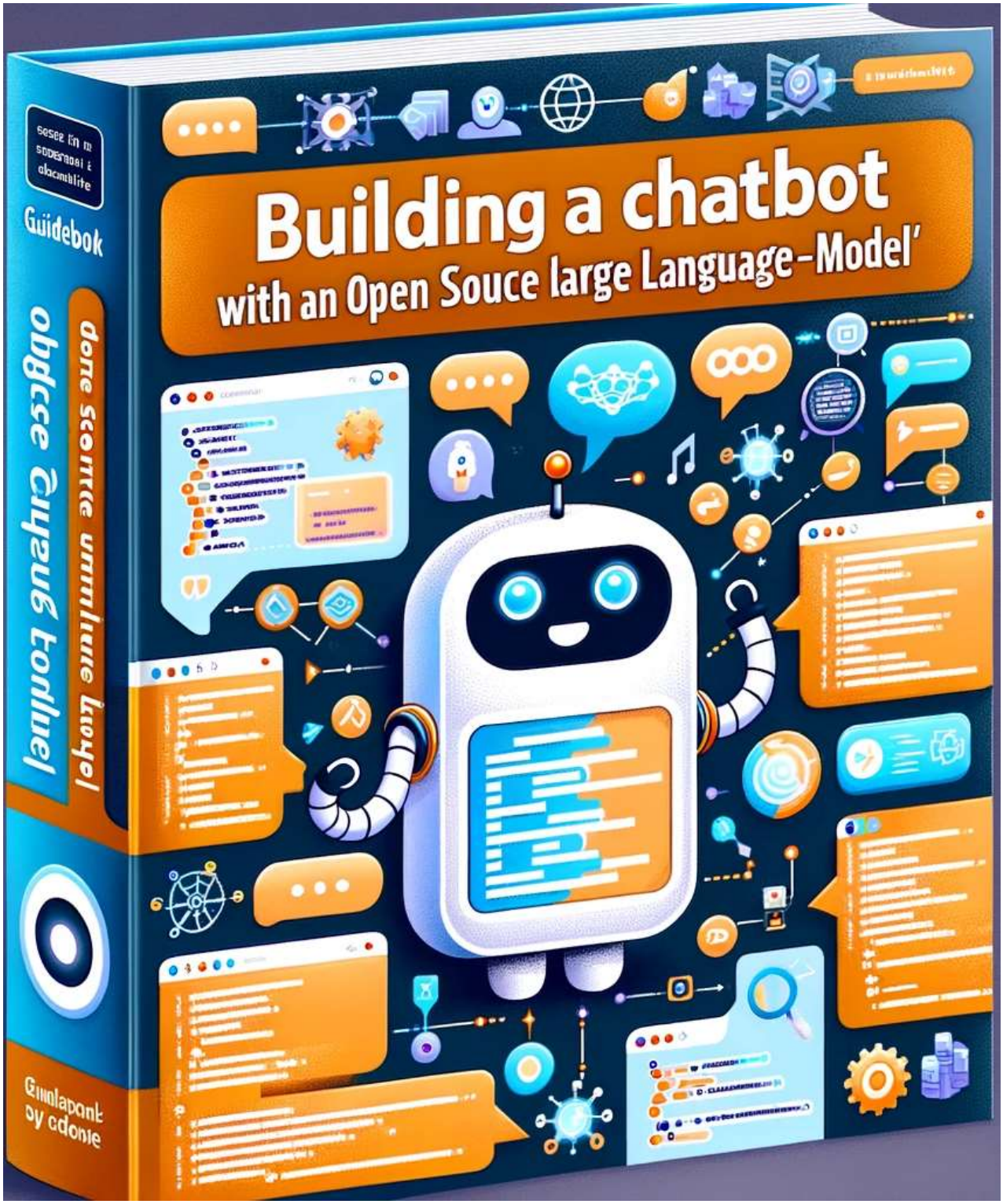

Build your chatbot using an open source LLM

GPT4All & Llama

Compiled by: Mohamed Ashour

Inspired from: Abid Saudagar

December 2023

About the author

The author, with a robust background spanning nearly seven years in the construction industry, brings a unique blend of expertise and academic achievement to the table. Holding a bachelor's degree in construction and engineering management, he has laid a solid foundation in the technical aspects of the industry. Further elevating his qualifications, the author pursued and attained a Master of Science degree in Commercial Management and Quantity Surveying, a discipline that marries the technicalities of construction with the nuances of business management.

Complementing their construction-centric education, the author also delved into the realm of data analytics, acquiring a degree that marks a significant pivot in their career trajectory. This educational journey is crowned by their chartered status from two prestigious institutions: the Royal Institution of Chartered Surveyor and the British Computer Society, reflecting a rare confluence of construction expertise and computational acumen.

In recent years, the author has shifted their focus towards the data world, dedicating the past three years to working intensively in this domain. Their interest particularly lies in the deployment of Artificial Intelligence (AI) and Machine Learning (ML) within the construction industry, a sector ripe for digital transformation. Recognizing the potential of AI and ML to revolutionize traditional practices, the author has been at the forefront of integrating these technologies into construction processes, aiming to enhance efficiency, accuracy, and overall project management.

One of the author's notable contributions is the development of a series of chatbots using open-source large language models. These chatbots represent a significant innovation, leveraging the power of AI to streamline communication, automate routine tasks, and provide intelligent assistance in various construction-related scenarios. The author's work in this area not only showcases their technical prowess but also their commitment to driving the construction industry forward through the adoption of cutting-edge technologies. Their unique blend of construction knowledge, data analytics expertise, and passion for AI and ML positions them as a visionary figure, poised to make a lasting impact on the industry.

You can reach out to me on my LinkedIn page: https://www.linkedin.com/in/mohamedashour-0727/ or via email on mohamed_ashour@apcmasterypath.co.uk.

Disclaimer

You can contact the author of this document via email on mohamed_ashour@apcmasterypath.co.uk & mo_ashour1@outlook.com.

Table of Contents

Problem statement

In the construction industry there are a lot of documents which contain useful unstructured data. This data could be changed into structured data that could be further embedded into a SQL database or any other type of databases.

In the current climate of having open-source large language model such as Mistral, Llama, Falcon, and many others, it is becoming much easier to convert non-structured data into structured ones.

Querying/Chatting with text documents is getting easier using data loaders available under different python packages. A multitude of formats are supported under these data loaders including txt files, word documents, PDF files and power point slides. This file will showcase how to make use of data loaders to upload PDF documents that will be queried using Llama 13B and GPT4All models.

Building chatbot Process

The whole process of prompting, processing and answering is called inference. The end user sends a specific instruction or a set of instructions to a large language model and the large language model is expected to provide and end result.

There is a multitude of chatbots already available out there in the market. Building a chatbot takes into account the following components:

- Queries: These are the prompts provided by the end user
- Input documents: the input documents can take a wide variety of shapes including powerpoint slides, word documents, pdfs, html site or even merely a text query.
- Embedding model: A wide variety of free embedding models are available on the Huggingface website. Some embedding models are available for commercial use under APACHE 2.0 license. The embedding process allows for conversion of the text chunks into numeric values. Models do not understand text face value. Text chunks have to be created first and then converted into numeric values for the model to process it.
- Vector Storage: The purpose of this step in the process is to store the numeric values of the text chunks into vectors.
- Database: Different databases are available such as Pinecone and Chroma. The databases are used to store the vectors created by the embedding process.
- Large Language Model: these models contain millions or billions of parameters and they are trained on gigabytes or terabytes of data. These models receive an input. The input is then embedded and encoded. The vector representation of the embeddings is then decoded into torch vectors or tensors using pytorch or tensor flow libraries. The final result is then provided in a text format using softmax library.
- Answers: That is the end product required by the user provided after using the LLM model.

The following diagram summarizes the processing undertaken within a typical chatbot.

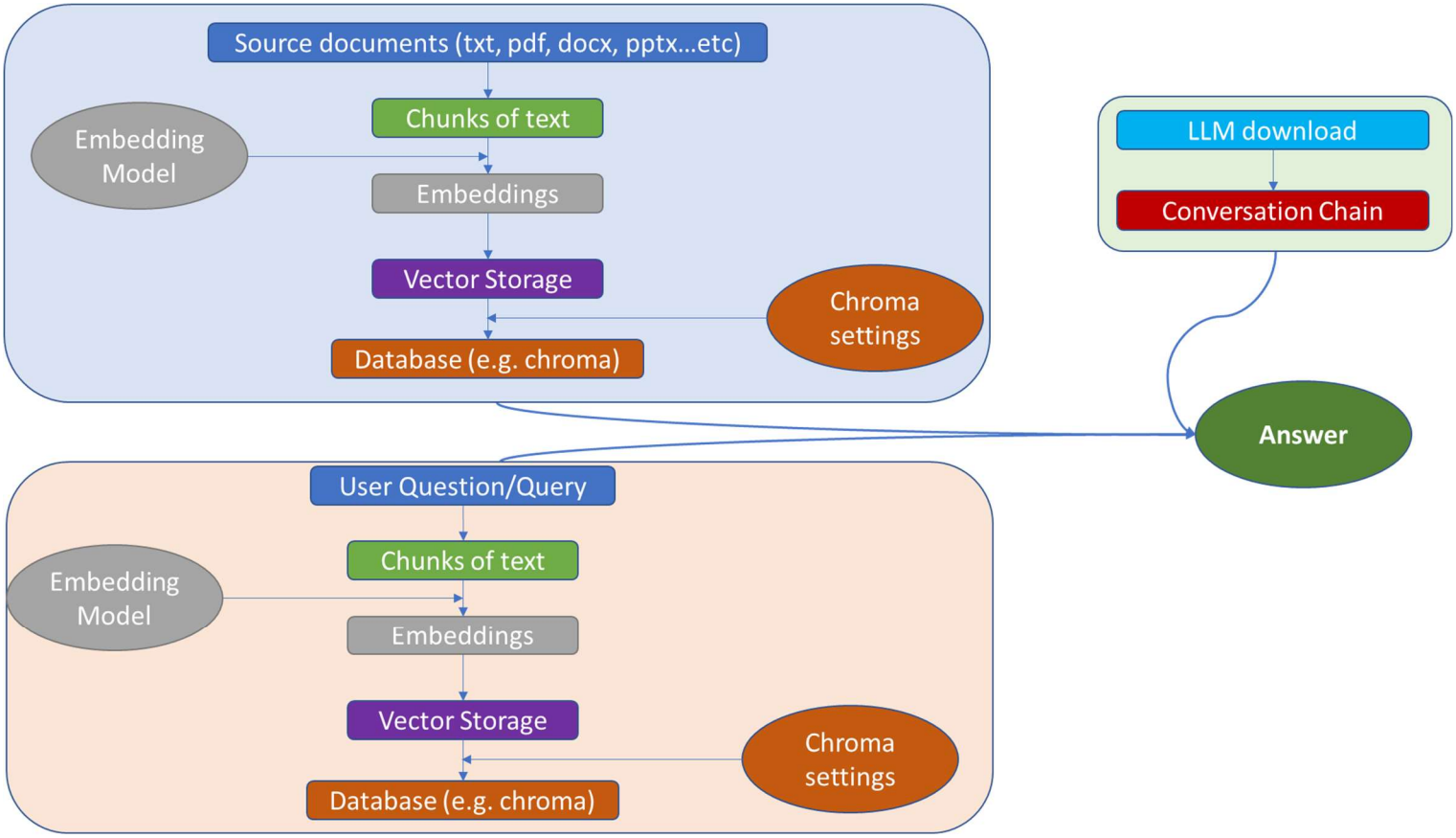

Figure 1:Typizal processing in a chatbot

Options

Building chatbots that are capable of querying and chatting with documents is getting much easier day by day. There are two main options available at the moment.

Chatbots could be fabricated using open source LLMs available on the huggingface website. The most famous LLMs at the moment are Mistral developed by Mistral AI, Llama developed by Meta, Palm developed by Google and Falcon. There are thousands of open-source large language models that are being added monthly on the HuggingFace models repository. This option has its pros and cons. The most obvious advantage is that you could have a closed loop when dealing with sensitive data. The model could be downloaded on the local server and then confidential data could be manipulated for further processing. The major disadvantage of this option is large amount of monies invested at start to build the infrastructure required for inferencing. The costs could be a bit lowered if a cloud computing service is adopted (i.e. Google colab, Microsoft Azure or AWS Sagemaker) taking into account that this service should be aligned with the company's data security policies.

Chatbots could also be built using licence based LLMs with their corresponding APIs. The most famous option available now is OpenAI GPTs 3.5 & 4 through APIs. This option is advantageous as the processing is done on the host's GPUs. The monthly subscription is going to be the only cost invested. The major caveat here is that the end user would not have any control on the data storage on the cloud. This could potentially be a threat to confidentiality.

Costs

The open source LLMs are free of charge, and many can be used commercially under the APACHE 2.0 license. The main costs under the LLMs are dedicated to the cost of the infrastructure. High computing power is a must when dealing with large language modelling. The process of fine tuning requires a good graphics processing power. The cost of a computing machine required for a good performance with a large language model could be in the excess of £5,000. The high cost would be invested once. High computing power could be accessible through Google Colab Pro, Microsoft Azure, or AWS Sagemaker. The hourly price for a decent computer to do the finetuning and inferencing starts from $2/hr.

The other alternative would be through using API based services such as OpenAI GPT4. The price of processing is circa $0.0004/1k token.

Component files of the model:

Overview

The chatbot is composed of 5 files as shown below:

- ***Requirements:*** Defines the packages that need to be installed before undertaking the coding process of the chatbot.
- ***Environment:*** Highlights a number of variables that could be loaded into the main project code.
- ***Constants:*** contains the values of some parameters set beforehand. These parameters could be changed for the purpose of other chatbots according to the needs.
- ***Ingest***: Provides a number of functions required for parsing and preparation of the input documents for future steps.
- ***PrivateGPT***: Paves the way for downloading the chosen LLM as well as its deployment when getting a query from the user.

The main interface used in writing the code in each of the files is Visual Studio Code (VS code). You can download and install Visual Studio Code through the following link: https://code.visualstudio.com/download. Further explanation of the contents of every document will be provided below.

For the purpose of this chatbot, I created a folder named source_documents where I saved all the text files to be imported within this chatbot for further processing. The end-user can change the name of the folder as required and this change has to be reflected in different code blocks.

Requirements

There is a number of packages that the end-user would need to install in order to have the full code functioning properly as shown below:

- llama-index: It provides appropriate tools to support data ingestion from various sources, vector databases for data indexing, and query interfaces for querying large documents. In short, Llama Index is a one-stop shop for building retrieval augmented generation applications.
- langchain: LangChain is a new library written in Python and JavaScript that helps developers work with Large Language Models (or LLM for short) such as Open AIs GPT-4 to develop complex solutions.

- llamacpp: llama. cpp - an excellent open source library designed to get Llama and related models (Llama 2, Alpaca, etc.) running on consumer hardware, with lots of options for optimizations to make the models smaller and faster. llama-cpp-python - a set of Python bindings for llama.
- python-dotenv: Python-dotenv is a library that enables secure management of environment variables in Python applications. It allows you to separate sensitive information from your application code, enhancing security.
- chromadb: Chroma is the open-source embedding database. Chroma makes it easy to build LLM apps by making knowledge, facts, and skills pluggable for LLMs.
- Os: The OS module in Python provides functions for interacting with the operating system. OS comes under Python's standard utility modules. This module provides a portable way of using operating system-dependent functionality.
- glob: The glob module, which is short for global, is a function that's used to search for files that match a specific file pattern or name. It can be used to search CSV files and for text in files.
- gpt4all: This is a large language model that is available for use on local machines without the need for any future access to the internet. It provides a wide range of functionalities and it is available for commercial use under the APACHE 2.0 license.

The main method used to install all these modules/libraries is through the terminal/command prompt and then typing a line of code “pip install THE_REQUIRED_MODULE” (i.e. pip install langchain)

If you faced an error due to not having pip installed on your machine, you can find good guidance in the following link : https://www.geeksforgeeks.org/how-to-install-pip-on-windows/.

*Disclaimer: *There might be a number of dependencies that have not been mentioned here and that is due to the fact that the I have worked on various projects and have hundreds of packages installed on my machine. In case there are any errors when executing the code, you could install the required package(s) using the pip command explained above.*

Environment

The main purpose of this document is to define a number of variables for the environment set for this project. This document is optional and could be embedded within the ingest.py file. It is a good practice to have a separate .env file if the end user is going to build a virtual environment for every project. If the end-user set a virtual environment, VS code will prompt the end-user to provide the requirements.txt and .env files to complete the set up.

This file contains the value for the persist directory for the Chroma database, the large language model name, the embedding model name in addition to other variables as detailed in Appendix 1.

Constants

The purpose of having Constants.py file is to set further details for the chroma database and the persist directory for the database. It is a good practice to have the constant file if the end user is planning on building a graphical user interface using HTML. The whole interface could be saved in the constants.py file and then recalled in the main code if the chatbot is going to be hosted by a service such as chainlit or streamlit. This is quite efficient in the reduction of cluttering the code.

Appendix 2 explains in a greater level of details the values set for the constants used within the code and the virtual environment set.

Ingest

Ingest.py file paves the way for importing and processing the text documents. The file starts with installing the required packages, if the end-user does not prefer to have requirements.txt file, alongside importing some important libraries. This files makes use of all the work created in requirements.txt, .env and constants.py files.

The main functions embedded within this file include:

- Loading environment variables from the .env file
- Importing document loaders (i.e., PowerPoint, Word, webpages…etc)
- Mapping the file extensions to document loaders (i.e., docx to word reader, pptx to powerpoint reader , html to html reader…etc)
- Uploading the document(s) to the memory and mapping the document to the relevant loader as explained above.
- Processing the document(s) by splitting the text into chunks with an overlap amount.
- Creating embeddings for the processed chunks of text using pre-chosen embedding model (The chosen model in our case was all-MiniLM-L6-v2).
- Storing embeddings in a vector store linked to a database (Chroma in our case). This is quite vital for a future step in the chain of conversation between the end-user and the chatbot.

Appendix 3 provides all the details related to the process of uploading, importing, embedding, and processing of source documents.

PrivateGPT

Like ingest.py, privateGPT.py makes use of all the lines of code created in requirements.txt, .env and constants.py. This file aims at importing a large language model (GPT4All and Llama in our case), creating a conversation chain (using langchain), initializing an embedding model for the queries (The chosen model in our case was all-MiniLM-L6-v2), storing the conversation and give agreeable results to the end user. The file also gives the user the chance of choosing either GPT4All or Llama for the inference process as well as the capability to show or hide the source documents used to arrive at this result.

Appendix 4 showcases how a large language model could be imported and coupled with the ingest file alongside all the values set for the environment and the constant to build fully functioning chatbot inside the windows/mac terminal environment.

Limitations & Next steps

The code prepared above showcases the extent to which people and organisations can reach when creating a chatbot. The chatbot created does not have a user-friendly interface. It relies on running the ingest.py first in the terminal followed by running privateGPT.py in the terminal. The terminal would give the end user all the instructions and messages that would help with the inferencing process as a whole. However, this obliges the end user to have some knowledge about terminal/command prompt and how to run python code in either of them.

The ingest.py and privateGPT.py could be further embedded into streamlit or a chainlit environment for a better graphical user interface. HTML code could be also created on the side for an enhanced look on both streamlit and chainlit. The end user has to take into account to install both streamlit or chainlit in their machines using PIP as explained before.

Chatbots are now linked to the idea of having agents where every agent is specialised in a specific task or set of tasks. There is then one main model which controls the different agents and align them all within a chosen structure. This paves the way to Artificial General Intelligence where a main model could delegate the tasks provided by the end-user to a set of agents working in the background and providing a satisfactory result to the end-user through the main model.

Reference

The main idea of creating this chatbot comes from Abid Saudagar. I added a large number of comments in the code to be make it more understandable and user friendly. In addition, I indented and removed unusable code blocks. I also created this report to provide a holistic dummy guide towards building this chatbot. The link to the original video is: https://www.youtube.com/watch?v=kUxfr2i2zn8.

Appendices

Appendix 1 – Code for the environment (.env) file

```
PERSIST_DIRECTORY=db
MODEL_TYPE=GPT4All
MODEL_PATH=models/ggml-gpt4all-j-v1.3-groovy.bin
EMBEDDINGS_MODEL_NAME=all-MiniLM-L6-v2
MODEL_N_CTX=1000
MODEL_N_BATCH=8
TARGET_SOURCE_CHUNKS=4
```

Appendix 2 – Code for the constants.py file

```
import os
from dotenv import load_dotenv
from chromadb.config import Settings

load_dotenv()

#Define the folder for story database
PERSIST_DIRECTORY=os.environ.get('PERSIST_DIRECTORY')
if PERSIST_DIRECTORY is None:
    raise Exception ('Please set the PERSISTENT_DIRECTORY environment
variable')

#Define the Chroma settings
CHROMA_SETTINGS= Settings(
    persist_directory=PERSIST_DIRECTORY,
    anonymized_telemetry=False
)
```

Appendix 3 – Code for the ingest.py file

```
#Installing required pacakges
# !Pip install os
# !Pip install glob
# !Pip install langchain
# !Pip install python-dotenv
# !Pip install chromadb
# !Pip install sentence-transformers
# !Pip install transformers


#Importing required libraries
import os
import glob
import chromadb

from chromadb.config import Settings
from typing import List
from dotenv import load_dotenv
from multiprocessing import Pool
from tqdm import tqdm
from langchain.document_loaders import (
    CSVLoader,
    EverNoteLoader,
    TextLoader,
    UnstructuredEmailLoader,
    UnstructuredHTMLLoader,
    UnstructuredMarkdownLoader  ,
    UnstructuredODTLoader,
    UnstructuredPowerPointLoader,
    UnstructuredWordDocumentLoader,
    UnstructuredEPubLoader,
    UnstructuredHTMLLoader,
    UnstructuredMarkdownLoader,
    PyMuPDFLoader,
    PyPDFLoader
)
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import Chroma
from langchain.embeddings import HuggingFaceBgeEmbeddings
from langchain.docstore.document import Document
from constants import CHROMA_SETTINGS #This line of code imports the chroma
settings defined in constants.py file created in the same workspace.

#Loading environment variables
if not load_dotenv:
   print("Could not load.env file or it is empty. Please check it exists and
is readable.")
```

```
    exit(1)



chunk_size=1000
chunk_overlap=100
persist_directory=os.environ.get('PERSIST_DIRECTORY')
source_directory=os.environ.get('SOURCE_DIRECTORY','source_documents')
embeddings_model_name=os.environ.get('EMBEDDINGS_MODEL_NAME')

#Map file extensions to document loaders and their arguments
LOADER_MAPPING={
    ".csv":(CSVLoader,{}),
    ".doc":(UnstructuredWordDocumentLoader,{}),
    ".docx":(UnstructuredWordDocumentLoader,{}),
    ".enex":(EverNoteLoader,{}),
    ".epub":(UnstructuredEPubLoader,{}),
    ".html":(UnstructuredHTMLLoader,{}),
    ".md":(UnstructuredMarkdownLoader,{}),
    ".odt":(UnstructuredODTLoader,{}),
    ".ppt":(UnstructuredPowerPointLoader,{}),
    ".pptx":(UnstructuredPowerPointLoader,{}),
    ".pdf":(PyMuPDFLoader,{}),
    ".txt":(TextLoader,{"encoding":"utf8"})
}

#Function to upload a single PDF document
def load_single_document(file_path:str) -> List[Document]:
    ext="."+file_path.rsplit(".",1)[-1].lower()
    if ext in LOADER_MAPPING:
        loader_class,loader_args=LOADER_MAPPING(ext)
        loader=loader_class(file_path,**loader_args)
        return loader.load()
    raise ValueError(f"Unsupported file extension'{ext}'")

#Function to upload multiple PDFs
def load_documents(source_dir:str,ignored_files:List[str]=[]) ->
List[Document]:
    """
    Loads all documents from the source documents directory and ignoring
specified files
    """
    all_files=[]
    for ext in LOADER_MAPPING:
        all_files.extend(
            glob.glob(os.path.join(source_dir,f"**/*{ext.lower()}"),recursive=
True)
        )
        all_files.extend(
```

```
            glob.glob(os.path.join(source_dir,f"**/*{ext.upper()}"),recursive=
True)
        )
        filtered_files=[file_path for file_path in all_files if file_path not
in ignored_files]

        with Pool(processes=os.cpu.count()) as pool:
            results=[]
            with tqdm(total=len(filtered_files),desc='Loading new
documents',ncols=80) as pbar:
                for i,docs in
enumerate(pool.imap_unordered(load_single_document,filtered_files)):
                    results.extend(docs)
                    pbar.update
        return results

#Create a function to process uploaded documents and split them into chunks
required for the embedding process
def process_documents(ignored_files:List[str]=[]) ->List[Document]:
    """
    Load docuemnts and split in chunks
    """
    print(f"Loading documents from {source_directory}")
    documents=load_documents(source_directory,ignored_files)
    if not documents:
        print("No new documents to load")
        exit(0)
    print(f"Loaded{len(documents)} new documents from {source_directory}")
    text_splitter=RecursiveCharacterTextSplitter(chunk_size=chunk_size,chunk_o
verlap=chunk_overlap)
    texts=text_splitter.split_documents(documents)
    print(f"Split into {len(texts)} chunks of text (max. {chunk_size} tokens
each)")
    return texts

#Create a function to create embeddings for text chunks and store them into
vectors
def
does_vectorstore_exist(persist_directory:str,embeddings:HuggingFaceBgeEmbeddin
gs) -> bool:
    """
    Checks if VectorStore exists
    """
    db = Chroma(persist_directory=persist_directory,
embedding_function=embeddings)

    if not db.get()['documents']:
        return False
```

```
    return True

def main():
    # Create embeddings
    embeddings=HuggingFaceBgeEmbeddings(model_name=embeddings_model_name)
    #Chroma client
    chroma_client=chromadb.PersistentClient(settings=CHROMA_SETTINGS,path=pers
ist_directory)

    if does_vectorstore_exist(persist_directory,embeddings):
        #Update and store locally vectorstore
        print(f"Appending to existing vectorstore at {persist_directory}")
        db = Chroma(persist_directory=persist_directory,
embedding_function=embeddings, client_settings=CHROMA_SETTINGS,
client=chroma_client)

        collection=db.get()
        texts=process_documents([metadata['source'] for metadata in
collection['metadatas']])
        print (f"Creating embeddings. May take some minutes...")
        db.add_documents(texts)
    else:
        #Create and store locally vectorstore
        print("Creating new vectorstore")
        texts=process_documents()
        print(f"Creating embeddings. May take some minutes...")
        db = Chroma.from_documents(texts, embeddings,
persist_directory=persist_directory, client_settings=CHROMA_SETTINGS,
client=chroma_client)

    db.persist()
    db=None
    print(f"Ingestion complete! You can now run PrivateGPT.py to query your
documents")


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

Appendix 4 – Code for the privateGPT.py file

```
#Import required libraries
import chromadb
import os
import argparse
import time
from dotenv import load_dotenv
from langchain.chains import RetrievalQA
from langchain.embeddings import HuggingFaceBgeEmbeddings
from langchain.callbacks.streaming_stdout import
StreamingStdOutCallbackHandler
from langchain.vectorstores import Chroma
from langchain.llms import GPT4All, Llamacpp
from constants import CHROMA_SETTINGS


#Checking if the .env file exists and load the variables from it in the
backend.
if not load_dotenv():
    print("Could not load .env file or it is empty. Please check if it exists
and is readable.")
    exit(1)

embeddings_model_name=os.environ.get("EMBEDDINGS_MODEL_NAME")
persist_directory=os.environ.get("PERSIST_DIRECTORY")
model_type=os.environ.get("MODEL_TYPE")
model_path=os.environ.get("MODEL_PATH")
model_n_ctx=os.environ.get("MODEL_N_CTX")
model_n_batch=int(os.environ.get("MODEL_N_BATCH"),8)
target_source_chunks=int(os.environ.get("TARGET_SOURCE_CHUNKS"),4)

#This is the main function which prepares the LLms and recalls the chromadb
alongside the embedding model. This function also includes the preparation of
questions and answers.
def main():
    #Parse the command line arguments
    args=parse_arguments()
    embeddings= HuggingFaceBgeEmbeddings(model_name=embeddings_model_name)
    chroma_client=chromadb.PersistentClient(settings=CHROMA_SETTINGS,path=pers
ist_directory)
    db=Chroma(persist_directory=persist_directory,embedding_function=embedding
s,client_settings=CHROMA_SETTINGS,client=chroma_client)
    retriever=db.as_retriever(search_kwargs={"k":target_source_chunks})

    #activate/deactivate the streaming Stdout callback for LLMs
    callbacks=[] if args.mute_stream else [StreamingStdOutCallbackHandler()]

    #Prepare the LLM
```

```
    match model_type:
        case"LlamaCpp":
            llm=Llamacpp(model_path=model_path,max_tokens=model_n_ctx,n_batch=
model_n_batch,callbacks=callbacks,verbose=False)
        case "GPT4ALL":
            llm=GPT4All(model=model_path,max_tokens=model_n_ctx,backend='gptj'
,n_batch=model_n_batch,callbacks=callbacks,verbose=False)
        case _default:
            #raise exception if model type is not supported
            raise Exception(f"Model Type {model_type} is not supported. Please
choose one of the following: LlamaCpp or GPT4All")

    qa=RetrievalQA.from_chain_type(llm=llm,chain_type="stuff",retriever=retrie
ver,return_source_documents=not args.hide_source)

    #Interactive questions and answers
    while True:
        query=input("\n Enter a query: ")
        if query== "exit":
            break
        if query.strip()=="":
            continue

        #Get the answer from the chain
        start=time.time()
        res=qa(query)
        answer,docs=res['result'],[] if args.hide_source else res
['source_documents']
        end=time.time()

        #Print the result
        print(" \n\n > Question:")
        print(query)
        print(f"\n> Answer (took {round(end-start,2)}s.):")
        print(answer)

        #Print the relevant sources used for the answer
        for document in docs:
            print("\n>"+document.metadata["source"]+":")
            print(document.page_content)

#Thiis function allows the end user to show/hide source documents and to
show/hide the insertion of every word into the LLMs for embedding, chunking
and storing of vectors.
def parse_arguments():
    parser=argparse.ArgumentParser(description='PrivateGPT: Ask questions to
your documents without an internet connection,'
                                    "using the power of LLMs.")
```

```
    parser.add_argument("--hide source","-S",action='store_true',
                        help='Use this flag to disable printing of source
documents used for answers.')
    parser.add_argument("--mute-stream","-M",
                        action='store_true',
                        help='Use this flag to diable the streaming StdOut
callback for LLMs.')
    return parser.parse_args

#This allow running the whole code above and embed them under the main
function
if __name__=="__main__":
    main()
```