Let's break down the code step by step, explaining each function and its utilization.

**1. Library Imports:**

import os

import time

import streamlit as st

from dotenv import load\_dotenv

from langchain\_groq import ChatGroq

from langchain\_community.document\_loaders import PyPDFLoader

from langchain\_community.embeddings import OllamaEmbeddings

from langchain\_community.vectorstores import FAISS

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain.chains.combine\_documents import create\_stuff\_documents\_chain

from langchain\_core.prompts import ChatPromptTemplate

from langchain.chains import create\_retrieval\_chain

* **os**: Provides a way to interact with the operating system. Here, it's used for file handling (e.g., deleting temporary files).
* **time**: Used for performance measurement (tracking response time).
* **streamlit**: A framework for creating web apps. It is used to create a user interface for document summarization.
* **dotenv**: Loads environment variables from a .env file into Python, typically to store sensitive information like API keys.
* **PyPDFLoader**: Used to load and process PDFs.
* **OllamaEmbeddings**: Provides an embedding model to transform textual data into vector representations using Ollama's embedding service.
* **FAISS**: A vector database for efficient similarity search. It helps in storing and retrieving document embeddings.
* **RecursiveCharacterTextSplitter**: A utility that splits long documents into smaller chunks for efficient processing.
* **create\_stuff\_documents\_chain**: Creates a chain where multiple document summaries or outputs are "stuffed" or combined together.
* **ChatPromptTemplate**: Used to create a prompt template for the LLM that accepts a context and returns a structured output.
* **create\_retrieval\_chain**: Builds a pipeline where the retriever first pulls relevant information from the vector database, and then the chain processes that information.

**2. Loading Environment Variables:**

load\_dotenv()

* This loads the environment variables from the .env file (if any), such as API keys required for external services like Groq LLM.

**3. Initializing the LLM:**

groq\_api\_key = "############################"

* llm = Ollama(model="qwen2:1.5b",temperature=0.3)
* **model**: A placeholder key that indicates model name.
* **temperature**: set llm model temperature.

**4. Streamlit UI Setup:**

st.set\_page\_config(layout="wide")

st.title("Document Summarization App with Groq")

* **set\_page\_config**: Configures the layout of the Streamlit app to be wide, so the web interface utilizes the full screen width.
* **st.title**: Displays the title at the top of the web app.

**5. Initializing Session State:**

if "vector" not in st.session\_state:

st.session\_state.embeddings = OllamaEmbeddings(model="paraphrase-multilingual:latest ")

st.session\_state.documents = []

st.session\_state.vector = None

* **st.session\_state**: Streamlit's way of keeping track of variables across multiple user interactions.
* **OllamaEmbeddings**: Initializes the embeddings model from Ollama to transform text into vectors for later retrieval tasks.
* **st.session\_state.vector**: Stores the FAISS vector database, which will later be populated (used)with document embeddings.

**6. Preprocessing the Uploaded File:**

def preprocess\_file(uploaded\_file):

temp\_file\_path = "temp\_" + uploaded\_file.name

with open(temp\_file\_path, "wb") as temp\_file:

temp\_file.write(uploaded\_file.read())

loader = PyPDFLoader(temp\_file\_path)

pages = loader.load\_and\_split()

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1500, chunk\_overlap=300)

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

os.remove(temp\_file\_path)

return texts

* **uploaded\_file**: A PDF file uploaded by the user.
* **temp\_file\_path**: The temporary path to store the uploaded PDF file.
* **PyPDFLoader**: Loads the PDF and splits it into pages.
* **RecursiveCharacterTextSplitter**: Splits the pages into smaller chunks of 1500 characters with 300 characters of overlap to avoid losing context when the document is summarized.
* **os.remove**: Deletes the temporary PDF file after processing.

**7. Setting up the Vector Database:**

def setup\_vector\_db(docs):

vectordb = FAISS.from\_documents(docs, st.session\_state.embeddings)

return vectordb

* **setup\_vector\_db**: Converts the list of document chunks into vectors and stores them in a FAISS vector database using the Ollama embeddings.
* **FAISS.from\_documents**: Creates the FAISS index by embedding the document chunks and adding them to the database.

**8. Streamlit UI and Summarization Logic:**

def main():

uploaded\_file = st.file\_uploader("Upload your PDF file", type=['pdf'])

if uploaded\_file is not None:

if st.button("Summarize"):

st.session\_state.documents = preprocess\_file(uploaded\_file)

st.session\_state.vector = setup\_vector\_db(st.session\_state.documents)

# Refined summarization prompt for concise output in Arabic

prompt\_template = """

أنت خبير في تلخيص النصوص. يرجى تقديم ملخص موجز ودقيق للوثيقة التالية باللغة العربية على شكل نقاط رئيسية.

يرجى التأكد من أن الملخص يحتوي على 10 نقاط على الأقل، وكل نقطة يجب أن تكون موجزة وتلخص فكرة رئيسية من المستند.

<context>

{context}

</context>

الملخص في شكل نقاط رئيسية (10 نقاط على الأقل):

- """

prompt = ChatPromptTemplate.from\_template(prompt\_template)

document\_chain = create\_stuff\_documents\_chain(llm, prompt)

retriever = st.session\_state.vector.as\_retriever()

retrieval\_chain = create\_retrieval\_chain(retriever, document\_chain)

start = time.process\_time()

summary\_response = retrieval\_chain.invoke({"input": ""})

response\_time = time.process\_time() - start

st.write(f"Response time: {response\_time} seconds")

st.success(summary\_response["answer"])

* **st.file\_uploader**: Allows users to upload a PDF file.
* **preprocess\_file**: The uploaded PDF is preprocessed into chunks using the previously defined function.
* **setup\_vector\_db**: A vector database (FAISS) is created using the document chunks.
* **prompt\_template**: A template for the LLM to summarize the document. {context} is the placeholder where document chunks will be inserted.
* **ChatPromptTemplate**: Creates a chat-style prompt using the provided template.
* **create\_stuff\_documents\_chain**: Combines document chunks and sends them to the LLM for summarization.
* **retriever**: The FAISS vector store is used to retrieve relevant document chunks.
* **create\_retrieval\_chain**: The retrieval chain connects the retriever and document summarization chain to produce a final summary.
* **time.process\_time**: Measures the time taken to process and generate the summary.
* **st.write**: Displays the response time in seconds.
* **st.success**: Displays the summarization result in the app.

**9. Main Function:**

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

main()

* This ensures that the main() function is run when the script is executed directly.

Each component and function helps build a system where a user uploads a PDF, which is split into chunks, embedded into vectors, and summarized by llm provided by ollama , with the results displayed via the Streamlit app.