Docx-DRS: Advanced Document Retrieval System

**1. Introduction**

* **1.1 Project Title:** Advanced Document Search System with FAISS
* **1.2 Overview:** This project is a comprehensive document search and analysis tool built using Python, leveraging cutting-edge techniques in Natural Language Processing (NLP), Artificial Intelligence (AI), and vector database technology. The application provides a user-friendly web interface, powered by Streamlit, allowing users to upload documents, ask questions in natural language, and receive direct answers extracted from those documents.
* **1.3 Purpose:** The primary objective of this project is to create an efficient and intelligent system that can quickly and accurately extract information from diverse document types. Unlike traditional keyword-based searches, this system performs semantic searches, understanding the meaning and context of queries, which leads to more relevant results.
* **1.4 Target Users:** The system is designed for users who frequently work with documents, such as researchers, analysts, project managers, and support staff, or anyone who needs to efficiently retrieve information from large volumes of document data.

**2. Project Goals and Objectives**

* **2.1 Primary Goal:** To develop a robust and user-friendly document search application that can accurately extract information from documents through natural language queries.
* **2.2 Key Objectives:**
  + **Support for Multiple File Types:** The system should be able to process various document formats, including PDF, DOCX, XLSX, PNG, and JPG.
  + **Text Extraction:** Implement robust text extraction capabilities using relevant libraries, including Optical Character Recognition (OCR) for images and scanned PDF files.
  + **Semantic Search:** Implement a semantic search using FAISS, capable of understanding the context and intent of user queries.
  + **Intelligent Suggestions:** Provide AI-powered question suggestions to guide users with query generation, based on the uploaded documents.
  + **Direct Answer Extraction:** Extract precise answers from the documents based on the context of user questions using LLM.
  + **Efficient Search:** Provide fast retrieval of relevant documents, utilizing FAISS for quick similarity calculations.
  + **User-Friendly Interface:** Create an intuitive and user-friendly web interface using Streamlit.
  + **Easy Access:** Allow user to download the original documents from the results page.

**3. Software Development Lifecycle (SDLC)**

* **3.1 SDLC Model:** The project follows a flexible, iterative, and incremental approach, which is well suited to this project's requirements.
  + **Iterative Nature:** The project is built over several iterations, where the current stage is improved and extended. This will involve continuous planning, design, development and testing.
  + **Incremental Approach:** Each iteration delivers a partial but functioning product.
* **3.2 Stages of the SDLC:**
  + **3.2.1 Planning:**
    - **Requirements Gathering:** Detailed collection of functional and non-functional user requirements.
    - **Scope Definition:** Clearly defining the scope and boundaries of the project.
    - **Technology Selection:** Choosing relevant frameworks and libraries (Streamlit, Groq, FAISS, etc.).
    - **Resource Allocation:** Estimating needed time and budget and identifying team members and responsibilities.
    - **Risk Management:** Identifying risks such as technical challenges or lack of APIs and having strategies to manage them.
  + **3.2.2 Design:**
    - **System Architecture Design:** Planning out high level components (UI, Document Processor, vector store)
    - **User Interface (UI) Design:** Sketching user journeys, interaction flow and wireframes.
    - **Data Model Design:** Deciding how data is structured and organized during processing and storage.
    - **Algorithm Design:** How is semantic similarity calculated, and how is the search performed.
    - **API Design:** How is the system going to interact with the Groq LLM.
  + **3.2.3 Development:**
    - **Code Implementation:** Writing the source code and developing the functionalities in app.py and document\_processor.py.
    - **Database Implementation:** Setting up and managing the FAISS vector index used to store the document embeddings.
    - **API Integration:** Developing interfaces for interacting with the Groq API for suggestions and answer extraction.
    - **Unit Testing:** Regularly writing and running tests for each function/module, focusing on smaller components.
  + **3.2.4 Testing:**
    - **Functional Testing:** Ensuring that all requirements and the system work as expected.
    - **Performance Testing:** Measure the speed and efficiency of the application under various loads.
    - **User Acceptance Testing:** Users test the system and confirm whether it meets their requirements.
    - **Edge Case Testing:** Check how the system handles incorrect inputs or edge cases.
  + **3.2.5 Deployment:**
    - **Deployment Plan:** Deciding on which platform the application will be hosted and how it is going to be deployed (Local, Streamlit cloud, other cloud provider)
    - **Infrastructure Setup:** Setting up servers/cloud resources required for running the application.
    - **Roll Out:** Deploying the application to the production environment.
  + **3.2.6 Maintenance:**
    - **Bug Fixes:** Resolving reported issues.
    - **Performance Improvements:** Optimizing system performance and response time.
    - **Feature Enhancements:** Adding new features or functionalities based on user feedback.
    - **Version Control:** Managing codebase through a Git repository

**4. app.py File Description**

* **4.1 File Purpose:** The app.py file acts as the main application script, controlling the user interface, the flow of the program, and the user experience. It leverages the Streamlit library to create an interactive web application.
* **4.2 Detailed Breakdown:**
  + **4.2.1 Imports:**
    - import streamlit as st: Imports the Streamlit library to create web UI elements.
    - import os: Imports the OS library, mainly used for handling environment variables.
    - from document\_processor import DocumentProcessor: Imports the DocumentProcessor class from the document\_processor.py file.
  + **4.2.2 Main Function main ():**
    - **Application Title:** st.title("📄 Advanced Document Search System with FAISS") sets the application's title, shown on the web page.
    - **Session State Initialization:**
      * if 'documents' not in st.session\_state: and subsequent statements initializes session variables.
      * The use of st.session\_state is crucial for maintaining the app's state between interactions. This is important when handling data from different components of your app, such as file uploads and searches.
    - **File Uploader Component:**
      * st.file\_uploader(...): Provides a button for the user to upload files and configures which file types are accepted.
      * The accepted file types are pdf, docx, xlsx, png, and jpg.
    - **Document Processing:**
      * When user uploads files, the app first shows a st.spinner to indicate that it's loading
      * It clears any existing documents and file details from st.session\_state, preventing the previous values from showing.
      * For each uploaded file it calls the DocumentProcessor.extract\_text method to extract text from the uploaded documents.
      * The extracted text is added to a list in session state, along with the file name and file data.
      * The method DocumentProcessor.generate\_suggestions is called to produce suggested questions based on the uploaded content
      * Then, the DocumentProcessor.create\_vector\_store method is called to create the vector store with FAISS using all the uploaded documents.
    - **Suggestion Display:**
      * The application shows a header called Document Suggestions to display the suggestion section.
      * The suggestions will be displayed in a select box using st.selectbox. This allows users to select one of the AI-generated suggestions.
      * The function suggestion\_selected handles the search when user selects a suggestion.
      * The function uses the st.session\_state.doc\_processor.semantic\_search method to search for the relevant passages.
  + **Custom Search:**
    - st.button("Custom Search"): A custom search button allows user to do a custom search instead of using the suggestions.
    - if st.session\_state.custom\_search\_active: is used to conditionally show a custom search input.
    - st.text\_input gives user a place to write a custom query.
    - When the user presses the search button, a new semantic search is triggered using the st.session\_state.doc\_processor.semantic\_search method  
      \* **Search Results Display:**
    - A section named Search Results is displayed at the bottom.
    - If there are search results, it loops through each result and calls semantic\_search () to search for relevant passages and answers for the given document.
    - For each document, it shows the document name, similarity score, the question, and the answer.
    - It also provides a download button for each document, using st.download\_button.
    - A st.markdown("---") line is added to separate results clearly.  
      \* **Warning Message:**
    - If no files are uploaded the application will show a st.warning message.
  + **Sidebar:**
    - The application has a st.sidebar to show extra details.
    - It displays the app title, capability list and acknowledgements at the bottom.  
      \* **Main Execution Block:**
    - if \_\_name\_\_ == "\_\_main\_\_": this conditional block ensures that main() is called when the app.py script is executed directly.

**5. document\_processor.py File Description**

* **5.1 File Purpose:** The document\_processor.py file contains the DocumentProcessor class, which encapsulates all logic related to processing documents, generating embeddings, performing semantic searches, and extracting answers. This modular design promotes code reusability and maintainability.
* **5.2 Detailed Breakdown:**
  + **5.2.1 Imports:**
    - import os: For handling environment variables.
    - from langchain\_groq import ChatGroq: To interface with the Groq LLM API.
    - import PyPDF2: For PDF file reading.
    - import docx: For Word file reading.
    - import pandas as pd: For reading Excel files.
    - from PIL import Image: For loading images.
    - import pytesseract: For Optical Character Recognition (OCR).
    - import re: For regular expressions and text manipulation.
    - from pdf2image import convert\_from\_path: For converting PDF to images.
    - import faiss: For creating a FAISS index for vector similarity search.
    - import numpy as np: For numerical operations with arrays.
    - from typing import List, Dict: For type hinting.
    - import torch: Support library for sentence-transformers
    - from sentence\_transformers import SentenceTransformer: For text embedding generation.
    - import streamlit as st: To handle streamlit error messages.
  + **5.2.2 DocumentProcessor Class:**
    - \_\_init\_\_: Constructor that initializes the SentenceTransformer model and sets the chunk\_size.
      * self.embedder = SentenceTransformer('all-MiniLM-L6-v2'): Initializes the sentence transformer model.
      * self.chunk\_size = 3: defines chunk size as 3 sentences
    - extract\_text(uploaded\_file): (static method)
      * This static method accepts uploaded files and returns the raw string.
      * It determines the file type using uploaded\_file.type and depending on the file type it will use a different method for extraction
      * For PDF files it will use PyPDF2.PdfReader to extract text from the pdf, if it fails it will attempt to do image processing using pdf2image and OCR using pytesseract.  
        \* For DOCX files it uses docx.Document to extract the text.  
        \* For XLSX files it uses pandas.read\_excel and creates a single string from all the cells.  
        \* For image files it uses PIL.Image to load the image and pytesseract to perform OCR.
    - preprocess\_text(text: str) -> List[str]: Splits the input text into chunks of sentences, enhancing indexing effectiveness.
      * This function first removes leading/trailing whitespace and multiple spaces and converts to a single space.
      * It will split the text into sentences using regular expressions
      * It will create chunks of self.chunk\_size number of sentences and return them as a list
    - create\_vector\_store(documents: List[str]) -> tuple: Creates FAISS index from processed documents.
      * This function will take a list of str documents as input.
      * It calls self.preprocess\_text for each document and creates the chunks.
      * For each chunk it will save the index of the original document.
      * It will encode the chunks using self.embedder model to get the embeddings.
      * It creates a FAISS index using faiss.IndexFlatL2
      * It adds the embeddings to the FAISS index
      * The method then returns the FAISS index and the list of chunks of text.
    - extract\_answer(self, question, context): Uses the Groq LLM to extract direct answers for given questions and a context.
      * This method uses the groq\_api\_key from environment variables or defaults to a dummy\_key if it is not available.
      * It initializes the ChatGroq class with Llama3-8b-8192 as the model’s name
      * It has a prompt that will use the context and the question to extract an answer from the Groq model.
      * The answer is then returned, if it fails for any reason it will return "Error getting answer".
    - semantic\_search(query: str, index, chunks: List[str], chunk\_to\_doc\_map: List[int], k: int = 3) -> List[Dict]: Performs semantic searches using the FAISS index.
      * This method takes a query and the index and chunks created by create\_vector\_store as well as the number of search results to return.
      * First, it converts the query to an embedding using self.embedder.
      * It will search for similar embeddings using index.search.
      * For each match it will get the associated document index using the chunk\_to\_doc\_map, then it will call extract\_answer to extract the answer from the corresponding chunk.
      * Then it creates a dict with the original document index, question, extracted answer and the distance.
      * The results will be sorted by distance and then grouped by document index
      * Finally, the results are transformed to have questions and answers grouped together per document and is returned.
  + generate\_suggestions(text): (static method)
    - This static method generates suggested questions for a user based on a document's text.
    - It uses the groq\_api\_key from environment variables or defaults to a dummy\_key if it is not available.
    - It initializes the ChatGroq class with Llama3-8b-8192 as the model’s name
    - A prompt is given to Groq LLM to ask questions about documents focusing on specific details.  
      \* It processes and returns the suggestions as a list of strings.  
      \* Tesseract\_path: Defines the path to Tesseract OCR executable using an environment variable, if it is not provided a default path is provided.

**6. Technologies Used**

* **Streamlit:** For creating the web-based user interface.
* **Langchain-Groq:** For accessing the Groq LLM and getting intelligent question suggestions and answer extraction.
* **PyPDF2:** To extract text from PDF documents.
* **python-docx:** To read and extract text from Word (DOCX) documents.
* **pandas:** To process and read data from Excel (XLSX) files.
* **Pillow (PIL):** For image processing, specifically for loading and manipulating images.
* **pytesseract:** For OCR (Optical Character Recognition), used to extract text from images or scanned documents.
* **pdf2image:** For converting PDF files into images before performing OCR.
* **FAISS (Facebook AI Similarity Search):** For efficient and fast vector similarity searches.
* **sentence-transformers:** For generating text embeddings.
* **Torch:** Support library for sentence-transformers
* **numpy:** For numerical operations.
* **os:** For interacting with the operating system, especially for handling environment variables.
* **re:** For text manipulation using regular expressions.

**7. Setup and Installation**

* **7.1 Prerequisites:**
  + Python 3.7 or higher
  + Pip package manager
* **7.2 Installation Instructions:**
  + Create a new virtual environment (recommended):
  + python -m venv venv
  + source venv/bin/activate # On macOS/Linux

venv\Scripts\activate # On Windows

* + Install the dependencies:

pip install -r requirements.txt

pip install streamlit langchain-groq PyPDF2 python-docx pandas pillow pytesseract pdf2image faiss-cpu numpy sentence-transformers torch

* + Set the GROQ\_API\_KEY environment variable with your actual Groq API key.
  + Create an account in Groq website for API access.
  + Run the application from the terminal:

streamlit run app.py

1. Access the application through your browser at the provided link.

**8. Code Structure and Conventions:**

* **Modularity:** The code is divided into two main files (app.py and document\_processor.py) for better organization and maintainability.
* **Class-Based Approach:** The DocumentProcessor class encapsulates all document processing logic.
* **Type Hinting:** Type hints are used throughout the code to improve readability and prevent errors.
* **Clear Naming:** Variable and method names are descriptive and follow Python conventions (snake\_case).
* **Error Handling:** The code includes basic error handling to catch exceptions and prevent crashes.
* **Commenting:** The code contains detailed comments to explain the purpose of each function.

**9. Future Enhancements**

* Support for more file types (e.g. HTML, TXT)
* Advanced result ranking or grouping
* User authentication
* More sophisticated prompts for the LLM, to improve answer extraction
* Multi-document indexing