Retrieval-Augmented Generation (RAG) Pipeline for PDF or Word File

# 🔍 Overview

This document explains the full code and super-detailed explanation of how to build a Retrieval-Augmented Generation (RAG) pipeline to process PDF or Word files. This includes reading, chunking, embedding, retrieving, and generating responses using a language model.

# ✅ Tools Used

- PyMuPDF (fitz) for PDF parsing  
- python-docx for Word file reading  
- Langchain for pipeline building  
- FAISS for vector store  
- HuggingFace or OpenAI LLM for answer generation  
- sentence-transformers for embeddings

# 🔧 1. Install Required Libraries

Run the following command in your terminal:

pip install langchain openai faiss-cpu sentence-transformers python-docx pymupdf

# 📚 2. Import Libraries

from langchain.embeddings import HuggingFaceEmbeddings  
from langchain.vectorstores import FAISS  
from langchain.llms import OpenAI  
from langchain.chains import RetrievalQA  
from langchain.text\_splitter import RecursiveCharacterTextSplitter  
  
import fitz # for PDF  
import docx # for DOCX  
import os

# 📤 3. Step-by-Step RAG Pipeline Code

## Step 1: Read content from PDF or DOCX

def load\_pdf(file\_path):  
 doc = fitz.open(file\_path)  
 text = ""  
 for page in doc:  
 text += page.get\_text()  
 return text  
  
def load\_docx(file\_path):  
 doc = docx.Document(file\_path)  
 text = "\n".join([para.text for para in doc.paragraphs])  
 return text  
  
def load\_file(file\_path):  
 if file\_path.endswith('.pdf'):  
 return load\_pdf(file\_path)  
 elif file\_path.endswith('.docx'):  
 return load\_docx(file\_path)  
 else:  
 raise ValueError("Unsupported file type. Use PDF or DOCX.")

## Step 2: Split Text into Chunks

def split\_text(text, chunk\_size=500, chunk\_overlap=100):  
 splitter = RecursiveCharacterTextSplitter(  
 chunk\_size=chunk\_size,  
 chunk\_overlap=chunk\_overlap  
 )  
 chunks = splitter.split\_text(text)  
 return chunks

## Step 3: Embed Chunks and Store in FAISS Vector DB

def create\_vector\_store(chunks):  
 embeddings = HuggingFaceEmbeddings(model\_name="all-MiniLM-L6-v2")  
 vector\_store = FAISS.from\_texts(chunks, embeddings)  
 return vector\_store

## Step 4: Create RetrievalQA Chain

def create\_qa\_chain(vector\_store):  
 retriever = vector\_store.as\_retriever(search\_type="similarity", k=3)  
 llm = OpenAI(temperature=0)  
 qa\_chain = RetrievalQA.from\_chain\_type(  
 llm=llm,  
 retriever=retriever,  
 chain\_type="stuff"  
 )  
 return qa\_chain

## Step 5: Ask Questions!

def ask\_question(qa\_chain, query):  
 answer = qa\_chain.run(query)  
 return answer

# 🚀 Full Pipeline Usage Example

file\_path = "sample.pdf" # or sample.docx  
  
text = load\_file(file\_path)  
chunks = split\_text(text)  
vector\_store = create\_vector\_store(chunks)  
qa\_chain = create\_qa\_chain(vector\_store)  
  
query = "What is the document about?"  
answer = ask\_question(qa\_chain, query)  
print("Answer:", answer)

# 📌 Explanation Summary

|  |  |
| --- | --- |
| Step | Functionality |
| 1. Load File | Extracts raw text from PDF or Word |
| 2. Chunk Text | Breaks long text into overlapping pieces |
| 3. Embeddings | Converts text to vectors using sentence-transformers |
| 4. FAISS Store | Stores embeddings for fast retrieval |
| 5. QA Chain | Combines retriever + LLM for final answer |
| 6. Query | User asks a question and gets a response |

To process a PDF or Word file using a Retrieval-Augmented Generation (RAG) pipeline, you'll need to set up a system that can handle document ingestion, data chunking, embedding, storage, retrieval, and finally, generation. This example uses Python libraries like langchain, pypdf, docx2txt, and faiss.

**Setting up the Environment**

First, install the necessary libraries.

Bash

pip install langchain langchain-openai pypdf docx2txt faiss-cpu tiktoken

You'll need an API key for OpenAI to use their embedding and large language models (LLMs). Set this as an environment variable.

Bash

export OPENAI\_API\_KEY="your-api-key"

**1. Document Loading**

The first step is to load the content from the files. Different file types require different loaders. We'll create a function that handles both PDF and Word files.

Python

from langchain.document\_loaders import PyPDFLoader, Docx2txtLoader

import os

def load\_document(file\_path):

"""

Loads a document from a given file path.

Supports PDF and DOCX formats.

"""

file\_extension = os.path.splitext(file\_path)[1]

if file\_extension == ".pdf":

loader = PyPDFLoader(file\_path)

elif file\_extension == ".docx":

loader = Docx2txtLoader(file\_path)

else:

raise ValueError("Unsupported file type")

return loader.load()

This function checks the file extension and uses the appropriate langchain loader to read the document's content.

**2. Document Splitting (Chunking)**

Large documents need to be broken down into smaller, manageable pieces called **chunks**. This is crucial for RAG because it allows the model to retrieve relevant, specific information rather than processing an entire large document at once. A good chunk size prevents loss of context. We'll use a RecursiveCharacterTextSplitter.

Python

from langchain.text\_splitter import RecursiveCharacterTextSplitter

def split\_documents(documents, chunk\_size=1000, chunk\_overlap=200):

"""

Splits a list of documents into smaller chunks.

"""

text\_splitter = RecursiveCharacterTextSplitter(

chunk\_size=chunk\_size,

chunk\_overlap=chunk\_overlap,

length\_function=len

)

return text\_splitter.split\_documents(documents)

The chunk\_size determines the maximum number of characters in a chunk, while chunk\_overlap ensures that chunks have some common text at their boundaries to maintain context.

**3. Embedding and Vector Store Creation**

After chunking, each chunk is converted into a numerical vector using an **embedding model**. This vector represents the semantic meaning of the text. These vectors are then stored in a **vector database** (or vector store), which is optimized for fast similarity searches. We'll use FAISS (Facebook AI Similarity Search) as our vector store and OpenAI's text-embedding-ada-002 model for embeddings.

Python

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.vectorstores import FAISS

def create\_vector\_store(chunks):

"""

Creates a FAISS vector store from document chunks.

"""

embeddings = OpenAIEmbeddings()

vector\_store = FAISS.from\_documents(chunks, embeddings)

return vector\_store

FAISS.from\_documents takes the text chunks and the embedding model, computes the embeddings for each chunk, and builds an index that allows for efficient retrieval.

**4. Retrieval and Generation**

This is the core of the RAG pipeline. When a user asks a question, the process is:

1. The user query is also converted into an embedding using the same model.
2. The vector store is queried to find the chunks with the most similar embeddings to the query. These are the most relevant chunks.
3. These retrieved chunks, along with the original user query, are then passed to a powerful LLM (like gpt-4).
4. The LLM uses this retrieved context to generate a precise, informed answer.

Python

from langchain.chains import ConversationalRetrievalChain

from langchain.chat\_models import ChatOpenAI

def get\_rag\_chain(vector\_store):

"""

Creates a RAG chain for question-answering.

"""

llm = ChatOpenAI(model\_name="gpt-3.5-turbo", temperature=0)

# We use a retriever to get the most relevant documents

retriever = vector\_store.as\_retriever()

# ConversationalRetrievalChain is designed for chat-like interactions

rag\_chain = ConversationalRetrievalChain.from\_llm(

llm=llm,

retriever=retriever,

return\_source\_documents=True

)

return rag\_chain

The ConversationalRetrievalChain is a robust tool from langchain that handles the retrieval and generation steps seamlessly. It uses a **retriever** to fetch the relevant documents and an llm to formulate the final answer.

**Full Pipeline Integration**

Now, let's put all the pieces together into a single, executable script. This script will perform all the steps from loading the document to answering a user query.

Python

import os

from langchain.document\_loaders import PyPDFLoader, Docx2txtLoader

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

# Set up your OpenAI API key

os.environ["OPENAI\_API\_KEY"] = "your\_api\_key\_here"

# 1. Document Loading

def load\_document(file\_path):

"""Loads a document from a given file path."""

file\_extension = os.path.splitext(file\_path)[1]

if file\_extension == ".pdf":

loader = PyPDFLoader(file\_path)

elif file\_extension == ".docx":

loader = Docx2txtLoader(file\_path)

else:

raise ValueError("Unsupported file type")

return loader.load()

# 2. Document Splitting

def split\_documents(documents, chunk\_size=1000, chunk\_overlap=200):

"""Splits a list of documents into smaller chunks."""

text\_splitter = RecursiveCharacterTextSplitter(

chunk\_size=chunk\_size,

chunk\_overlap=chunk\_overlap,

length\_function=len

)

return text\_splitter.split\_documents(documents)

# 3. Embedding and Vector Store Creation

def create\_vector\_store(chunks):

"""Creates a FAISS vector store from document chunks."""

embeddings = OpenAIEmbeddings()

vector\_store = FAISS.from\_documents(chunks, embeddings)

return vector\_store

# 4. Retrieval and Generation

def get\_rag\_chain(vector\_store):

"""Creates a RAG chain for question-answering."""

llm = ChatOpenAI(model\_name="gpt-3.5-turbo", temperature=0)

retriever = vector\_store.as\_retriever()

rag\_chain = ConversationalRetrievalChain.from\_llm(

llm=llm,

retriever=retriever,

return\_source\_documents=True

)

return rag\_chain

# Main execution function

def main\_rag\_pipeline(file\_path, user\_query):

"""

Executes the full RAG pipeline for a given document and query.

"""

print("Step 1: Loading document...")

documents = load\_document(file\_path)

print(f"Loaded {len(documents)} pages/sections.")

print("Step 2: Splitting document into chunks...")

chunks = split\_documents(documents)

print(f"Created {len(chunks)} chunks.")

print("Step 3: Creating vector store...")

vector\_store = create\_vector\_store(chunks)

print("Vector store created successfully.")

print("Step 4: Setting up RAG chain...")

rag\_chain = get\_rag\_chain(vector\_store)

print("RAG chain ready. Answering query...")

# Answer the user query

chat\_history = []

response = rag\_chain({"question": user\_query, "chat\_history": chat\_history})

print("\n--- RAG Pipeline Results ---")

print("Question:", user\_query)

print("Answer:", response['answer'])

print("\n--- Source Documents ---")

for i, doc in enumerate(response['source\_documents']):

print(f"Source {i+1}:")

print(f"Content: {doc.page\_content[:200]}...") # Show first 200 characters

print(f"Source Metadata: {doc.metadata}\n")

return response

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

# Create a dummy PDF and DOCX file for testing

with open("example.pdf", "w") as f:

f.write("This is a dummy PDF file content. It talks about the history of artificial intelligence and its impact on the modern world. AI has evolved significantly over the last few decades, becoming a pivotal technology.")

with open("example.docx", "w") as f:

f.write("This is a dummy DOCX file content. It discusses the basics of machine learning, including supervised and unsupervised learning. Machine learning is a subset of AI that focuses on enabling systems to learn from data.")

# Process a PDF file

pdf\_response = main\_rag\_pipeline("example.pdf", "What is artificial intelligence?")

print("\n" + "="\*50 + "\n")

# Process a DOCX file

docx\_response = main\_rag\_pipeline("example.docx", "What are the types of machine learning mentioned?")

This script demonstrates a **complete and functional RAG pipeline**, from file ingestion to a contextualized answer. The return\_source\_documents=True argument in the chain is particularly useful as it allows you to inspect which parts of the original document were used to formulate the answer, providing **transparency and traceability**.