D. Grigorov, *Intermediate Python and Large Language Models*
https://doi.org/10.1007/979-8-8688-1475-4_3

3. Building Advanced Applications Powered by LLMs with LangChain and Python

Dilyan Grigorov[1]
(1) Varna, Varna, Bulgaria

In the rapidly evolving world of artificial intelligence, large language models (LLMs) have emerged as powerful engines that drive innovative applications, transforming the way we interact with technology. This chapter is a deep dive into the sophisticated strategies and techniques that harness the full potential of these models. Here, we explore how to go beyond basic implementations and craft complex, robust systems that leverage LLMs to address real-world challenges.

At the core of our discussion is LangChain—a versatile framework designed to streamline the integration of LLMs into advanced application architectures. LangChain provides a modular and extensible environment that simplifies the orchestration of language model tasks, allowing developers to build applications that can manage multistep reasoning, handle dynamic interactions, and maintain contextual continuity across extended dialogues. Coupled with the power and flexibility of Python, LangChain equips you with the tools to push the boundaries of what is possible in modern software development.

In this chapter, we begin by revisiting the foundational concepts behind LLMs, setting the stage for a more nuanced understanding of their capabilities and limitations. We then transition into an exploration of LangChain's architecture, examining its key components and how they work together to facilitate advanced workflows. Through detailed examples and hands-on exercises, you will learn how to implement complex pipelines that integrate external data sources, manage iterative processing, and optimize performance under demanding conditions.

As we progress, the focus shifts to the practical challenges encountered when building advanced LLM-powered applications. We will discuss strategies for fine-tuning model behavior, mitigating errors, and ensuring scalability in production environments. Special emphasis is placed on designing systems that not only perform efficiently but also maintain high levels of reliability and security. This chapter provides insights into best practices for monitoring, debugging, and continuously improving your applications, ensuring that they remain robust in the face of evolving requirements and emerging threats.

Moreover, we will highlight cutting-edge use cases that demonstrate the transformative impact of advanced LLM applications. From intelligent virtual assistants that seamlessly manage complex conversations to automated content generation systems capable of nuanced analysis and synthesis, you will see how the principles discussed can be applied to a diverse array of challenges. By dissecting these real-world examples, you will gain a deeper appreciation of the potential for innovation when combining LangChain's orchestration capabilities with Python's rich ecosystem.

Whether you are an experienced developer looking to elevate your skill set or a curious practitioner eager to explore the next frontier in AI application development, this chapter is designed to equip you with the knowledge and tools necessary to build advanced, production-grade systems. By the end of our journey, you will have a comprehensive understanding of how to harness LLMs effectively, enabling you to create applications that are not only intelligent but also adaptive, scalable, and ready for the challenges of tomorrow.

In the next pages, we will

- **Build a YouTube Video Summarizer**
  - Automatically transcribe and summarize long YouTube videos for quick content digestion
- **Create a GitHub repository chatbot**
  - Interact with code bases conversationally by indexing and querying repository files
- **Develop a financial report analysis tool**
  - Analyze and extract insights from financial documents using AI-driven retrieval and Q&A
- **Enhance blog content with Google Search**
  - Use LLMs and live web data to intelligently expand and enrich blog posts
- **Automate YouTube scriptwriting**
  - Generate structured, engaging scripts from video transcripts with GPT models
- **Design an AI-powered email generator**
  - Instantly craft professional, personalized email responses using a customizable prompt
- **Analyze CSV data with AI assistance**
  - Load, summarize, and visualize datasets with natural language commands and visual tools

Each app is presented with

- Step-by-step implementation instructions
- Clear explanations of LangChain and OpenAI integrations
- Tips for optimizing performance, usability, and scalability

By the end of this chapter, you'll be equipped to build your own intelligent, production-ready applications with Python and LLMs.

App 1: YouTube Video Summarizer

In the digital age, YouTube has become a vast repository of knowledge, offering millions of videos on various topics, from educational lectures to industry insights. However, watching long videos to extract key information can be time-consuming. This is where a **YouTube Video Summarizer with LangChain** comes into play.

A **YouTube Video Summarizer** is an AI-powered tool that automatically transcribes and summarizes YouTube videos, providing users with a concise and structured overview of the content. By leveraging **LangChain**, a framework designed for building applications with **large language models (LLMs)**, the summarizer efficiently processes video transcripts and distills essential insights.

This tool utilizes **natural language processing (NLP)** to extract meaningful information, making it easier for users to grasp the key points of a video in seconds rather than minutes or hours. Whether you are a researcher, student, or content creator, a **YouTube Video Summarizer with LangChain** enhances productivity by offering quick and accurate video summaries, helping you stay informed without watching entire videos.

How to Build the App

Step 1: Get Your OpenAI API Key

You need to get your OpenAI API key here: https://platform.openai.com/settings/organization/api-keys.

Step 2: Run the Following Commands

Run the following commands in your environment—in our case, this is Google Colab—to install libraries needed:

```
!pip install langchain==0.3.23 activeloop-deeplake==3.9.5 openai==1.3.12 tiktoken==0.7.0 langcha
!pip install -q yt_dlp
!pip install -q git+https://github.com/openai/whisper.git
Also install ffmpeg:
conda install ffmpeg
For Google Colab:
!apt-get update -qq && apt-get install -y ffmpeg
```

NoteDue to the security precautions taken by YouTube, you need to download a browser extension to download the cookies from your browser for your desired video.

For example, for Chrome you can use: https://chromewebstore.google.com/detail/get-cookiestxt-locally/cclelndahbckbenkjhflpdbgdldlbecc. Then, upload the cookie file to your Google Colab files.

Step 3: Execute the Following Command

Execute the following command to download your video with your desired file name:

```
!yt-dlp --cookies cookies.txt -f "bestvideo[ext=mp4]+bestaudio[ext=m4a]/best[ext=mp4]" -o "my_v
```

Explanation of the command:

- `-f "bestvideo[ext=mp4]+bestaudio[ext=m4a]/best[ext=mp4]"`
  - This ensures that yt-dlp downloads the **best quality MP4 video** and **best M4A audio** and then merges them.
  - If a single **MP4 format video with audio** is available, it downloads that.

- `-o "my_video.mp4"`
  - This sets the output filename as `my_video.mp4`.
- `--cookies cookies.txt`
  - This allows yt-dlp to use authentication for downloading restricted videos.

Step 4: Import the Whisper Model and Process the Video

```
import whisper
model = whisper.load_model("base")
result = model.transcribe("my_video.mp4")
print(result['text'])
```

This Python script utilizes OpenAI's Whisper model for automatic speech recognition (ASR) to transcribe the audio from a given video file. It begins by importing the Whisper library, which is responsible for handling the transcription process. Next, it loads a pretrained Whisper model, specifically the "base" version, which is a lightweight model compared to larger variants like "medium" or "large." If the model is not already available locally, it will be automatically downloaded from OpenAI's servers.

Once the model is loaded, the script processes the specified video file by extracting its audio and converting the spoken content into text. Finally, the transcribed text is extracted from the result and printed to the console.

Step 5: Read the Written Content in a File

```
with open ('text.txt', 'w') as file:
   file.write(result['text'])
```

Step 6: Use LangChain to Split a Text File into Smaller Chunks

Let's use LangChain to split a text file into smaller chunks for further processing, such as feeding into a language model.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.docstore.document import Document
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=0, separators=[" ", ",", "\n"]
)
with open('text.txt') as f:
    text = f.read()
texts = text_splitter.split_text(text)
docs = [Document(page_content=t) for t in texts[:4]]
```

Step-by-Step Explanation

1. **Import necessary modules from LangChain**
   1. **RecursiveCharacterTextSplitter**: A utility for breaking long texts into smaller pieces while trying to maintain meaning
   2. **Document**: A simple wrapper around text content, useful when dealing with large documents
2. **Initialize the text splitter**

1. The RecursiveCharacterTextSplitter is configured with
   1. chunk_size=1000: Each chunk will have a maximum of **1000 characters**.
   2. chunk_overlap=0: No overlap between chunks.
   3. separators=[“ ”, “,”, “\n”]: The text will be split **preferentially** at spaces, then commas, then newlines.

3.**Read text from a file (text.txt)**

1. The text content is loaded into memory as a single string.

4.**Split the text into smaller chunks**

1. split_text(text): The loaded text is split into multiple **chunks** of up to 1000 characters each.
2. The splitting occurs **recursively**, prioritizing the given separators.

5.**Convert the first four chunks into Document objects**

1. The first four chunks (texts[:4]) are wrapped in Document objects.
2. Each Document stores its corresponding chunk as page_content.

Step 7: Summarize the Preprocessed Content

```
from langchain.chains.summarize import load_summarize_chain
import textwrap
from langchain_openai import OpenAI
from langchain.chains import LLMChain
from langchain.chains.summarize import load_summarize_chain
from langchain.prompts import PromptTemplate
# Initialize OpenAI LLM
llm = OpenAI(api_key="sk-proj-Mi0xm6IEXSZ8ULyof8caT3BlbkFJNHaISzb3nz3hsau8tqyn", model="gpt-3.5
chain = load_summarize_chain(llm, chain_type="map_reduce")
output_summary = chain.invoke(docs)
wrapped_text = textwrap.fill(output_summary["output_text"], width=100)
```

Step-by-step execution:

1.**Import necessary libraries**

1. `load_summarize_chain`: A utility to create a summarization pipeline
2. `textwrap`: Used to format the output text
3. `OpenAI`: Initializes OpenAI’s GPT model for text processing
4. `LLMChain`: A generic LangChain wrapper for using LLMs
5. `PromptTemplate`: Allows customization of prompts for the LLM

2.**Initialize OpenAI language model**

1. The `OpenAI` LLM is initialized with
   1. The **GPT-3.5 Turbo Instruct** model
   2. **Temperature** = **0**, ensuring deterministic (consistent) responses
   3. The **API key** (which should be kept secret)

3.**Load a summarization chain**

1. The `load_summarize_chain(llm, chain_type="map_reduce")` initializes a **two-step summarization pipeline**:
   1. **Map Stage**: Each document is summarized individually.
   2. **Reduce Stage**: The individual summaries are combined into a final, coherent summary.

4.**Invoke the summarization chain**

1. `chain.invoke(docs)`: The chain takes `docs` (a list of `Document` objects) and processes them.
2. `output_summary["output_text"]`: Extracts the final summarized text.

5. **Format the summary output**
    1. `textwrap.fill(output_summary["output_text"], width=100)` wraps the summary text so that lines do not exceed **100 characters** in width.

Step 8: Define a Prompt Template Using LangChain's PromptTemplate

The following code defines a prompt template using LangChain's PromptTemplate to structure input for an LLM (large language model), such as OpenAI's GPT models:

```
prompt_template = """Write a concise bullet point summary of the following:
{text}
CONSCISE SUMMARY IN BULLET POINTS:"""
BULLET_POINT_PROMPT = PromptTemplate(template=prompt_template,
                        input_variables=["text"])
```

Step 9: Summarization Pipeline

```
chain = load_summarize_chain(llm,
                             chain_type="stuff",
                             prompt=BULLET_POINT_PROMPT)
output_summary = chain.run(docs)
wrapped_text = textwrap.fill(output_summary,
                             width=1000,
                             break_long_words=False,
                             replace_whitespace=False)
print(wrapped_text)
```

This code sets up a summarization pipeline using LangChain and an LLM, such as OpenAI's GPT-3.5 Turbo. The process begins by initializing a summarization chain with a specific configuration. The chain type is set to process all text at once, rather than breaking it into smaller sections. A custom prompt template is used to instruct the model to generate a structured bullet-point summary.

Once the summarization chain is created, it is executed using a list of documents as input. The chain processes the text and generates a concise summary. The resulting summary is then formatted for better readability by ensuring that lines do not exceed a certain width, words are not split across lines, and whitespace formatting is preserved. Finally, the formatted summary is displayed as output.

App 2: Chat with a GitHub Repository

This Python application enables users to interact with a GitHub repository using natural language. It utilizes **LangChain**, **OpenAI embeddings**, and **FAISS vector storage** to process and retrieve relevant code snippets, documentation, and README contents from a repository.

How It Works

1. **Fetches Repository Data**: Uses the GitHub API to retrieve all files in the repository

2. **Embeds and Indexes Content**: Converts text into embeddings for efficient search
3. **Conversational Retrieval**: Allows users to ask questions and get relevant information
4. **Memory Support**: Maintains context in ongoing conversations for a better chat experience

Step 1: Select a GitHub Repository and Download It As Zip

For example, https://github.com/milaan9/07_Python_Advanced_Topics, and get its username and repo name—in this case, its **milaan9**, and the name of the repo is **07_Python_Advanced_Topics**. Or in other words, you can find it in the form https://github.com/{user_name}/{repo_name}.

Step 2: Install All Libraries Required

```
!pip install langchain==0.3.23 openai==1.3.12 faiss-cpu==1.8.0 tiktoken==0.7.0 requests==2.31.0
!pip install langchain-community==0.3.23
```

Step 3: Import the Libraries and Obtain the Needed API Keys

For OpenAI: https://platform.openai.com/api-keys

For GitHub: https://github.com/settings/tokens

```
import os
import requests
from dotenv import load_dotenv
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.llms import OpenAI
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI
load_dotenv()
GITHUB_TOKEN = os.getenv("GITHUB_TOKEN")
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
headers = {"Authorization": f"token {GITHUB_TOKEN}"}
```

Step 4: Get Repository Content

The function is designed to **retrieve the contents** of a GitHub repository using the GitHub API. It allows you to access files and directories within a repository.

1. **It accepts three inputs**
   1. The **owner** of the repository (a username or organization name).
   2. The **repository name** to fetch data from.
   3. An optional **path** specifying a file or folder within the repository. If no path is provided, it retrieves the root directory.
2. **It builds a URL**
   1. The function constructs a web address following GitHub's API format. This URL points to the requested repository and its contents.
3. **It sends a request to GitHub**
   1. A request is made to GitHub's servers to fetch the contents of the specified file or folder.

4. **It checks for a successful response**
    1. If GitHub responds successfully, the function extracts and returns the content in a structured format (as data).
    2. If the request fails (e.g., due to incorrect repository details, permission issues, or rate limits), an error message is displayed, and an empty response is returned.
5. **It handles different repository structures**
    1. If the request targets a directory, the function retrieves a list of its files and subdirectories.
    2. If it targets a file, it fetches the file's content and relevant metadata.

```
def get_repo_contents(owner, repo, path=""):
    url = f"https://api.github.com/repos/{owner}/{repo}/contents/{path}"
    response = requests.get(url, headers=headers)
    if response.status_code == 200:
        return response.json()
    else:
        print("Error fetching repo contents:", response.json())
        return []
```

Step 5: Fetch All Files

```
def fetch_all_files(owner, repo, path="", collected_files=None):
    if collected_files is None:
        collected_files = {}
    contents = get_repo_contents(owner, repo, path)
    for item in contents:
        if item["type"] == "file":
            file_content = requests.get(item["download_url"], headers=headers).text
            collected_files[item["path"]] = file_content
        elif item["type"] == "dir":
            fetch_all_files(owner, repo, item["path"], collected_files)
    return collected_files
```

In this function, we **retrieve all files** from a GitHub repository, including those inside subdirectories. It works by **recursively** navigating through the repository's structure and collecting file contents.

Step-by-Step Explanation

1. **It initializes a dictionary to store files**
    1. If no dictionary is provided, an empty one is created to store file paths and their contents.
2. **It retrieves repository contents**
    1. The function calls another function to fetch the list of files and folders at a given location in the repository.
    2. If no specific path is provided, it starts from the root directory.
3. **It loops through each item in the retrieved list**
    1. If the item is a **file**, the function
        1. Downloads its content from GitHub
        2. Stores the file's path as a key and its content as a value in the dictionary

    2. If the item is a **directory**, the function
        1. Calls itself again (recursion), using the directory's path as the new starting point
        2. This ensures all nested files and folders are processed
4. **It returns a dictionary containing all files**
    1. After processing all files and directories, the function returns a dictionary where
        1. Each **key** represents a file's path within the repository
        2. Each **value** contains the corresponding file's content

Step 6: Creating a Searchable Database

```
def create_vector_db(files):
    texts = []
    for path, content in files.items():
        texts.append(f"### {path}\n{content}")
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
    docs = text_splitter.create_documents(texts)
    embeddings = OpenAIEmbeddings()
    vectorstore = FAISS.from_documents(docs, embeddings)
    return vectorstore
```

In this function, we **create a searchable database** from a collection of files by converting their contents into numerical representations (embeddings) that allow efficient retrieval.

Step-by-Step Explanation

1. **It prepares the text data**
    1. The function starts with an empty list to store text data.
    2. It loops through each file in the input dictionary, which contains file paths as keys and their contents as values.
    3. Each file's content is formatted with a header that includes the file path, ensuring that file names remain associated with their contents.
2. **It splits the text into chunks**
    1. Since some files may be large, the function breaks them into smaller chunks.
    2. A text-splitting tool is used to divide the text while maintaining some overlap between chunks to preserve context.
    3. This ensures that each chunk is not too large for processing while still making sense when analyzed.
3. **It converts text into embeddings**
    1. The function uses an embedding model to transform the text chunks into **numerical vectors**.
    2. These embeddings capture the **semantic meaning** of the text, making it possible to search for similar content based on meaning rather than exact words.
4. **It stores the embeddings in a searchable database**
    1. A specialized database (FAISS) is used to store these embeddings efficiently.
    2. FAISS allows quick searching and retrieval of relevant text based on similarity to a given query.
5. **It returns the searchable database**
    1. The final result is a structured vector database that enables quick searches for relevant file contents.

Step 7: Creating the Actual Chatting Feature Function

```
def chat_with_repo(owner, repo):
    files = fetch_all_files(owner, repo)
    vectorstore = create_vector_db(files)
    retriever = vectorstore.as_retriever()
    memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
    chat = ConversationalRetrievalChain.from_llm(
        llm=ChatOpenAI(model_name="gpt-4", temperature=0.5),
        retriever=retriever,
        memory=memory
    )
    print("Chat with the GitHub repository! Type 'exit' to quit.")
    while True:
        query = input("You: ")
        if query.lower() == "exit":
            break
        response = chat.run(query)
        print("Bot:", response)
```

It allows a user to **interactively chat** with a GitHub repository by retrieving relevant information from its contents.

Step-by-Step Explanation

1. **It fetches all files from the repository**
    1. The function retrieves the entire repository's contents, including files in subdirectories.
    2. This ensures that all text-based content is available for processing.
2. **It creates a searchable vector database**
    1. The fetched files are processed and converted into a **vector database**.
    2. This allows the chatbot to **search for relevant information** efficiently.
3. **It sets up a retriever**
    1. A retriever is initialized from the vector database.
    2. The retriever helps the chatbot find relevant file contents when answering questions.
4. **It initializes memory for conversations**
    1. A memory module is added to keep track of past interactions.
    2. This allows the chatbot to maintain context throughout the conversation.
5. **It creates a conversational AI model**
    1. A **GPT-4 language model** is loaded to generate responses.
    2. The model uses the retriever to find relevant repository content when answering questions.
6. **It starts an interactive chat**
    1. The function displays a message prompting the user to start chatting.
    2. It continuously takes user input, processes it, and provides responses.
    3. If the user types `"exit"`, the chat ends.

Output:

```
Enter GitHub owner/org: milaan9
Enter repository name: 07_Python_Advanced_Topics
Chat with the GitHub repository! Type 'exit' to quit.
You: What are the advanced topics in the repo?
Bot: The advanced topics in the repository are:
1. Python Iterators
2. Python Generators
3. Python Closure
4. Python Decorators
   - Python args and kwargs
5. Python Property
6. Python RegEx
```

App 3: Financial Report Analysis App

This app is designed to streamline financial data analysis by leveraging **AI-powered document retrieval** and **natural language processing**. Built with **LangChain, FAISS, and OpenAI models**, it allows users to efficiently search and analyze financial reports, specifically those from Amazon, but it can be adapted for any financial documents.

Key Features

- **Automated PDF Parsing**: Extracts financial data from multiple PDF reports.
- **AI-Driven Search**: Uses advanced embeddings and retrieval mechanisms to provide quick answers to financial queries.
- **Efficient Data Management**: Utilizes FAISS for fast and scalable vector-based document retrieval.
- **Conversational Querying**: Enables users to ask specific questions (e.g., *"What was Amazon's revenue in Q3 2021?"*) and get direct answers.

This tool is ideal for **financial analysts, researchers, and business professionals** who need instant insights from large datasets without manually scanning through reports. Whether you're tracking revenue trends, identifying financial performance, or analyzing key business metrics, this app provides a **seamless and intelligent solution** to financial document analysis.

Step 1: Install All Required Libraries

```
!pip3 install langchain faiss-cpu pypdf openai tiktoken langchain-openai langchain-community
```

Step 2: Set Up OpenAI API Key and Add It to the Code

```
# Set API keys (Use environment variables for security)
import os
os.environ["OPENAI_API_KEY"] = “Your OpenAI Key”
```

Step 3: Import All Required Libraries

```
from langchain_openai import OpenAIEmbeddings
from langchain.vectorstores import FAISS
from langchain.text_splitter import CharacterTextSplitter
from langchain_openai import OpenAI
from langchain.chains import RetrievalQA
from langchain_openai import ChatOpenAI
from langchain.document_loaders import PyPDFLoader
import requests
import tqdm
from typing import List
```

Step 4: Process Financial Reports

```
import requests
import tqdm
from typing import List
# financial reports of amamzon, but can be replaced by any URLs of pdfs
urls = ['https://s2.q4cdn.com/299287126/files/doc_financials/Q1_2018_-_8-K_Press_Release_FILED.
        'https://s2.q4cdn.com/299287126/files/doc_financials/Q2_2018_Earnings_Release.pdf',
        'https://s2.q4cdn.com/299287126/files/doc_news/archive/Q318-Amazon-Earnings-Press-Relea
        'https://s2.q4cdn.com/299287126/files/doc_news/archive/AMAZON.COM-ANNOUNCES-FOURTH-QUAR
        'https://s2.q4cdn.com/299287126/files/doc_financials/Q119_Amazon_Earnings_Press_Release
        'https://s2.q4cdn.com/299287126/files/doc_news/archive/Amazon-Q2-2019-Earnings-Release.
        'https://s2.q4cdn.com/299287126/files/doc_news/archive/Q3-2019-Amazon-Financial-Results
        'https://s2.q4cdn.com/299287126/files/doc_news/archive/Amazon-Q4-2019-Earnings-Release.
        'https://s2.q4cdn.com/299287126/files/doc_financials/2020/Q1/AMZN-Q1-2020-Earnings-Rele
        'https://s2.q4cdn.com/299287126/files/doc_financials/2020/q2/Q2-2020-Amazon-Earnings-Re
        'https://s2.q4cdn.com/299287126/files/doc_financials/2020/q4/Amazon-Q4-2020-Earnings-Re
        'https://s2.q4cdn.com/299287126/files/doc_financials/2021/q1/Amazon-Q1-2021-Earnings-Re
        'https://s2.q4cdn.com/299287126/files/doc_financials/2021/q2/AMZN-Q2-2021-Earnings-Rele
        'https://s2.q4cdn.com/299287126/files/doc_financials/2021/q3/Q3-2021-Earnings-Release.p
        'https://s2.q4cdn.com/299287126/files/doc_financials/2021/q4/business_and_financial_upd
        'https://s2.q4cdn.com/299287126/files/doc_financials/2022/q1/Q1-2022-Amazon-Earnings-Re
        'https://s2.q4cdn.com/299287126/files/doc_financials/2022/q2/Q2-2022-Amazon-Earnings-Re
        'https://s2.q4cdn.com/299287126/files/doc_financials/2022/q3/Q3-2022-Amazon-Earnings-Re
        'https://s2.q4cdn.com/299287126/files/doc_financials/2022/q4/Q4-2022-Amazon-Earnings-Re
        ]
def load_reports(urls: List[str]) -> List[str]:
    """ Load pages from a list of urls"""
    pages = []
    for url in tqdm.tqdm(urls):
        r = requests.get(url)
        path = url.split('/')[-1]
        with open(path, 'wb') as f:
            f.write(r.content)
        loader = PyPDFLoader(path)
        local_pages = loader.load_and_split()
        pages.extend(local_pages)
    return pages
pages = load_reports(urls)
```

The code downloads Amazon's financial reports in PDF format, extracts their text, and stores the content in a list. It starts by iterating through a predefined list of URLs, downloading each PDF using the `requests` library, and saving the files locally. Then, it processes each file with `PyPDFLoader` to extract and split the text into pages, which are appended to a list. The `tqdm` library provides a progress bar to track the downloading process. Finally, the extracted text from all PDFs is stored in the `pages` list for further analysis. However, the script is missing an import for `PyPDFLoader`, which would cause an error unless added manually. Additionally, the saved PDFs are not deleted after extraction.

Step 5: Preparing and Indexing Text Data for Efficient Retrieval Using AI-Powered Search and Question Answering

(QA)

1. Splitting the Extracted Text into Smaller Chunks

```
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(pages)
```

- **What it does:**
  - The extracted text from financial reports (stored in `pages`) is often long and unstructured.
  - The `CharacterTextSplitter` takes these texts and **breaks them into smaller chunks** of 1000 characters each (`chunk_size=1000`).
  - There is **no overlap** between chunks (`chunk_overlap=0`), meaning each piece of text is distinct.
- **Why it's needed:**
  - Splitting text into smaller sections allows for **better indexing** and **faster retrieval** when querying later.

2. Generating Text Embeddings

```
embeddings = OpenAIEmbeddings()
```

- **What it does:**
  - Uses **OpenAI's embedding model** to convert text chunks into **numerical vectors**.
  - These vectors **capture the meaning of the text** so they can be efficiently searched.
- **Why it's needed:**
  - A numerical representation (embedding) allows us to perform **semantic search**—meaning we can find relevant information **even if the search query does not exactly match the words in the document**.

3. Storing and Indexing the Text Chunks in a FAISS Database

```
db = FAISS.from_documents(texts, embeddings)
```

- **What it does:**
  - Uses **FAISS (Facebook AI Similarity Search)**, a powerful **vector database**, to store the generated embeddings.
  - FAISS allows for **efficient and fast searching** of similar text embeddings.
- **Why it's needed:**
  - Instead of searching through raw text, we **search through embeddings**, making retrieval **faster and more accurate**.

4. Setting Up the AI-Powered Retrieval and QA System

```
qa = RetrievalQA.from_chain_type(llm=ChatOpenAI(model='gpt-3.5-turbo'), chain_type='stuff', ret
```

- **What it does:**
  - Uses **ChatGPT (gpt-3.5-turbo)** to answer user queries based on the indexed documents.
  - The `retriever=db.as_retriever()` ensures that **relevant text chunks are retrieved** from FAISS before being processed by the AI.
  - The `chain_type='stuff'` method combines the retrieved text into a single response.
- **Why it's needed:**
  - This setup allows the app to **answer questions** like *"What was Amazon's revenue in Q3 2021?"* based on financial reports **without needing a human to manually search the documents**.

Step 6: Ask a Question

```
qa.invoke("What is the revenue in 2021 Q3?")
```

Output:

```
{'query': 'What is the revenue in 2021 Q3?',
 'result': 'The revenue for Amazon in 2021 Q3 was $110.8 billion.'}
```

App 4: Automate and Enhance Your Blog Posts with LangChain and Google Search

Artificial intelligence is transforming the field of copywriting by acting as a powerful writing assistant. Modern language models can detect grammar and spelling errors, adjust tone, summarize content, and even expand text. However, these models sometimes lack the deep domain expertise needed to provide high-quality extensions for specific topics.

In this lesson, we'll guide you through building an application that seamlessly enhances text sections. The process starts by prompting a language model (such as ChatGPT) to generate relevant search queries based on the existing content. These queries are then used with the Google Search API to retrieve authoritative information from the Web. Finally, the most relevant results are provided as context to the model, allowing it to generate more accurate and well-informed content suggestions.

Step 1: Install All Required Libraries

```
!pip install langchain==0.0.208 deeplake==3.9.27 openai==0.27.8 tiktoken
!pip install -q newspaper3k==0.2.8 python-dotenv
!pip install lxml_html_clean
```

Step 2: Define Three Variables—Title, Text All, and Text to Change

Here, we have three variables that store an article's title and content (`text_all`) from *Artificial Intelligence News*. Additionally, the `text_to_change` variable identifies the specific section of the text that we want to expand. These constants serve as reference points and will remain unchanged throughout the lesson.

```
title = "OpenAI Chief: AI Oversight 'Crucial' for Future Innovation"
text_all = """ Altman underscored the immense potential of AI advancements such as ChatGPT and I
```

```
text_to_change = """ Senators Josh Hawley and Richard Blumenthal acknowledged AI's disruptive in
```

We start by generating potential search queries from the paragraph we want to expand. These queries are then used to retrieve relevant documents from a search engine (such as Bing or Google Search), which are subsequently broken down into smaller chunks. Next, we compute embeddings for these chunks and store both the chunks and their embeddings in a Deep Lake dataset. Finally, the most relevant chunks are retrieved from Deep Lake based on their similarity to the original paragraph. These retrieved chunks are then incorporated into a prompt to enhance the paragraph with additional context and information.

Step 3: Define Your API Keys

```
# Set API keys (Use environment variables for security)
import os
os.environ["OPENAI_API_KEY"] = "Your API Key"
os.environ["GOOGLE_API_KEY"] = "Your API Key"
os.environ["GOOGLE_CSE_ID"] = "Your ID"
```

How to Get Your API Keys and ID

To use the Google Search API, we first need to set up an API key and a custom search engine. Start by navigating to the Google Cloud Console and creating a project, and then, enable the **Custom Search API** under **Enable APIs and Services** (Google will provide instructions if necessary). After that, generate an API key by clicking **CREATE CREDENTIALS** at the top and selecting **API KEY**.

Once these steps are complete, configure the environment variables "**GOOGLE_CSE_ID**" and "**GOOGLE_API_KEY**", allowing the Langchain Google wrapper to connect seamlessly with the API.

Next, go to the Programmable Search Engine dashboard: https://programmablesearchengine.google.com/controlpanel/create, create a custom search engine, and ensure that the "Search the entire web" option is selected. The search engine ID will be displayed in the Details section.

Go to the page with all search engines: https://programmablesearchengine.google.com/controlpanel/all, and click the one you have just created. Then, copy the Search engine ID.

Step 4: Generate Search Results

The following code leverages OpenAI's ChatGPT model to analyze an article and generate three relevant search queries. It begins by defining a prompt that instructs the model to suggest Google search queries for gathering more information on the topic. The "LLMChain" component connects the "ChatOpenAI" model with the "ChatPromptTemplate", forming a structured pipeline for interacting with the model.

Once the response is received, the code splits it by newline and removes the initial characters to extract the search queries. This approach works because the API was instructed to format each query as a new line starting with "-". (Alternatively, the same result can be achieved using the "OutputParser" class.)

Before executing the code, ensure that your OpenAI API key is stored in the “OPENAI_API_KEY” environment variable.

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain.prompts.chat import (
    ChatPromptTemplate,
    HumanMessagePromptTemplate,
)
template = """ You are an exceptional copywriter and content creator.
You're reading an article with the following title:
----------------
{title}
----------------
You've just read the following piece of text from that article.
----------------
{text_all}
----------------
Inside that text, there's the following TEXT TO CONSIDER that you want to enrich with new detai
----------------
{text_to_change}
----------------
What are some simple and high-level Google queries that you'd do to search for more info to add
Write 3 queries as a bullet point list, prepending each line with -.
"""
human_message_prompt = HumanMessagePromptTemplate(
    prompt=PromptTemplate(
        template=template,
        input_variables=["text_to_change", "text_all", "title"],
    )
)
chat_prompt_template = ChatPromptTemplate.from_messages([human_message_prompt])
# Before executing the following code, make sure to have
# your OpenAI key saved in the “OPENAI_API_KEY” environment variable.
chat = ChatOpenAI(model_name="gpt-4o-mini", temperature=0.5)
chain = LLMChain(llm=chat, prompt=chat_prompt_template)
response = chain.run({
    "text_to_change": text_to_change,
    "text_all": text_all,
    "title": title
})
response_queries = [line[2:] for line in response.split("\n")]
queries = [item.replace('"', "") for item in response_queries]
print(queries)
Output: ['impact of AI on elections and democracy 2023  ', 'AI voice cloning technology example
```

Step 5: Get Search Results

To use the Google Search API, we first need to set up an API key and a custom search engine. Start by navigating to the **Google Cloud Console**, and then, generate an API key by clicking **CREATE CREDENTIALS** at the top and selecting **API KEY**. Next, go to the **Programmable Search Engine** dashboard, and ensure that the **“Search the entire web”** option is enabled. The search engine ID will be displayed in the Details section.

Additionally, you may need to enable the **“Custom Search API”** under **Enable APIs and Services** (Google will provide further instructions if required). Once these steps are complete, configure the environment variables GOOGLE_CSE_ID and GOOGLE_API_KEY, enabling the Google wrapper to interact with the API.

The next step is to use the generated search queries from the previous section to retrieve relevant sources from Google. The **LangChain** library offers the GoogleSearchAPIWrapper, which handles search queries and retrieves results. To process the results efficiently, we define a function using the top_n_results parameter.

Then, the Tool class creates a wrapper around this function, making it compatible with AI agents so they can interact with external data sources. We request only the **top five search results** and then concatenate them for each query into the all_results variable for further processing.

```
from langchain.tools import Tool
from langchain.utilities import GoogleSearchAPIWrapper
# Initialize the Google Search API Wrapper
search = GoogleSearchAPIWrapper()
TOP_N_RESULTS = 5
def top_n_results(query):
    """Fetch top N search results for a given query."""
    results = search.results(query, TOP_N_RESULTS)
    if not results:
        return [{"Result": "No good Google Search Result was found"}]
    return results
# Define the search tool
search_tool = Tool(
    name="Google Search",
    description="Search Google for recent results.",
    func=top_n_results
)
# Sample queries list
queries = ['Senators Josh Hawley Richard Blumenthal AI regulation statements', 'impact of AI on
all_results = []
# Run search for each query
for query in queries:
    try:
        results = search_tool.run(query)
        all_results.extend(results)
    except Exception as e:
        all_results.append({"Error": str(e)})
# Print all collected search results
print(all_results)
```

The “all_results” variable may contain a different number of web addresses—derived from three search queries generated by ChatGPT, each returning the top five Google search results. However, using all retrieved content as context in our application is not an optimal approach due to technical, financial, and contextual constraints.

First, **language models (LLMs) have input length limitations**, typically ranging from **2K to 4K tokens**, depending on the model. While alternative chain types can help bypass this constraint, staying within the model’s token window is often more efficient and produces better results.

Second, **cost considerations** come into play. The more text we send to the API, the higher the cost. Although splitting prompts into multiple chains is an option, we must be mindful that API pricing is based on token usage, making excessive input size financially inefficient.

Finally, **contextual relevance matters**. The retrieved search results will likely contain overlapping or similar information. Instead of using all results indiscriminately, selecting the most relevant ones ensures a more focused and meaningful expansion of the content.

```
Output: [{'title': '[2023-09-08] Blumenthal & Hawley Announce Bipartisan Framework ...', 'link'
```

Step 6: Find the Most Relevant Results

As previously noted, Google Search provides URLs for each source, but we still need to extract the actual content from these pages. This is where the **newspaper** package comes in handy—it allows us to retrieve webpage content using the .parse() method. The following code iterates through the search results and attempts to extract the text from each linked page.

```
import newspaper
pages_content = []
for result in all_results:
  try:
    article = newspaper.Article(result["link"])
    article.download()
    article.parse()
    if len(article.text) > 0:
      pages_content.append({ "url": result["link"], "text": article.text })
  except:
    continue
print("Number of pages: ", len(pages_content))
Output: Number of pages:  11
```

Step 7: Split into Chunks

The output above indicates that only 11 pages were processed instead of the expected 15. This discrepancy can occur because the newspaper library may struggle to extract content in certain cases, such as when search results lead to PDF files or when websites impose restrictions on web scraping.

Next, it’s essential to split the extracted content into smaller chunks to prevent exceeding the model’s input length. The code below achieves this by segmenting the text based on either newlines or spaces, depending on the structure of the content. It ensures that each chunk contains 3000 characters with an overlap of 100 characters between consecutive chunks to maintain context.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.docstore.document import Document
text_splitter = RecursiveCharacterTextSplitter(chunk_size=3000, chunk_overlap=100)
docs = []
for d in pages_content:
    chunks = text_splitter.split_text(d["text"])
    for chunk in chunks:
        new_doc = Document(page_content=chunk, metadata={ "source": d["url"] })
        docs.append(new_doc)
print("Number of chunks: ", len(docs))
Output: Number of chunks:  26
```

Step 7: Create Embeddings

As shown, the docs variable now contains 26 chunks of data. The next step is to identify the most relevant chunks to use as context for the large language model. To achieve this, we leverage the OpenAIEmbeddings class, which utilizes OpenAI to convert text into a vector space that captures semantic meaning.

We then proceed to embed both the document chunks and the target sentence from the main article that we want to expand. This sentence, which was selected at the start of the lesson, is stored in the text_to_change variable. By comparing embeddings, we can retrieve the most relevant chunks to enrich the expanded content.

```
from langchain.embeddings import OpenAIEmbeddings
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")
docs_embeddings = embeddings.embed_documents([doc.page_content for doc in docs])
query_embedding = embeddings.embed_query(text_to_change)
```

To measure the relevance of document chunks, we use the **cosine similarity** metric, which calculates the distance between high-dimensional embedding vectors. This metric helps determine how closely two points are positioned within the vector space. Since embeddings capture contextual meaning, **closer vectors indicate stronger semantic similarity**, making high-scoring documents ideal sources for expansion.

We utilize the “cosine_similarity” function from the **sklearn** library to compute the similarity between each document chunk and the target sentence. This function returns the **indices of the top three most relevant chunks**, ensuring that the model receives the most meaningful context for generating expanded content.

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
def get_top_k_indices(list_of_doc_vectors, query_vector, top_k):
    # convert the lists of vectors to numpy arrays
    list_of_doc_vectors = np.array(list_of_doc_vectors)
    query_vector = np.array(query_vector)
    # compute cosine similarities
    similarities = cosine_similarity(query_vector.reshape(1, -1), list_of_doc_vectors).flatten(
    # sort the vectors based on cosine similarity
    sorted_indices = np.argsort(similarities)[::-1]
    # retrieve the top K indices from the sorted list
    top_k_indices = sorted_indices[:top_k]
    return top_k_indices
top_k = 3
best_indexes = get_top_k_indices(docs_embeddings, query_embedding, top_k)
best_k_documents = [doc for i, doc in enumerate(docs) if i in best_indexes]
```

Step 8: Extend the Sentence

Now, we can define the **prompt** using additional information retrieved from Google Search. The template includes six input variables:

- **`title`**: Holds the main article’s title
- **`text_all`**: Represents the full article being processed
- **`text_to_change`**: The specific section of the article that requires expansion
- **`doc_1`**, **`doc_2`**, **`doc_3`**: The top three most relevant Google search results, used as contextual references

The rest of the code follows the same structure as the **Google query generation process**. It defines a **`HumanMessage`** template, ensuring compatibility with the ChatGPT API. The model is set with a **high temperature** to promote creative output. Finally, the **`LLMChain`** class constructs a processing chain that integrates the model and the prompt, executing the expansion task using the `.run()` method.

```
template = """You are an exceptional copywriter and content creator.
You're reading an article with the following title:
----------------
{title}
----------------
You've just read the following piece of text from that article.
----------------
{text_all}
----------------
Inside that text, there's the following TEXT TO CONSIDER that you want to enrich with new detai
----------------
{text_to_change}
----------------
Searching around the web, you've found this ADDITIONAL INFORMATION from distinct articles.
----------------
{doc_1}
----------------
{doc_2}
----------------
{doc_3}
----------------
Modify the previous TEXT TO CONSIDER by enriching it with information from the previous ADDITIO
"""
human_message_prompt = HumanMessagePromptTemplate(
    prompt=PromptTemplate(
        template=template,
        input_variables=["text_to_change", "text_all", "title", "doc_1", "doc_2", "doc_3"],
    )
)
chat_prompt_template = ChatPromptTemplate.from_messages([human_message_prompt])
chat = ChatOpenAI(model_name="gpt-4o-mini", temperature=0.9)
chain = LLMChain(llm=chat, prompt=chat_prompt_template)
response = chain.run({
    "text_to_change": text_to_change,
    "text_all": text_all,
    "title": title,
    "doc_1": best_k_documents[0].page_content,
    "doc_2": best_k_documents[1].page_content,
    "doc_3": best_k_documents[2].page_content
})
print("Text to Change: ", text_to_change)
print("Expanded Variation:", response)
Output: Text to Change:   Senators Josh Hawley and Richard Blumenthal acknowledged AI's disrupt
Expanded Variation: Certainly! Here's an enriched version of the previously specified text, inc
---
Senators Josh Hawley (R-MO) and Richard Blumenthal (D-CT), Chair and Ranking Member of the Sena
```

App 6: YouTube Scriptwriting Tool

The YouTube Scriptwriting Tool is an AI-driven assistant designed to help content creators craft engaging, well-structured scripts for their videos. By leveraging GPT-powered AI, this tool streamlines the scriptwriting process, ensuring compelling storytelling, clear messaging, and audience engagement.

Step 1: Install All Required Libraries and Import Them

These dependencies are essential for building a YouTube Scriptwriting Tool with AI-powered transcription, content generation, and automation. Here's why each package is used:

- `openai`: Provides access to GPT models for generating YouTube video scripts, improving structure, and enhancing content
- `langchain`: A framework that integrates LLMs, text processing, and retrieval-based AI for better script structuring
- `google-colab`: Ensures compatibility with Google Colab, allowing the tool to run smoothly in a cloud environment
- `yt_dlp`: A powerful tool for downloading YouTube videos, enabling AI-based script generation by transcribing existing content
- `langchain-community`: Extends LangChain's capabilities with community-maintained integrations for improved AI workflows
- `openai-whisper`: A state-of-the-art AI model for speech-to-text transcription, used to convert YouTube videos into text-based scripts
- `torch`: A deep learning framework required for Whisper's AI model, enabling fast and efficient transcription

```
!pip install openai langchain google-colab yt_dlp langchain-community
!pip install -U openai-whisper torch
import openai
import os
import re
import subprocess
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from google.colab import auth
import whisper
```

Step 2: Authenticate in Google Drive As We Use Google Colab and Insert Your OpenAI API Key

```
auth.authenticate_user()
os.environ['OPENAI_API_KEY'] = input("Enter your OpenAI API Key: ")
gpt = ChatOpenAI(temperature=0.7, model_name="gpt-4")
```

Step 3: Download Your Desired YouTube Video, Extract the Audio, and Convert It to MP3

```
### Step 2: Download YouTube Audio
def download_audio(video_url):
    video_id_match = re.search(r"(?:v=|\/)([0-9A-Za-z_-]{11}).*", video_url)
    if not video_id_match:
        return None
    video_id = video_id_match.group(1)
    audio_filename = f"{video_id}.mp3"
    command = f"yt-dlp -x --audio-format mp3 -o '{audio_filename}' {video_url}"
    os.system(command)
    return audio_filename if os.path.exists(audio_filename) else None
```

Step 4: Transcribe Audio

```
### Step 3: Transcribe Audio
def transcribe_audio(audio_filename):
    model = whisper.load_model("small")
    result = model.transcribe(audio_filename)
    return result["text"]
```

This function **transcribes audio into text** using **OpenAI's Whisper model**.

1. **Loads Whisper's "small" model (`whisper.load_model("small")`)**
2. **Transcribes the given audio file (`model.transcribe(audio_filename)`)**
3. **Returns the extracted text (`result["text"]`)**

Step 5: Generate Outline

```
### Step 4: Generate an Outline
def generate_outline(transcript_text):
    outline_prompt = PromptTemplate(
        input_variables=["transcript_text"],
        template="""
        You are a professional YouTube scriptwriter. Analyze the following transcribed YouTube 
        "{transcript_text}"
        Create an engaging script outline, including an introduction, key sections, and a concl
        """
    )
    outline_chain = LLMChain(llm=gpt, prompt=outline_prompt)
    return outline_chain.run(transcript_text)
```

This function above generates a structured outline for a YouTube script from a transcribed video.

1. Defines a prompt template (`PromptTemplate`) that instructs the AI to analyze the transcript and create an outline with an introduction, key sections, and a conclusion
2. Creates an AI processing chain **(`LLMChain`) using `gpt` (a GPT model)**
3. Runs the AI model to generate an engaging script outline from the given transcript

Step 6: Expand the Script

```
### Step 5: Expand Script
def expand_script(outline):
    script_prompt = PromptTemplate(
        input_variables=["outline"],
        template="""
        Given the following YouTube script outline:
        {outline}
        Expand each section into a complete, engaging script with natural dialogue and a strong
        Include timestamps and suggested visuals where relevant.
        """
    )
    script_chain = LLMChain(llm=gpt, prompt=script_prompt)
    return script_chain.run(outline)
```

This function expands a script outline into a full YouTube script using AI.

1. Defines a prompt template (PromptTemplate) that instructs the AI to convert the outline into a detailed script, ensuring natural dialogue and strong narrative flow
2. Creates an AI processing chain (LLMChain) using gpt (a GPT model)
3. Runs the AI model to generate a fully developed script, including timestamps and suggested visuals for better content structuring

Step 7: Combine All and Run the Tool

```
### Step 6: Run the Tool
if __name__ == "__main__":
    video_url = input("Enter the YouTube video URL: ")
    print("\nDownloading audio from video...\n")
    audio_filename = download_audio(video_url)
    if not audio_filename:
        print("Audio download failed! Exiting.")
    else:
        print("Audio downloaded successfully!")
        print("\nTranscribing audio...\n")
        transcript_text = transcribe_audio(audio_filename)
        print("Transcript generated successfully!\n")
        print(transcript_text)
        print("\nGenerating script outline from transcript...\n")
        outline = generate_outline(transcript_text)
        print(outline)
        input("Press Enter to generate the full script...")
        print("\nExpanding into full script...\n")
        full_script = expand_script(outline)
        print(full_script)
```

Output:

```
Enter the YouTube video URL: https://www.youtube.com/shorts/9YFT5HqL5m8
Downloading audio from video...
Audio downloaded successfully!
Transcribing audio...
Transcript generated successfully!
 While loop in Python. Firstly write out the following lines of code, making sure you remember
Generating script outline from transcript...
Title: Mastering the While Loop in Python
Introduction:
- Welcoming viewers to the channel and the video
- Briefly discussing the importance of understanding Python loops, especially the "while loop"
- Outlining the objectives for the video
Section 1: Understanding the While Loop
- Explaining what a while loop is in the context of Python
- Discussing the use cases and benefits of using while loops
Section 2: Structuring the While Loop
- Explaining the syntax of the while loop, emphasizing the importance of colons and indents
- Showing on screen an example of the structure of a basic while loop
Section 3: Writing the Code
- Taking viewers through the process of writing a simple while loop code
- Highlighting key points such as the use of colons and indents, how to structure the loop, and
Section 4: Saving and Running the Code
- Demonstrating how to save and run the code
- Discussing potential errors that could occur and how to troubleshoot
Conclusion:
- Recapping the importance and structure of while loops in Python
- Encouraging viewers to practice writing their own while loops
- Reminding viewers to like, share, and subscribe for more Python tutorials
- Teasing the topic of the next video and bidding viewers farewell until next time.
Press Enter to generate the full script...
Expanding into full script...
Title: Mastering the While Loop in Python
[Introduction 00:00]
(Visual: Channel logo animation)
HOST: "Hey there coding enthusiasts, welcome back to our channel, your trusted guide to everyth
(Visual: Text Animation - "Mastering the While Loop in Python")
[Section 1: Understanding the While Loop 00:30]
(Visual: Video Animation - "While Loop Concept")
```

```
HOST: "So what exactly is a while loop? In Python, a while loop is used for iterative tasks, wh
[Section 2: Structuring the While Loop 01:15]
(Visual: Screen recording - Python IDE with blank code file)
HOST: "Now, let's talk about how we structure a while loop in Python. The syntax is straightforw
(Visual: Coding example on Python IDE)
[Section 3: Writing the Code 02:30]
(Visual: Screen Recording - Python IDE with code example)
HOST: "Let's write a simple while loop code together, shall we? Remember, our indents and colons
(Visual: Host typing and explaining the code)
[Section 4: Saving and Running the Code 04:50]
(Visual: Screen Recording - Python IDE)
HOST: "Once we've written our code, it's time to save and run it. But remember, errors can occur
(Visual: Demonstration of saving, running, and troubleshooting the code)
[Conclusion 06:40]
(Visual: Host on screen)
HOST: "And that, my friends, is the while loop in Python! Remember, practice is key, so try wri
(Visual: End screen with like, share, and subscribe animation)
```

App 7: Email Generator

The AI Email Generator is a powerful tool designed to automate and enhance email writing using AI. By leveraging GPT-powered language models, this tool helps users craft professional, personalized, and context-aware emails in seconds.

Key Features

- **Automated Email Drafting**: Generate emails based on prompts or key points.
- **Personalization**: Adjust tone, style, and recipient details for a tailored approach.
- **Quick Edits and Refinements**: Modify content instantly with AI suggestions.
- **Template-Based Generation**: Create emails for business, customer support, marketing, and more.
- **Grammar and Tone Enhancement**: Ensure clarity, professionalism, and engagement.

Ideal for professionals, businesses, and individuals, the AI Email Generator streamlines communication, saves time, and improves email effectiveness with AI-driven precision.

Step 1: Install All Required Libraries and Import Them

```
!pip install langchain openai langchain_community
from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
import os
```

These dependencies are essential for building an AI-powered Email Generator using LangChain and OpenAI. Here's why each package is needed:

- `langchain`: The core framework for integrating LLMs (like GPT-4) to generate, refine, and personalize email content
- `openai`: Provides access to GPT-powered AI for drafting professional, context-aware emails

- `langchain_community`: Enhances LangChain with community-supported integrations for better performance and extended capabilities

This setup enables smart, AI-driven email generation, making the process faster, more efficient, and highly personalized.

Step 2: Generate Response with OpenAI

```
def generate_email_response(api_key, original_email, sender_name, recipient_name, response_tone=
    os.environ["OPENAI_API_KEY"] = api_key
    template = PromptTemplate(
        input_variables=["original_email", "sender_name", "recipient_name", "response_tone"],
        template="""
        Read the following email from {sender_name} and generate a well-structured, contextually
        Ensure the tone of the response is {response_tone} and appropriately addresses the cont
        Original Email:
        {original_email}
        Keep the response concise yet informative, maintaining politeness and clarity.
        """,
    )
    llm = OpenAI(model="gpt-3.5-turbo")
    chain = LLMChain(llm=llm, prompt=template)
    response_email = chain.run({
        "original_email": original_email,
        "sender_name": sender_name,
        "recipient_name": recipient_name,
        "response_tone": response_tone
    })
    return response_email
```

In this code, the `generate_email_response()` function takes an API key, an original email, the sender and recipient names, and an optional response tone (defaulting to "professional").

It first sets the OpenAI API key as an environment variable (`os.environ["OPENAI_API_KEY"] = api_key`) to authenticate requests to OpenAI's API.

A `PromptTemplate` is then defined, guiding the AI to read the original email and generate a contextually relevant, well-structured response. The AI

- Adapts the tone (e.g., professional, friendly)
- Addresses the recipient appropriately
- Keeps the response concise, polite, and informative

An LLM model (`OpenAI()`) is initialized, and an LLMChain (`chain`) is created to process the prompt dynamically.

The function executes the chain with the given email details, generating an AI-written email response, which is then returned.

This setup saves time, enhances professionalism, and ensures clarity, making it useful for customer support, business communication, and automated email responses.

Step 3: Combine All Together and Generate Email

```
if __name__ == "__main__":
    api_key = input("Enter your OpenAI API key: ")
    original_email = input("Enter the original email content: ")
    sender_name = input("Enter the sender's name: ")
    recipient_name = input("Enter the recipient's name: ")
    response_tone = input("Enter the response tone (e.g., professional, friendly, casual): ")
    response = generate_email_response(api_key, original_email, sender_name, recipient_name, res
    print("\nGenerated Email Response:\n")
    print(response)
```

NoteDon’t forget to generate your OpenAI API key.

Output:

```
Enter your OpenAI API key:
Enter the original email content: Let's have a meeting together?
Enter the sender's name: Anthony
Enter the recipient's name: James
Enter the response tone (e.g., professional, friendly, casual): Professional
<ipython-input-5-279420fd8784>:30: LangChainDeprecationWarning: The class `OpenAI` was deprecate
  llm = OpenAI()
<ipython-input-5-279420fd8784>:31: LangChainDeprecationWarning: The class `LLMChain` was depreca
  chain = LLMChain(llm=llm, prompt=template)
<ipython-input-5-279420fd8784>:33: LangChainDeprecationWarning: The method `Chain.run` was depre
  response_email = chain.run({
Generated Email Response:
Dear Anthony,
Thank you for reaching out to me about having a meeting together. I am always open to discussing
Could you please provide more details about the meeting? This will help me prepare and make the
I look forward to meeting with you and discussing further.
Best regards,
James
```

App 8: CSV Data Analysis App

The CSV Data Analysis App is a powerful tool designed to help users efficiently analyze, visualize, and extract insights from structured datasets. By leveraging AI, data processing libraries, and interactive visualizations, this app makes it easy to explore large CSV files, perform statistical analysis, and generate meaningful reports.

Step 1: Install All Required Libraries and Import Them

```
!pip install pandas langchain openai matplotlib seaborn langchain_community langchain_experimen
import pandas as pd
import langchain
from langchain.llms import OpenAI
from langchain_experimental.agents import create_pandas_dataframe_agent
import matplotlib.pyplot as plt
import seaborn as sns
import os
```

These dependencies enable a CSV Data Analysis App by integrating AI, data processing, and visualization:

- **`pandas`**: Loads and manipulates CSV files.
- **`langchain` and `openai`**: Uses GPT-4 for AI-powered insights and queries
- **`matplotlib` and `seaborn`**: Creates professional data visualizations
- **`langchain_community` and `langchain_experimental`**: Enhances AI integration with modern tools

This setup allows users to analyze, visualize, and gain AI-driven insights from CSV datasets, making data exploration faster and smarter.

Step 2: Generate and Add Your OpenAI API Key

```
# Set API keys (Use environment variables for security)
import os
os.environ["OPENAI_API_KEY"] = <Your API Key>
```

Step 3: Load Your CSV File

This code below loads, previews, and analyzes a CSV file using pandas.

The `load_csv(file_path)` function takes a file path as input and loads the CSV file into a pandas DataFrame using `pd.read_csv()`.

It then prompts the user to enter the CSV file path, loads the data into `df`, and displays key insights:

1. Data Preview: Prints the first five rows of the dataset using `df.head()`, providing a quick look at the data structure
2. Basic Statistics: Prints summary statistics with `df.describe()`, showing key metrics like mean, min, max, and standard deviation for numerical columns

```
# Function to load CSV
def load_csv(file_path):
    return pd.read_csv(file_path)
# Load CSV
file_path = input("Enter the path to your CSV file: ")
df = load_csv(file_path)
# Display data preview
print("\n### Data Preview")
print(df.head())
# Display basic statistics
print("\n### Basic Statistics")
print(df.describe())
```

NoteTo get the path of a file in Google Colab, upload the file, right-click with your cursor, and select “Copy path.”

Output:

```
Enter the path to your CSV file: /content/langchain_broad-match_us_2025-02-19.csv
### Data Preview
                    Keyword         Intent  Volume  \
0              langchain js   Navigational    1000
1          langchain openai  Informational    1000
2           langchain tools  Informational    1000
3        langchain tutorial   Navigational    1000
4  langchain_community.llms  Informational    1000
                                               Trend  Keyword Difficulty  \
0  1.00,0.38,1.00,0.68,0.46,0.68,0.38,0.52,0.52,0...                  44
1  0.62,1.00,0.81,0.62,0.36,0.62,0.45,0.55,0.45,0...                  35
2  0.37,0.52,0.08,1.00,0.68,0.52,0.68,0.52,0.52,0...                  41
3  0.44,0.44,0.81,0.81,0.62,0.62,0.55,0.55,1.00,0...                  54
4  0.13,0.36,1.00,0.54,0.36,0.02,0.04,0.03,0.00,0...                  18
   CPC (USD)  Competitive Density  \
0       0.00                 0.01
```

```
1      3.63                 0.01
2      2.48                 0.00
3      2.98                 0.11
4      0.00                 0.00
                                          SERP Features  Number of Results
0  Sitelinks, Video, People also ask, Related sea...            6260000
1           Video, People also ask, Related searches           11600000
2                 Sitelinks, Video, Related searches           16400000
3  Sitelinks, Video, People also ask, Related sea...            8510000
4                                  Image pack, Video                 42
### Basic Statistics
            Volume  Keyword Difficulty   CPC (USD)  Competitive Density  \
count   284.000000          284.000000  284.000000           284.000000
mean    288.204225           28.419014    0.878275             0.019014
std     200.324685           12.485063    2.212721             0.069697
min     110.000000            0.000000    0.000000             0.000000
25%     140.000000           20.000000    0.000000             0.000000
50%     210.000000           27.000000    0.000000             0.000000
75%     320.000000           36.000000    0.000000             0.010000
max    1000.000000           83.000000   17.340000             0.830000
       Number of Results
count       2.840000e+02
mean        3.444161e+06
std         5.098099e+06
min         0.000000e+00
25%         9.700000e+01
50%         5.330000e+05
75%         5.745000e+06
max         3.630000e+07
```

Step 4: Create a LangChain Agent

```
# LangChain Agent for querying data
OPENAI_API_KEY = os.environ["OPENAI_API_KEY"]
# LangChain Agent for querying data
if OPENAI_API_KEY:
    llm = OpenAI(temperature=0, openai_api_key=OPENAI_API_KEY)
    agent = create_pandas_dataframe_agent(llm, df, verbose=True, allow_dangerous_code=True)
    query = input("\nAsk a question about the data: ")
    if query:
        print("\nAnalyzing...")
        response = agent.run(query)
        print("\n**Response:**", response)
else:
    print("\nWarning: OpenAI API Key not found. Please set it as an environment variable.")
```

The code above sets up a LangChain agent to interact with a CSV dataset using GPT-powered AI queries.

First, it retrieves the OpenAI API key from the environment (`os.environ["OPENAI_API_KEY"]`). If the key exists, it initializes an LLM instance (`OpenAI`) with `temperature=0` for deterministic responses.

It then creates a Pandas DataFrame agent using `create_pandas_dataframe_agent(llm, df, verbose=True, allow_dangerous_code=True)`. This agent allows users to ask natural language questions about the dataset, and the AI will analyze and generate insights based on the data.

The program then prompts the user for a query. If a question is provided, the agent processes the request, runs the query on the DataFrame, and returns an AI-generated response.

If the API key is missing, it prints a warning message, instructing the user to set up the key.

This setup enables AI-powered data analysis, allowing users to interact with CSV datasets using natural language instead of manual coding.

Output:

```
Ask a question about the data: What's the data about?
Analyzing...
> Entering new AgentExecutor chain...
<ipython-input-16-9aca66b979d8>:11: LangChainDeprecationWarning: The method `Chain.run` was dep
  response = agent.run(query)
Thought: The data is about keywords and their corresponding attributes.
Action: python_repl_ast
Action Input: df.info()<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284 entries, 0 to 283
Data columns (total 9 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Keyword              284 non-null    object
 1   Intent               284 non-null    object
 2   Volume               284 non-null    int64
 3   Trend                284 non-null    object
 4   Keyword Difficulty   284 non-null    int64
 5   CPC (USD)            284 non-null    float64
 6   Competitive Density  284 non-null    float64
 7   SERP Features        284 non-null    object
 8   Number of Results    284 non-null    int64
dtypes: float64(2), int64(3), object(4)
memory usage: 20.1+ KB
 The data has 284 rows and 9 columns.
Action: python_repl_ast
Action Input: df.shape(284, 9)I now know the final answer
Final Answer: The data has 284 rows and 9 columns. It contains information about keywords, thei
> Finished chain.
```

App 9: Knowledge Base Voice Assistant

The Knowledge Base Voice Assistant is an AI-driven system that enables users to interact with a knowledge base using natural voice commands. By integrating speech recognition, large language models (LLMs), and vector search, this assistant allows for seamless and intelligent access to vast amounts of information.

Designed for businesses, research teams, and customer support, this voice-enabled assistant can retrieve answers, summarize documents, and provide real-time insights from structured and unstructured data sources. By leveraging LangChain, OpenAI's GPT models, and vector databases, the assistant delivers accurate and context-aware responses in a conversational format.

Step 1: Install the Required Libraries and Import Them

```
# Install dependencies
!pip install SpeechRecognition gtts langchain faiss-cpu openai
import speech_recognition as sr
from gtts import gTTS
```

```
import os
import IPython.display as ipd
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.chat_models import ChatOpenAI
from langchain.chains import RetrievalQA
from langchain.document_loaders import WebBaseLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import FAISS
from google.colab import files
```

These dependencies enable a **voice-controlled AI assistant** by integrating **speech recognition, retrieval, and AI-powered responses**:

- **SpeechRecognition**: Converts speech to text for voice input
- **gTTS**: Converts AI-generated text to speech for voice output
- **LangChain**: Manages LLM interactions and knowledge retrieval
- **FAISS**: Enables fast, semantic search in the knowledge base
- **OpenAI**: Uses GPT-4 to generate intelligent responses

Together, these tools allow users to **speak queries, retrieve relevant information, and hear AI-generated answers**, making knowledge access seamless and intuitive.

Step 2: Generate and Add Your OpenAI API Key

```
# Set API keys (Use environment variables for security)
import os
os.environ["OPENAI_API_KEY"] = <Your API Key>
```

Step 3: Develop Voice Interaction

```
def speak(text):
    """Convert text to speech using gTTS and play it."""
    tts = gTTS(text=text, lang='en')
    tts.save("response.mp3")
    ipd.display(ipd.Audio("response.mp3"))
def listen():
    """Process uploaded audio file and convert to text."""
    recognizer = sr.Recognizer()
    print("Please upload an audio file (wav format).")
    uploaded = files.upload()
    for filename in uploaded.keys():
        with sr.AudioFile(filename) as source:
            audio = recognizer.record(source)
        try:
            return recognizer.recognize_google(audio)
        except sr.UnknownValueError:
            return "Sorry, I could not understand."
        except sr.RequestError:
            return "Could not request results."
```

This code enables voice interaction for an AI assistant by handling both text-to-speech (TTS) output and speech-to-text (STT) input using gTTS and SpeechRecognition.

The `speak(text)` function takes a text input, converts it into speech using gTTS (Google Text-to-Speech), and saves the generated audio as `"response.mp3"`. It then plays the audio using IPython's audio player (`ipd.Audio`), allowing users to hear the assistant's response.

The `listen()` function processes an uploaded audio file (in `.wav` format) and converts speech into text. It uses SpeechRecognition's Recognizer to handle transcription. **The user is prompted to upload an audio file, which is then processed:**

1. Loads the uploaded file
2. Extracts audio data using `sr.AudioFile()`
3. Recognizes speech using Google's speech-to-text API (`recognize_google`)
4. Returns the transcribed text or an error message if speech is unclear (`UnknownValueError`) or if the API request fails (`RequestError`)

This setup allows the assistant to listen to user queries, process them as text, and respond with AI-generated speech, enabling a full voice-based knowledge assistant experience.

NoteSince we develop this app in Google Colab, this platform doesn't provide us with microphone access, so all audio interaction has to be recorded as **.wav files and uploaded to Google Colab.**

Step 4: Load Knowledge Base from the Web and Create the QA Chain

```
# Load knowledge base from the web
def load_knowledge_base():
    """Load and process online resources for retrieval."""
    urls = [
        "https://en.wikipedia.org/wiki/Artificial_intelligence",
        "https://en.wikipedia.org/wiki/Natural_language_processing"
    ]
    loader = WebBaseLoader(urls)
    documents = loader.load()
    text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
    texts = text_splitter.split_documents(documents)
    embeddings = OpenAIEmbeddings()
    vectorstore = FAISS.from_documents(texts, embeddings)
    return vectorstore
def create_qa_chain(vectorstore):
    """Set up LangChain's RetrievalQA model."""
    llm = ChatOpenAI()
    retriever = vectorstore.as_retriever()
    return RetrievalQA.from_chain_type(llm=llm, retriever=retriever)
```

The code above fetches knowledge from the Web, processes it into a retrievable format, and sets up an AI-powered Q&A system using LangChain.

The `load_knowledge_base()` function

1. Defines a list of URLs containing knowledge (Wikipedia pages on AI and NLP)
2. Uses `WebBaseLoader` to fetch and extract the content from these web pages
3. Splits the extracted text into chunks of 1000 characters, ensuring 200-character overlap for better context retention using `CharacterTextSplitter`
4. Converts these text chunks into vector embeddings using `OpenAIEmbeddings`, allowing semantic search

5. Stores the processed embeddings in a FAISS vector database, which enables efficient retrieval of relevant knowledge

The `create_qa_chain(vectorstore)` function

1. Initializes an LLM-powered chatbot using `ChatOpenAI()`
2. Converts the FAISS vector store into a retriever, allowing the AI to find relevant information from stored knowledge
3. Creates a retrieval-based question-answering (QA) system using `RetrievalQA.from_chain_type()`, enabling users to ask natural language questions and get context-aware answers

This setup allows an AI assistant to retrieve and answer questions based on web-sourced knowledge, making it useful for automated research assistants, chatbots, and real-time information retrieval systems.

Step 5: Combine Them All Together

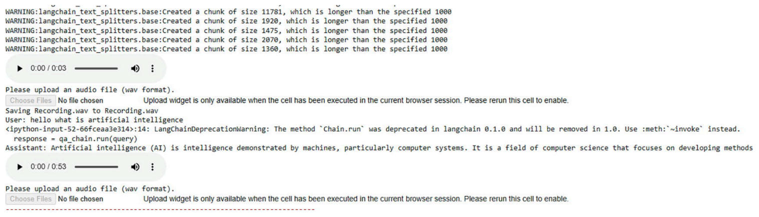
```
def main():
    """Main loop for voice interaction."""
    vectorstore = load_knowledge_base()
    qa_chain = create_qa_chain(vectorstore)
    speak("Hello! Please upload an audio file with your query.")
    while True:
        query = listen()
        if query.lower() in ["exit", "quit", "stop"]:
            speak("Goodbye!")
            break
        print(f"User: {query}")
        response = qa_chain.run(query)
        print(f"Assistant: {response}")
        speak(response)
```

This code creates a voice-interactive AI assistant that retrieves information from a web-based knowledge base and responds using speech.

The `main()` function

1. Loads the knowledge base by calling `load_knowledge_base()`, which fetches and processes online content into a retrievable format
2. Creates a Q&A system using `create_qa_chain(vectorstore)`, allowing AI-driven responses based on stored knowledge
3. Welcomes the user with speech using `speak("Hello! Please upload an audio file with your query.")`, prompting them to submit a voice query
4. Enters a loop where
   1. It listens for user input via `listen()`, which converts speech into text
   2. If the user says `"exit"`, `"quit"`, or `"stop"`, the assistant ends the conversation with a goodbye message
   3. Otherwise, it retrieves and generates an AI-powered response using `qa_chain.run(query)`, prints it, and speaks the response aloud using `speak(response)`

Output—if all works correctly:



```
WARNING:langchain_text_splitters.base:Created a chunk of size 11781, which is longer than the specified 1000
WARNING:langchain_text_splitters.base:Created a chunk of size 1920, which is longer than the specified 1000
WARNING:langchain_text_splitters.base:Created a chunk of size 1475, which is longer than the specified 1000
WARNING:langchain_text_splitters.base:Created a chunk of size 2070, which is longer than the specified 1000
WARNING:langchain_text_splitters.base:Created a chunk of size 1360, which is longer than the specified 1000
0:00 / 0:03
Please upload an audio file (wav format).
Choose Files No file chosen        Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving Recording.wav to Recording.wav
User: hello what is artificial intelligence
<ipython-input-52-66fceaa3e314>:14: LangChainDeprecationWarning: The method `Chain.run` was deprecated in langchain 0.1.0 and will be removed in 1.0. Use :meth:`~invoke` instead.
  response = qa_chain.run(query)
Assistant: Artificial intelligence (AI) is intelligence demonstrated by machines, particularly computer systems. It is a field of computer science that focuses on developing methods
0:00 / 0:53
Please upload an audio file (wav format).
Choose Files No file chosen        Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
```

Figure 3-1 Voice Assistant Output

App 10: Analyzing Codebase with LangChain

The Analyzing Codebase with LangChain app is an AI-powered tool designed to help developers, engineers, and teams efficiently explore and understand complex code bases. By leveraging LangChain's advanced language processing capabilities, the app can extract insights, answer questions, and provide recommendations based on the structure and logic of a given code repository.

Using large language models (LLMs) like GPT-4, along with vector search and semantic retrieval, this app enables users to quickly navigate source code, identify dependencies, summarize functions, and even detect potential issues—all without manually scanning through thousands of lines of code.

Step 1: Install All Required Libraries

```
!pip install langchain openai chromadb tiktoken
!pip install -U langchain-community
!pip install unstructured
```

These installation commands ensure that all necessary dependencies are available for building an AI-powered code base analysis tool using LangChain. Here's why each package is needed:

- **`langchain`**: The core framework that enables interaction with large language models (LLMs), vector databases, and advanced AI tools for processing and analyzing code.
- **`openai`**: Provides access to OpenAI's models (like GPT-4), which can generate insights, summarize code, and answer questions intelligently.
- **`chromadb`**: A vector database used for efficient storage and retrieval of embeddings. This is crucial for semantic search, allowing the AI to find relevant code snippets quickly.
- **`tiktoken`**: A tokenizer for OpenAI models that helps efficiently count and manage tokens, ensuring that the AI processes code efficiently while staying within model constraints.
- **`langchain-community`**: The updated package containing community-supported integrations for third-party tools like ChromaDB, OpenAI, FAISS, and more. Keeping this updated ensures compatibility with the latest LangChain features.
- **`unstructured`**: A powerful library for extracting and processing text from complex files and documents, including code files, PDFs, and markdowns. This helps in parsing, cleaning, and structuring raw code data before embedding it into a vector database.
- As an alternative, you can use Docling. It is an innovative document processing tool developed by xAI, designed to streamline the extraction and analysis of information from various file formats like PDFs, images, and text documents. It leverages advanced AI tech-

niques to enable users to quickly interpret and interact with complex documents, making it a valuable asset for research, data analysis, and knowledge management.

Step 2: Generate and Add Your OpenAI API Key

```
# Set API keys (Use environment variables for security)
import os
os.environ["OPENAI_API_KEY"] = <Your API Key>
```

Step 3: Upload and Load the Files

The next lines of code **scan a directory** for Python (`.py`) files, **reads their content**, and **stores them in a structured format** for further processing, such as embedding or AI-powered analysis.

The `load_code_files` function does the following:

1. Uses the `glob` module to **find all Python files** (`.py`) within the specified directory (`"./my_codebase"`) and its subdirectories (`recursive=True`)
2. Initializes an empty list called `documents` to store the extracted code
3. Iterates over each Python file found:
   1. Opens the file in **read mode** with UTF-8 encoding
   2. Reads the entire content of the file
   3. Appends a dictionary containing the file's **path** and **content** to the `documents` list
4. Returns the `documents` list, which now contains the **path and source code** of each Python file

The **example usage** calls `load_code_files()`, storing the result in `documents`. It then prints the total number of Python files loaded.

```
import glob
def load_code_files(directory="./my_codebase"):
    code_files = glob.glob(f"{directory}/**/*.py", recursive=True)
    documents = []
    for file_path in code_files:
        with open(file_path, "r", encoding="utf-8") as file:
            documents.append({"path": file_path, "content": file.read()})
    return documents
# Example Usage
documents = load_code_files()
print(f"Loaded {len(documents)} code files.")
```

Step 4: Create and Store Code Embeddings

This code **automates the process of loading, processing, and embedding a code base** for efficient search and retrieval using **LangChain and OpenAI embeddings**.

First, it uses **`DirectoryLoader`** to scan the `./my_codebase` directory (feel free to change it according to your needs) for all Python files (`**/*.py`) (you can look for the file extension according to your programming language). It loads these files into memory as **documents**, displaying a progress indicator during the process.

Next, the `RecursiveCharacterTextSplitter` is used to **split the loaded code into smaller chunks** of 500 characters, with a 50-character overlap between chunks. This ensures that when the AI retrieves and processes code, it maintains context across split sections.

After splitting the code, **embeddings** are generated using OpenAI's `OpenAIEmbeddings`, which converts each chunk into a **vector representation**. These embeddings allow for **semantic search**, meaning the AI can find relevant code snippets based on meaning rather than just exact keyword matches.

Finally, these embeddings are stored in a **Chroma vector database** using `Chroma.from_documents(docs, embedding_model)`, making the code searchable and retrievable based on AI-powered similarity searches.

This setup enables **AI-powered code search, understanding, and analysis**, making it useful for **automated documentation, intelligent code retrieval, and AI-assisted debugging**.

```
from langchain.document_loaders import DirectoryLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import Chroma
# Load code files
loader = DirectoryLoader("./my_codebase", glob="**/*.py", show_progress=True)
documents = loader.load()
# Split code into chunks
splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
docs = splitter.split_documents(documents)
# Generate embeddings
embedding_model = OpenAIEmbeddings()
vectorstore = Chroma.from_documents(docs, embedding_model)
print("Code embeddings stored successfully!")
```

Step 5: Create Retriever and Retrieval Chain

```
from langchain.chains import create_retrieval_chain
from langchain.chat_models import ChatOpenAI
from langchain.schema.runnable import RunnablePassthrough
from langchain.prompts import ChatPromptTemplate
# Setup LLM
llm = ChatOpenAI(model="gpt-4", temperature=0)
# Create retriever
retriever = vectorstore.as_retriever(search_kwargs={"k": 5})
# Define the prompt template
prompt = ChatPromptTemplate.from_template(
    "You are an AI code assistant. Explain the following query based on the code context:\n\nQu
)
# Create retrieval chain
qa_chain = create_retrieval_chain(retriever, prompt | llm | RunnablePassthrough())
# Function to Query the Codebase
def query_codebase(query):
    result = qa_chain.invoke({"input": query})  # Run the query
    print("\n Full Response:", result)  # Debug: print full output
    return result['answer']
# Example Usage
query = "How does the class Tomorrow work?"
response = query_codebase(query)
print("\n Search Result:\n", response)
```

This code **sets up an AI-powered code retrieval and explanation system** using **LangChain**, allowing users to ask questions about a code base and receive intelligent, context-aware responses.

First, it initializes a **GPT-4 model** using `ChatOpenAI` with `temperature=0`, ensuring **deterministic responses** for accurate code explanations.

Next, a **retriever** is created from the `vectorstore`, configured to return the **top five most relevant code snippets** (`k=5`) when queried. This ensures that only the most relevant parts of the code base are retrieved for explanation.

A **prompt template** is then defined using `ChatPromptTemplate.from_template()`, instructing the AI to act as a **code assistant**. It dynamically inserts

- **The user query** (`{input}`)
- **The retrieved code context** (`{context}`)

This ensures that responses are directly based on the actual code base.

The **retrieval chain** (`qa_chain`) is then created using `create_retrieval_chain()`. It follows a structured pipeline:

1. **Retrieve relevant code snippets** (`retriever`)
2. **Format the query and retrieved context into the prompt** (`prompt`)
3. **Generate an AI-powered explanation using GPT-4** (`llm`)
4. **Pass through the final response** (`RunnablePassthrough()`)

The `query_codebase` function allows users to input a **natural language question about the code base**. It runs the `qa_chain`, processes the query, and returns an AI-generated response. For debugging, it also prints the **full response object**.

Finally, an **example query** is run:

- The user asks **"How does the class Tomorrow work?" (replace it with your own query)**.
- The system searches the **vectorized code base** for relevant code snippets.
- GPT-4 generates an explanation based on the retrieved context.
- The response is printed, providing a clear AI-generated answer about the code.

This setup enables **AI-powered code analysis**, making it useful for **developer assistance, code documentation, debugging, and understanding large code bases**.

Output:

```
Search Result:
 content="The `Tomorrow` class is designed to handle asynchronous tasks in Python. It uses the `
```

App 11: Recommender System with LangChain

A **Recommender System with LangChain** is an AI-powered system that suggests relevant content, products, or information by leveraging

LangChain’s capabilities in natural language processing, vector databases, and large language models (LLMs). It integrates **retrieval-augmented generation (RAG)** techniques, semantic search, and embeddings to provide intelligent and personalized recommendations.

How It Works

1. **Data Ingestion and Processing**: The system processes and structures input data, which could be product descriptions, research papers, articles, or user preferences.
2. **Text Embedding and Vector Storage**: Text data is converted into embeddings using models like `OpenAIEmbeddings`. The embeddings are stored in a **vector database** like FAISS, Pinecone, or ChromaDB.
3. **Retrieval Based on Similarity**: When a user queries the system, their input is also converted into an embedding and compared against stored embeddings to find the most relevant matches.
4. **LLM-Enhanced Recommendations**: A language model (such as GPT-4) can refine and explain the recommendations by generating context-aware suggestions based on retrieved results.
5. **Personalization and Context Memory**: By integrating **memory mechanisms** like `ConversationBufferMemory()`, the system can refine recommendations based on user preferences and past interactions.

Step 1: Install and Import the Required Libraries

```
!pip install langchain openai==0.28 faiss-cpu tiktoken
!pip install -U langchain-community
```

These installation commands ensure that all necessary dependencies are available for building a **LangChain-based AI system**, such as a **PDF chatbot or a recommender system**. Here’s why each package is used:

- `langchain`: The core framework for integrating large language models (LLMs), vector databases, and retrieval-based AI applications.
- `openai==0.28`: Installs version 0.28 of the OpenAI Python package, which allows interaction with GPT-4, GPT-3.5, and embeddings models. Using a specific version ensures compatibility with LangChain.
- `faiss-cpu`: A vector database optimized for fast similarity search, used to store and retrieve text embeddings efficiently.
- `tiktoken`: A tokenizer for OpenAI models that helps optimize token usage and ensures the chatbot stays within token limits.
- `langchain-community`: The updated package for community-supported integrations, replacing older LangChain modules for better maintenance and compatibility with third-party tools like FAISS, Pinecone, and OpenAI.

Step 2: Generate and Add Your OpenAI API Key and Then Import All Libraries Required

```
from langchain.schema import Document
from langchain.vectorstores import FAISS
from langchain.chains import RetrievalQA
from langchain.memory import ConversationBufferMemory
from langchain.embeddings import OpenAIEmbeddings
from langchain.chat_models import ChatOpenAI
from langchain.tools import Tool
from langchain.agents import initialize_agent
# Set API keys (Use environment variables for security)
import os
os.environ["OPENAI_API_KEY"] = <Your API Key>
```

Step 3: Load Up Some Sample Data

```
# Sample data (list of items to recommend)
data = [
    "The Lord of the Rings - A fantasy novel by J.R.R. Tolkien.",
    "Harry Potter - A young wizard's journey by J.K. Rowling.",
    "The Matrix - A sci-fi movie about a simulated reality.",
    "Inception - A movie about dreams within dreams.",
    "The Witcher - A fantasy book and TV series about a monster hunter.",
    "Game of Thrones - A TV series based on A Song of Ice and Fire.",
    "Interstellar - A sci-fi movie about space exploration.",
    "Dune - A sci-fi novel by Frank Herbert about interstellar politics.",
    "Blade Runner - A dystopian sci-fi movie exploring artificial intelligence.",
    "Neuromancer - A cyberpunk novel by William Gibson about hackers and AI."
]
```

Step 4: Convert Data into LangChain Document Format and Generate Embeddings

```
# Convert data into LangChain Document format
documents = [Document(page_content=item) for item in data]
# Generate Embeddings
embeddings = OpenAIEmbeddings()
vector_store = FAISS.from_documents(documents, embeddings)
```

The code above processes raw text data, converts it into LangChain's **Document** format, and then generates vector embeddings to store in a **FAISS database** for efficient retrieval.

The first line transforms each item in the `data` list into a **LangChain Document** object, which is a standardized format for handling text in LangChain-based applications. This is necessary because LangChain's retrieval mechanisms expect data to be in this structured format.

Next, an instance of **OpenAIEmbeddings** is created. This model converts text into **numerical vector representations** (embeddings), which enable **semantic search**—meaning the system can find similar documents based on meaning rather than just keywords.

Finally, a **FAISS vector store** is created from the processed documents using their embeddings. FAISS (Facebook AI Similarity Search) is a vector database optimized for fast and efficient similarity search, allowing quick retrieval of relevant information from large datasets.

Step 5: Define an Advanced Retrieval Function

In the following, we define an **advanced retrieval function** that finds the most relevant documents based on a user's query using **vector similarity search**.

The function `get_advanced_recommendations` takes three parameters:

1. **`query`**: The user's input or search phrase
2. **`k`**: The number of top matching results to return (default is 3)
3. **`return_scores`**: A boolean flag indicating whether to return similarity scores alongside the recommendations

The function calls `vector_store.similarity_search_with_score(query, k=k)`, which searches the FAISS vector store for the `k` most similar documents based on their embeddings. The results include both the retrieved document objects and their similarity scores.

If `return_scores` is `True`, the function returns a list of tuples containing both the document content and its similarity score. Otherwise, it returns only the document content without scores.

This approach enables **semantic search and recommendation generation**, making it useful for applications like **chatbots, document retrieval systems, and AI-powered recommendation engines** that need to find the most contextually relevant information.

```
# Define an advanced retrieval function
def get_advanced_recommendations(query, k=3, return_scores=False):
    """Returns the top-k most similar items based on user query, optionally with similarity scor
    results_with_scores = vector_store.similarity_search_with_score(query, k=k)
    if return_scores:
        return [(doc.page_content, score) for doc, score in results_with_scores]
    else:
        return [doc.page_content for doc, _ in results_with_scores]
```

Step 6: Integrate a QA System Using LangChain

Then, it's time to set up a **question-answering (QA) system** using LangChain by integrating a **retrieval-based approach** with an **LLM (GPT-4)**.

The `RetrievalQA.from_chain_type` function creates a **QA pipeline** that retrieves relevant information from a **vector store** before generating answers. It takes three key parameters:

1. **`llm=ChatOpenAI(model="gpt-4")`**: Uses OpenAI's **GPT-4** model to process and generate responses.
2. **`retriever=vector_store.as_retriever()`**: Converts the **FAISS vector store** into a retriever that can find the most relevant documents based on user queries.
3. **`chain_type="stuff"`**: Specifies the document processing method. The `"stuff"` method takes the retrieved documents, combines their content, and passes them directly to the LLM for response generation.

This setup enables **semantic search-based question answering**, where the system retrieves the most relevant documents from the vector store and leverages GPT-4 to **generate accurate, context-aware answers**. It's useful for **chatbots, document search engines, and AI assistants** that need to provide precise responses based on stored knowledge.

```
# Integrate a QA system using LangChain
```

```
qa_chain = RetrievalQA.from_chain_type(
    llm=ChatOpenAI(model="gpt-4"),
    retriever=vector_store.as_retriever(),
    chain_type="stuff"
)
```

Step 7: Set Up an AI Conversational Agent

The code below continues by setting up an **AI-powered conversational agent** using LangChain, integrating both a **question-answering (QA) system** and a **recommendation system** with conversational memory.

The `answer_query` function takes a user's input and retrieves an AI-generated answer using the `qa_chain.run(query)` method. This ensures responses are based on relevant retrieved information.

A **conversational memory** is implemented using `ConversationBufferMemory(memory_key="chat_history")`. This allows the chatbot to remember previous interactions, improving the coherence of multiturn conversations.

Two tools are defined using the `Tool` class:

1. **`recommendation_tool`**: Calls `get_advanced_recommendations()` to retrieve the **top five most relevant recommendations** based on a query, including similarity scores
2. **`qa_tool`**: Calls `answer_query()` to generate **AI-powered answers** based on retrieved documents

The **LangChain Agent** is then initialized using `initialize_agent()`:

- It includes the **QA and recommendation tools**.
- Uses **GPT-4** as the language model (`llm=ChatOpenAI(model="gpt-4")`).
- Implements a **zero-shot-react-description** agent, meaning the AI can reason and select the best tool dynamically.
- Maintains a conversation history using `memory`.
- Runs in **verbose mode**, providing detailed execution logs for debugging.

This setup enables the **AI agent to act as an intelligent assistant**, capable of both answering questions and providing recommendations in an interactive and memory-enhanced conversation.

```
# Function to answer user queries intelligently
def answer_query(query):
    """Returns an AI-generated answer based on retrieved information."""
    return qa_chain.run(query)
# Implement Conversational Memory
memory = ConversationBufferMemory(memory_key="chat_history")
# Define tools for the LangChain agent
recommendation_tool = Tool(
    name="Recommendation System",
    func=lambda query: get_advanced_recommendations(query, k=5, return_scores=True),
    description="Provides top recommendations based on a user query."
)
qa_tool = Tool(
    name="QA System",
    func=answer_query,
    description="Answers questions using an AI-powered retrieval system."
)
# Initialize LangChain Agent
agent = initialize_agent(
    tools=[recommendation_tool, qa_tool],
    llm=ChatOpenAI(model="gpt-4"),
    agent="zero-shot-react-description",
    memory=memory,
    verbose=True
)
```

Step 8: Test the System

```
# Example Queries
query = "I love sci-fi movies about space."
recommendations = get_advanced_recommendations(query, k=5, return_scores=True)
print("Top Recommendations with Scores:")
for rec, score in recommendations:
    print(f"- {rec} (Score: {score:.4f})")
# Intelligent QA System Example
query_qa = "What are some movies about AI?"
answer = answer_query(query_qa)
print("\nAI-Powered Answer:")
print(answer)
# Interactive Agent Example
user_input = "Give me a recommendation for fantasy books."
agent_response = agent.run(user_input)
print("\nAgent Response:")
print(agent_response)
op Recommendations with Scores:
- Interstellar - A sci-fi movie about space exploration. (Score: 0.2047)
- The Matrix - A sci-fi movie about a simulated reality. (Score: 0.3131)
- Blade Runner - A dystopian sci-fi movie exploring artificial intelligence. (Score: 0.3341)
- Dune - A sci-fi novel by Frank Herbert about interstellar politics. (Score: 0.3552)
- Inception - A movie about dreams within dreams. (Score: 0.3911)
AI-Powered Answer:
Some movies about artificial intelligence include "Blade Runner" and "The Matrix".
> Entering new AgentExecutor chain...
The user is asking for a recommendation, not a factual answer.
Action: Recommendation System
Action Input: Fantasy books
Observation: [('The Lord of the Rings - A fantasy novel by J.R.R. Tolkien.', 0.2408208), ('The L
Thought:The recommendation system has provided a list of fantasy books. There's no need to go fu
Final Answer: Here are some fantasy books you might enjoy: 'The Lord of the Rings' by J.R.R. To
> Finished chain.
Agent Response:
Here are some fantasy books you might enjoy: 'The Lord of the Rings' by J.R.R. Tolkien, 'The Wi
```

App 12: PDF Files Chatbot

A **PDF Chatbot with LangChain** is an AI-powered assistant designed to interact with and extract insights from PDF documents. Using **LangChain**, a framework for building applications with large language models (LLMs), the chatbot can read, process, and answer questions based on the content of uploaded PDFs. This enables users to efficiently search for specific information, summarize sections, or analyze documents without manually going through large amounts of text.

Typically, a PDF chatbot integrates **text extraction tools** (like PyMuPDF or PDFMiner), **vector databases** for semantic search, and **LLM-powered reasoning** to provide accurate responses. This makes it useful for legal documents, research papers, contracts, and reports, improving workflow automation and knowledge retrieval.

Step 1: Install All Required Libraries

```
!pip install langchain pypdf faiss-cpu openai tiktoken
!pip install -U langchain-community
```

- `langchain`: The core framework that enables interaction with large language models (LLMs), document loading, vector databases, and reasoning capabilities.
- `pypdf`: A Python library for extracting text from PDF files, allowing the chatbot to read and process document content.
- `faiss-cpu`: A vector database library developed by Facebook AI for fast and efficient similarity search, crucial for storing and retrieving document embeddings.
- `openai`: Provides access to OpenAI's LLMs (like GPT-4) for natural language understanding and generating responses.
- `tiktoken`: A tokenizer used for counting tokens efficiently when working with OpenAI models, helping in cost estimation and ensuring token limits are managed properly.
- **langchain-community**: An updated version of LangChain's community-supported integrations. It includes various third-party tool integrations (like OpenAI, FAISS, Pinecone, and more) for better support and maintenance.

  Then, import them:

```
import os
import faiss
import pickle
import time
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import FAISS
from langchain.chains import ConversationalRetrievalChain
from langchain.chat_models import ChatOpenAI
from langchain.memory import ConversationBufferMemory
```

Step 2: Generate and Add Your OpenAI API Key

```
# Set API keys (Use environment variables for security)
import os
os.environ["OPENAI_API_KEY"] = <Your API Key>
```

Step 3: Upload Your PDF Files, Access Them, and Create a Vector Store Database

The following code sets up a system to process PDFs, extract text, split it into chunks, and store the processed data in a FAISS vector database for efficient retrieval. It first defines a directory named "**vector_store**" where the FAISS database will be stored.

The "**load_pdfs**" function takes a list of PDF file paths, extracts text from each file using "**PyPDFLoader**", and compiles all extracted text into a list.

It then uses “**RecursiveCharacterTextSplitter**” to break the text into smaller chunks of 500 characters, ensuring a 100-character overlap between chunks for context preservation.

The “**create_or_load_vector_store**” function checks if a FAISS vector store already exists in the specified directory. If it does, it loads the stored embeddings using “**FAISS.load_local**”. If not, it processes the PDFs by calling “**load_pdfs**”, generates text embeddings using “**OpenAIEmbeddings**”, and creates a new FAISS vector store. This new vector store is then saved locally for future use. The function returns the vector store, enabling efficient document search and retrieval using vector similarity.

```
# Directory to store vector database
DB_FAISS_PATH = "vector_store"
# Load and Process PDFs
def load_pdfs(pdf_paths):
    all_documents = []
    for pdf in pdf_paths:
        loader = PyPDFLoader(pdf)
        documents = loader.load()
        all_documents.extend(documents)
    # Split text into chunks
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100)
    chunks = text_splitter.split_documents(all_documents)
    return chunks
# Create or Load FAISS Vector Store
def create_or_load_vector_store(pdf_paths):
    if os.path.exists(DB_FAISS_PATH):
        print("[INFO] Loading existing vector store...")
        vectorstore = FAISS.load_local(DB_FAISS_PATH, OpenAIEmbeddings())
    else:
        print("[INFO] Creating new vector store...")
        chunks = load_pdfs(pdf_paths)
        vectorstore = FAISS.from_documents(chunks, OpenAIEmbeddings())
        vectorstore.save_local(DB_FAISS_PATH)
    return vectorstore
```

Step 4: Create a Chatbot with Memory

Then, we initialize a chatbot that can retrieve information from a vector store and engage in a conversation while maintaining memory. The **`get_chatbot` function** sets up a chatbot using the **GPT-4 model through `ChatOpenAI`.** It also initializes a **`ConversationBufferMemory`** to store the chat history, allowing the bot to remember previous exchanges within a session. The **`ConversationalRetrievalChain`** is then created, which enables the chatbot to retrieve relevant information from the vector store while keeping track of the conversation context.

The **`chat_with_bot` function** provides an interactive chat interface. It prints a message indicating that the chatbot is ready and waits for user input in a loop. If the user types “exit” or “quit,” the loop breaks, and the program terminates the chat session. Otherwise, the user’s query is passed to the **`qa_chain`**, which retrieves relevant information from the vector store and generates a response. The chatbot’s reply is then printed, allowing for a continuous back-and-forth interaction.

```
# Initialize Chatbot with Memory
def get_chatbot(vectorstore):
    llm = ChatOpenAI(model_name="gpt-4")
    memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
    qa_chain = ConversationalRetrievalChain.from_llm(
        llm, retriever=vectorstore.as_retriever(), memory=memory
    )
    return qa_chain
# Chat with the bot
def chat_with_bot(qa_chain):
    print("\n[INFO] Chatbot is ready! Type 'exit' to quit.")
    while True:
        query = input("\nYou: ")
        if query.lower() in ["exit", "quit"]:
            print("\n[INFO] Exiting chat...\n")
            break
        response = qa_chain.run(query)
        print(f"Bot: {response}")
```

Step 5: Ask the Chatbot and Receive an Answer

```
# Main function
if __name__ == "__main__":
    pdf_files = ["Jira_Software.pdf", "ML+Cheat+Sheet_2.pdf"]  # Replace with your PDFs
    vectorstore = create_or_load_vector_store(pdf_files)
    chatbot = get_chatbot(vectorstore)
    chat_with_bot(chatbot)
```

Output:

```
[INFO] Creating new vector store...
<ipython-input-6-5c12dafc1a0e>:25: LangChainDeprecationWarning: The class `OpenAIEmbeddings` wa
  vectorstore = FAISS.from_documents(chunks, OpenAIEmbeddings())
<ipython-input-7-ec9b7022866c>:3: LangChainDeprecationWarning: The class `ChatOpenAI` was depre
  llm = ChatOpenAI(model_name="gpt-4")
<ipython-input-7-ec9b7022866c>:4: LangChainDeprecationWarning: Please see the migration guide a
  memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
[INFO] Chatbot is ready! Type 'exit' to quit.
You: What is the model in the Jira software pdf?
<ipython-input-7-ec9b7022866c>:18: LangChainDeprecationWarning: The method `Chain.run` was depr
  response = qa_chain.run(query)
Bot: The model in the JIRA software PDF is a deep learning system designed to automate the cate
You: exit
[INFO] Exiting chat...
```

Summary

In Chapter 3, we explored the development of advanced applications powered by large language models (LLMs) using LangChain and Python. We examined how LangChain provides a modular approach to building AI-driven solutions, enabling capabilities like multistep reasoning, dynamic interactions, and seamless data integration. Through practical implementations, such as YouTube video summarization and intelligent document analysis, we demonstrated how LLMs can be applied to real-world problems. Additionally, we addressed key challenges, including optimizing model behavior, handling errors, and ensuring application scalability in production environments.

With a strong foundation in application development, we now shift our focus to deployment strategies in Chapter 4. Building LLM-powered applications is just the beginning—successfully deploying them at scale requires careful consideration of infrastructure, optimization, and performance management. This chapter explores cloud deployment strategies, memory and computational efficiency techniques, and best practices for ensuring security and scalability. By understanding the complexities of deploying LLM applications, you will be equipped to transition from development to real-world implementation, making AI-powered solutions accessible and efficient in production environments.