PROJECT REPORT

DocQuest: interactive document reader and Q&A Assistant

Submitted by:
Keerthi Devendiran
Final year (IV),
Btech AI&DS,
Panimalar Institute of Technology.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to the Liquid Propulsion Systems Centre (LPSC) of the Indian Space Research Organisation (ISRO) for giving us the chance to work on the '**DocQuest: Interactive Document Reader and Q&A Assistant'** in LPSC , ISRO. Our sincere thanks to **Manusubramanian S,** DDH CND/CISDG and **Sandeep Nithyanandan,** Sci/Engr 'SC', CND/CISDG , our project guides at Computer Infrastructure and Software Development Group , LPSC for their invaluable support and guidance throughout the course of our project. Their extensive knowledge , patience and willingness to provide constructive feedback were instrumented in the successful completion of this project.

We are also grateful to the entire team at LPSC , Valiamala , Trivandrum for providing an exceptional environment conducive to research and learning. The resources and facilities made available to us were indispensable , and the collaborative spirit within the center greatly enhanced our learning experience.

Lastly we would like to thank our family and friends for their constant encouragement and unwavering support which were vital in helping us navigate the challenges encountered during this project.

Table of Contents

INTRODUCTION:

“DocQuest: Interactive Document Reader and Q&A Assistant" is an innovative project designed to streamline the process of extracting and retrieving information from multiple PDF documents. The primary objective of DocQuest is to enable users to interactively query a collection of PDFs and receive precise and relevant answers. This system leverages advanced natural language processing techniques(OCR techniques) to extract text from PDFs, convert the text into vector representations and store these vectors in a vector database. Upon receiving a query, DocQuest uses sentence embeddings to find the most relevant text snippets from the document collection and generate concise answers. This tool aims to enhance productivity and efficiency by providing quick and accurate responses to user queries, making it an invaluable resource for professionals handling large volumes of textual data.

OBJECTIVE:

DocQuest is designed to streamline information extraction and retrieval from multiple PDFs. It allows users to interactively query a collection of PDFs and receive precise answers. Using advanced OCR techniques, it extracts text, converts it into vector representations, and stores these vectors in a database. Upon receiving a query, it uses sentence embeddings to find relevant text snippets and generate concise answers, enhancing productivity and efficiency for professionals handling large volumes of textual data.

METHODOLOGY:

OCR Techniques:

Optical Character Recognition (OCR) is a technology used to convert different types of documents, such as scanned paper documents, PDF files, or images captured by a digital camera, into editable and searchable data. OCR systems recognize the characters present in the document, such as letters, numbers, and symbols, and convert them into machine-readable text. Key components and steps in OCR include,

1. **Image Preprocessing:** Enhances the quality of the input image by removing noise, correcting distortions, and adjusting brightness and contrast to improve character recognition accuracy.
2. **Text Detection:** Identifies and isolates the text regions in the image, differentiating them from non-text regions such as images, graphics, and backgrounds.
3. **Character Recognition:** Uses algorithms to identify and convert the detected text regions into corresponding digital text. This involves pattern recognition and machine learning techniques to match characters with known character sets.
4. **Post-processing:** Corrects errors and refines the recognized text by using dictionaries and language models to improve accuracy, especially for complex documents or handwriting.

OCR Types:

For text extraction using OCR in Python, several libraries and frameworks can be integrated with machine learning libraries. These OCR tools provide robust solutions for extracting text from images and PDFs, for further processing, analysis, and custom model training. Here are some popular OCR tools suitable for text extraction,

1. Pytesseract
2. Doctr
3. EasyOCR
4. Keras-ocr
5. Google Cloud Vision
6. PaddleOCR

EasyOCR

EasyOCR is a deep learning-based Optical Character Recognition (OCR) library developed by the Jaided AI team. Built on PyTorch, it is designed to be easy to use, flexible, and capable of recognizing text in multiple languages. It leverages deep learning models trained on a diverse range of scripts, allowing it to handle various fonts, orientations, and challenging text layouts, such as rotated text or text within images.Users can perform OCR on images and PDFs, and the library supports both single-line and multi-line text recognition.

Features:

1. Deep Learning Based: Utilizes deep learning techniques, specifically convolutional neural networks (CNNs), for high-accuracy text recognition.
2. Multilingual Support: Supports over 80 languages, making it highly versatile for global applications.
3. Ease of Use: Designed to be user-friendly, with a simple API for quick deployment and integration.
4. Pre-trained Models: Includes pre-trained models that can be used out of the box for a variety of text recognition tasks.
5. Custom Model Training: Allows users to fine-tune existing models or train new ones on their own datasets.
6. Flexible Input: Capable of processing text from images, scanned documents, and even camera feeds.

Detailed Steps:

1. Image Preprocessing: EasyOCR includes internal preprocessing steps to enhance image quality and improve OCR accuracy. Users can also perform their own preprocessing (e.g., resizing, denoising) before passing images to EasyOCR.
2. Text Detection: The first stage involves detecting text regions within the image. EasyOCR uses a combination of neural network models to accurately identify areas containing text.
3. Text Recognition: Once text regions are detected, the text recognition model processes these regions to convert them into machine-readable text. This involves character segmentation and classification
4. Result Formatting: The recognized text, along with bounding box coordinates and confidence scores, is returned in a structured format. This makes it easy to further process, analyze, or visualize the results.

Advantages of EasyOCR

1. **High Accuracy:**Deep Learning Models: Uses state-of-the-art deep learning models for both text detection and recognition, providing high accuracy even in challenging conditions.
2. **Multilingual Support**:Wide Language Coverage: Supports over 80 languages, including complex scripts and right-to-left languages, making it suitable for international use.
3. **Ease of Use**: Simple API: Designed with a user-friendly API that simplifies the process of performing OCR tasks, making it accessible even to those with limited deep learning experience.

COMPARATIVE STUDY:

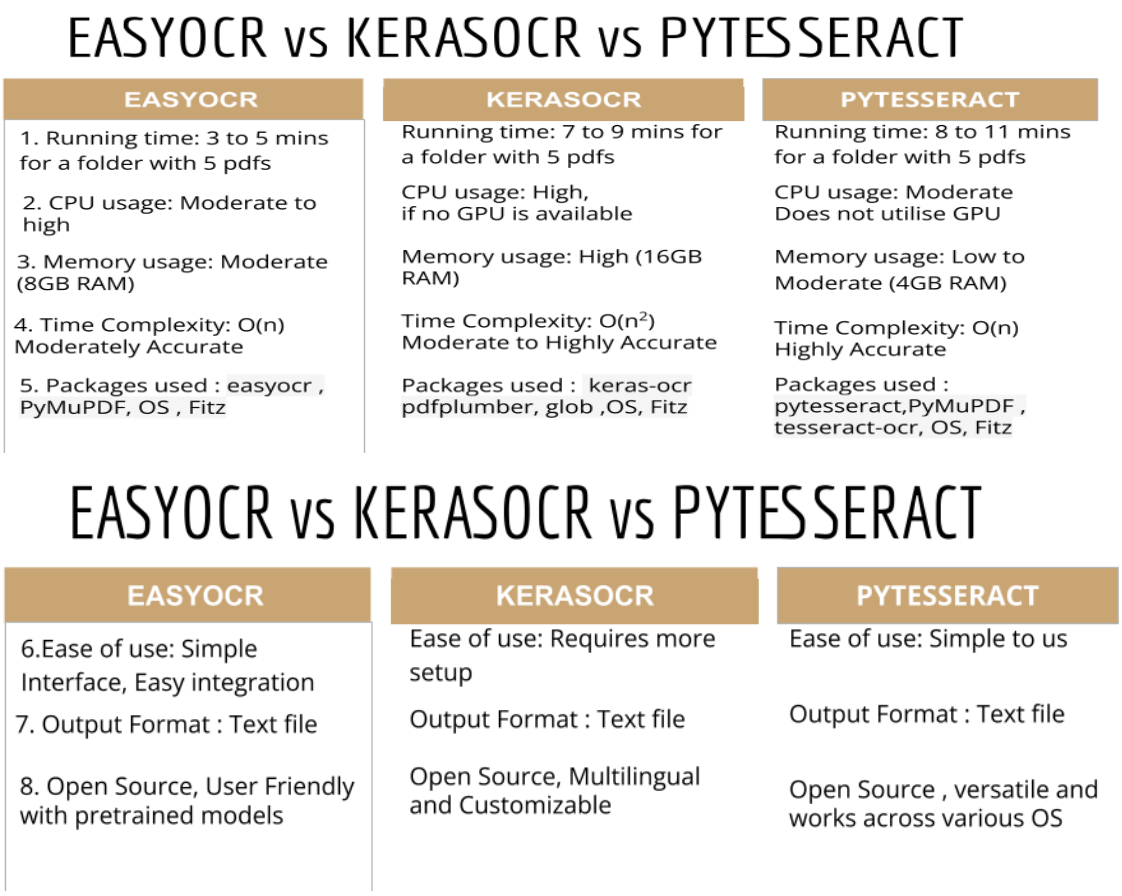

Fig.1: Comparison of OCRs

OVERALL EFFICIENCY:

When evaluating OCR solutions for PDFs, efficiency is determined by factors such as accuracy, ease of integration, support for complex layouts, and processing speed. PyTesseract is a reliable choice for straightforward text extraction due to its balance of accuracy, ease of integration, and extensive language support. It performs well with simple, well-formatted PDFs but may require additional preprocessing for documents with complex layouts. On the other hand, EasyOCR utilizes advanced deep learning models, offering high accuracy and effective handling of noisy or complex text. It supports over 80 languages, making it versatile for international PDFs, but can be more resource-intensive. Keras-OCR stands out for its state-of-the-art deep learning models and end-to-end pipeline, which excels in recognizing text from complex layouts and customized tasks. However, it may require more setup and familiarity with Keras and TensorFlow. Overall, Keras-OCR is highly efficient for handling intricate document layouts and specialized OCR needs, while PyTesseract and EasyOCR offer excellent performance with their

own strengths, with EasyOCR being particularly adept at dealing with complex text in challenging conditions.

IMPLEMENTATION:

1. **OCR**

EasyOcr:

➔ **Installations:**

pip install easyocr PyMuPDF

➔ **Import Libraries:**

import os
import easyocr
import fitz

➔ Initialize EasyOCR Reader: Creates an EasyOCR reader object for text extraction, specifying the language(s) as needed (in this case, English).

➔ Create Output Folder: Checks if the output folder exists; if not, it creates the folder.

➔ List PDF Files: Lists all PDF files in the input folder.

➔ Process Each PDF: For each PDF file:

- Extract Text: Calls extract_text_from_pdf() to extract text from the PDF.
- Save Text: Writes the extracted text to a .txt file in the output folder.
- Print Status: Outputs a message indicating the completion of the text extraction and saving process.

```
import os
import easyocr
import fitz  # PyMuPDF

# Function to extract text from a PDF file using PyMuPDF
def extract_text_from_pdf(pdf_path):
    text = ""
    doc = fitz.open(pdf_path)
    for page_num in range(len(doc)):
        page = doc.load_page(page_num)
        text += page.get_text()
    doc.close()
    return text

# Function to extract text from all PDFs in a folder using EasyOCR and save as text files
def extract_text_from_folder_and_save(input_folder, output_folder):
    reader = easyocr.Reader(['en'])  # Specify languages as needed
```

```
    # Create output folder if it doesn't exist
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    pdf_files = [f for f in os.listdir(input_folder) if f.endswith('.pdf')]
    for pdf_file in pdf_files:
        pdf_path = os.path.join(input_folder, pdf_file)
        text = extract_text_from_pdf(pdf_path)

        # Save extracted text to a text file in output folder
        text_file_path = os.path.join(output_folder, f"{os.path.splitext(pdf_file)[0]}.txt")
        with open(text_file_path, 'w', encoding='utf-8') as f:
            f.write(text)

        print(f"Extracted text from {pdf_file} and saved as {os.path.basename(text_file_path)}")

if __name__ == "__main__":
    input_folder = '/content/pdf'   # Replace with path to your input folder containing PDFs
    output_folder = '/content/output' # Replace with path to your output folder for saving text
files
    extract_text_from_folder_and_save(input_folder, output_folder)
```

Code Snippet.1: Easy OCR

2. VECTOR DATABASE:

Weaviate :

- ➔ **Installations:**

  ```
  pip install weaviate-client openai tiktoken langchain
  sentence-transformers transformers > /dev/null
  pip install -U langchain-community
  ```

- ➔ **Import Libraries:**

  ```
  from langchain.embeddings import HuggingFaceEmbeddings,
  SentenceTransformerEmbeddings
  from langchain.embeddings.openai import OpenAIEmbeddings
  from langchain.text_splitter import RecursiveCharacterTextSplitter
  import os
  from langchain.vectorstores import Weaviate
  ```

- ➔ Weaviate vector search engine along with the Hugging Face embeddings to create a searchable database of text paragraphs. It utilizes various libraries, including LangChain for embeddings, sentence-transformers for creating vector

embeddings, and Weaviate for managing the vector database and performing searches.

- ➔ Text Splitting Function:This function reads a text file and splits its content into smaller chunks of approximately 500 characters, with a 15-character overlap between chunks.
- ➔ Embedding Initialization:This sets up a Weaviate client with embedded options and an API key for Hugging Face. It then deletes any existing schema and retrieves the current schema.
- ➔ Schema Creation:This code creates a schema for Weaviate, defining a class called Paragraph with a single property content, which will be vectorized using the Hugging Face model.
- ➔ Text Splitting and Document Creation:This splits the text from the specified file into smaller chunks and stores them in the mail_docs list.
- ➔ Batch Ingestion:This code uses batch ingestion to add each chunk of text as a data object to the Weaviate database.
- ➔ Querying:This performs a similarity search on the stored paragraphs, looking for the concept "Thermocouple" and limiting the results to 2. The results are then printed in a JSON format.
- ➔ Vector Store Search:This initializes a Weaviate vector store and performs a similarity search for the term "Thermocouple".

```
import os
def process_text_files_in_folder(folder_path):
    all_docs = []
    for filename in os.listdir(folder_path):
        if filename.endswith('.txt'):  # Check if the file is a text file
            file_path = os.path.join(folder_path, filename)  # Create full file path
            print(f"Processing file: {file_path}")  # Optional: Print the file being processed
            docs = get_text_splits(file_path)  # Get text splits for the file
            all_docs.extend(docs)  # Add the splits to the all_docs list
    return all_docs
# Example usage
folder_path = '/content/pdfs'  # Replace with your folder path
all_text_splits = process_text_files_in_folder(folder_path)
print(len(all_text_splits))
def get_text_splits(text_file):
  with open(text_file,'r') as txt:
    data = txt.read()
  textSplit = RecursiveCharacterTextSplitter(chunk_size=500,
                                             chunk_overlap=15,
                                             length_function=len)
  doc_list = textSplit.split_text(data)
  return doc_list
```

```
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
print(embeddings)
import weaviate
from weaviate.embedded import EmbeddedOptions

client = weaviate.Client(
  embedded_options=EmbeddedOptions(),
  additional_headers={
      "X-HuggingFace-Api-Key": "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"
   }
)
client.schema.delete_all()
client.schema.get()
client.schema.create(schema)
mail_docs = get_text_splits("/content/extracted_text.txt")
with client.batch as batch:
   batch.batch_size=5
   for i, d in enumerate(mail_docs):
      properties = {
         "content": d,
      }

      client.batch.add_data_object(properties, "Paragraph")
nearText = {"concepts": ["Thermocouple"]}
result = (
   client.query
   .get("Paragraph", ["content"])
   .with_near_text(nearText)
   .with_limit(2)
   .do()
)
import json
print(json.dumps(result, indent=4))
vectorstore = Weaviate(client,
                  "Paragraph","content")
vectorstore.similarity_search("Thermocouple",k=1)
```

Code Snippet.2: Weaviate

3. SENTENCE TRANSFORMER:

all-MiniLM-L6-v2

The all-MiniLM-L6-v2 is a compact, pre-trained model from the Sentence-Transformers library. It's designed for producing sentence embeddings, capturing the semantic meaning of sentences or texts. Despite its small size, it offers robust performance, making it ideal for applications where resources are limited. In this project, it's used to convert extracted text from PDFs into dense vector embeddings. These embeddings are then stored in a vector database (VectorDB), allowing the system to efficiently retrieve and display relevant text chunks based on user queries, enhancing search accuracy.

COMPARATIVE STUDY:

Feature	**Chroma DB**	**Weaviate**	**FAISS**
Type	Open-source vector database	Open-source vector database	Open-source library for vector similarity search
General Efficiency	ChromaDB is designed to be a user-friendly interface with a focus on simplifying vector storage and retrieval.	Weaviate offers a user-friendly interface with built-in support for a variety of data types, including text and multimedia	While extremely fast, it requires careful tuning and may not support as complex querying mechanisms out-of-the-box as Weaviate.
Use Cases	AI applications, semantic search, recommendations	Semantic search, knowledge graphs, AI models	High-dimensional data retrieval, similarity search
Flexibility	Impressive ease of use	Offers a graphQL based API providing efficient interactions	It can handle vast amount of data with efficiency
Schema architecture	It does not require schema inference	Its Schema inference features automates the process of defining data structures	It does not require schema inference
Limitations	While it’s easy to use, it may not offer the same level of customization and control as other options.	Its setup and initial configuration might be more complex compared to ChromaDB, especially for beginners.	Faiss is a low-level library and requires more manual setup, making it less user-friendly for those unfamiliar with its API.

Table.1: Comparison of Vector DB

OVERALL EFFICIENCY:

ChromaDB offers a user-friendly and cloud-native solution, optimized for quick deployment and moderate scalability. It excels in ease of use, especially for small to medium-sized applications that require seamless integration with common ML models. While it may not match the advanced scalability and search speed of Faiss or the complex querying capabilities of Weaviate, ChromaDB strikes a balance between simplicity and performance. Its strength lies in providing efficient vector storage and retrieval for applications where ease of setup and integration outweigh the need for extreme-scale processing or custom search configurations.

4. USER INTERFACE:

STREAMLIT:

Streamlit is a Python library that simplifies the creation of interactive web applications for data science and machine learning. It provides an easy way to build web-based interfaces for Python code without requiring extensive knowledge of web development technologies like HTML, CSS, or JavaScript.

Key Features of Streamlit:

- **Simple API**: Streamlit uses a straightforward API that allows you to create user interfaces and display data with just a few lines of code.
- **Real-time Updates**: It supports real-time updates, so changes in the code or user input are reflected instantly in the web app.
- **Widgets**: Provides a variety of widgets like buttons, sliders, text inputs, and file uploaders to interact with users.
- **Data Display**: Easily display data using built-in functions for tables, charts, and graphs.
- **Customization**: While it's simple to use, it also offers ways to customize the look and feel of the application.

Development Phase:

Streamlit (st): Utilized to create the interactive web application. It facilitates converting PDF byte data into images and extracting text from these images using OCR. PIL Image: Handles image processing tasks. Chromadb: Serves as a client for interacting with the vector database.

SentenceTransformer: Transforms text into vector representations. os and tempfile: Assist with file management. RecursiveCharacterTextSplitter: Breaks down text into manageable chunks.

- Importing Streamlit allows you to use its functions and widgets to build the user interface.
- st.title("PDF Text Extraction and Query System") - Sets the title of the Streamlit app, which appears at the top of the web page.

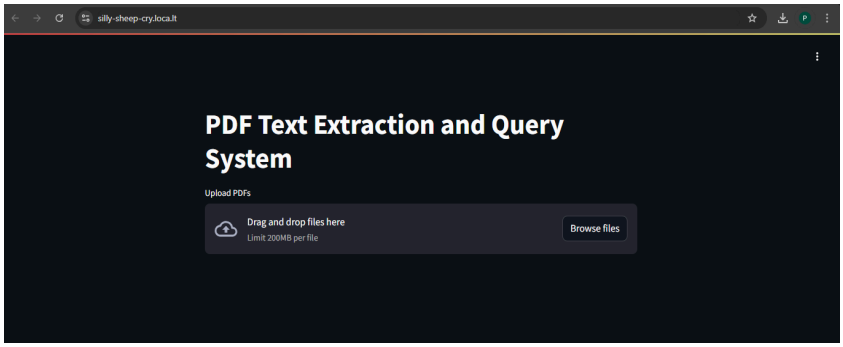

Fig.2: UI- Home Page

- uploaded_files = st.file_uploader("Upload PDFs", type=["pdf"], accept_multiple_files=True) - Provides a file upload widget that allows users to select and upload multiple PDF files. accept_multiple_files=True enables the selection of more than one file at a time.

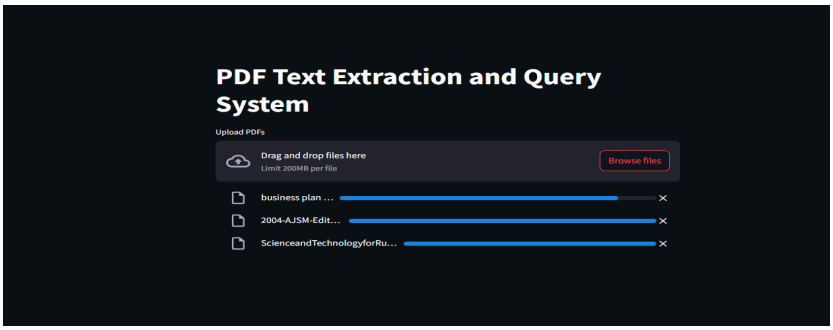

Fig.3: Uploading pdfs

- with st.spinner(f'Extracting text from {uploaded_file.name}...'): pdf_bytes = uploaded_file.read() - Uses st.spinner to show a loading spinner while the text extraction process is ongoing. This gives feedback to users that their file is being processed.

- st.success(f"Text extraction from {uploaded_file.name} completed.") - Displays a success message once the text extraction from a PDF file is completed.

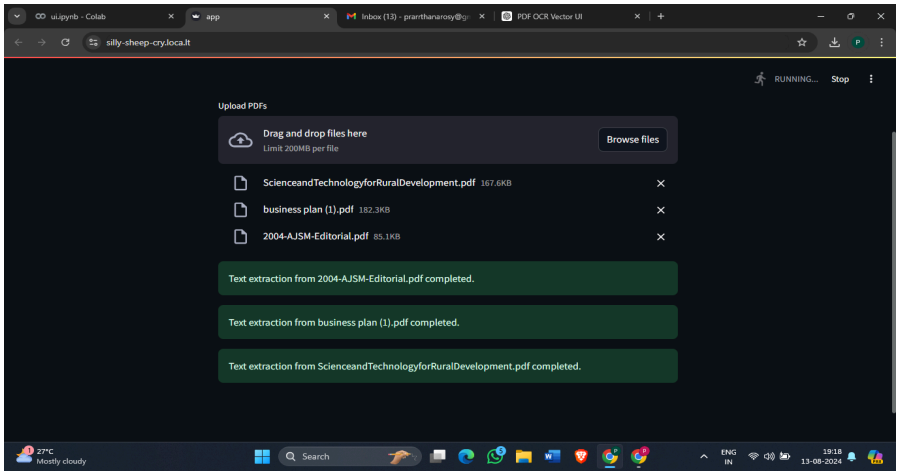

Fig.4: Extraction and storage of text in Vector DB

- query = st.text_input("Enter your query:") - Provides a text input box for users to enter their queries. This allows users to interact with the application by inputting questions or search terms.

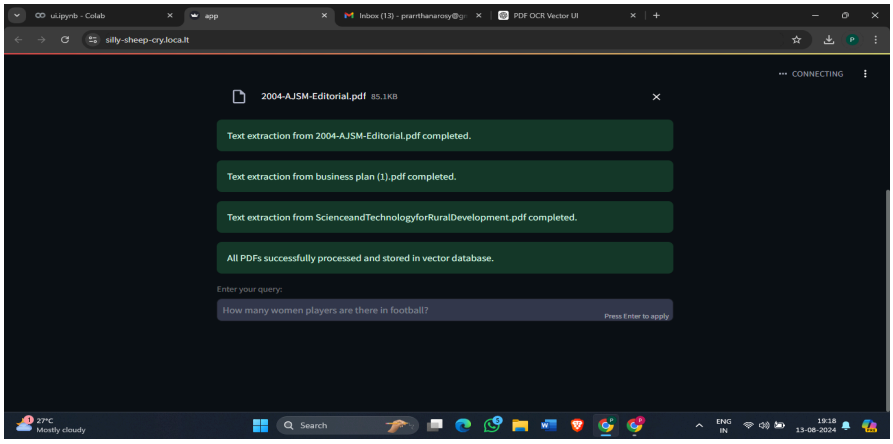

Fig.5: Querying pdfs

- st.write("Query Results:") for result in results['documents']: st.write(result) - Displays the results of the query. st.write is used to show the query results on the web page. This function can handle various data types, including text, data frames, and charts.

RESULT:

We utilized OCR to extract text from uploaded PDFs and stored the information in a vector database. When a user submits a query, the model retrieves the most relevant information from the vector database and presents it as the result.

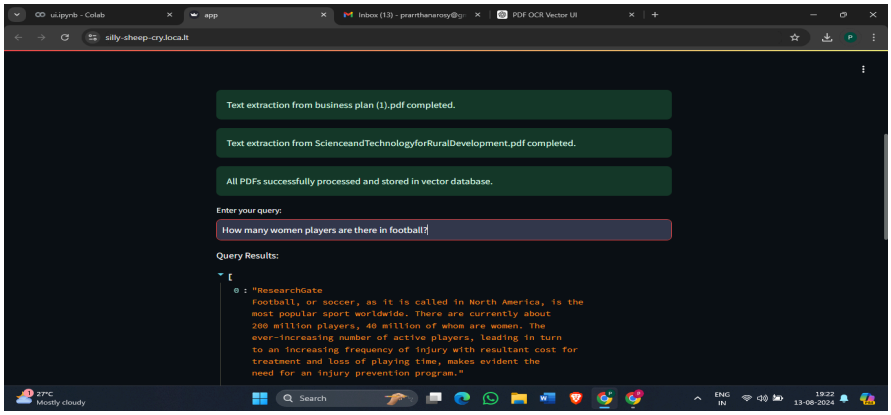

Fig.6: Result for the query

FUTURE WORKS:

Advanced Query System

- Natural Language Understanding: Incorporate more sophisticated NLP models to understand and respond to user queries more accurately, including handling synonyms, paraphrases, and context.
- Semantic Search: Implement semantic search capabilities to return more relevant results based on the meaning of the query rather than just keyword matching.

Document Management Features

- Metadata Extraction: Extract and store metadata (like author, title, and creation date) from PDFs, allowing for more refined search and filtering options.
- Document Categorization: Automatically categorize and tag documents based on their content, aiding in more organized and efficient retrieval.

Support for Additional File Formats

- Multi-Format Support: Extend the system to support other document formats like Word, Excel, and images, broadening the range of files users can upload and query.

Integration with Other Systems

- API Development: Develop an API for external applications to interact with the system, allowing for integration with other software solutions.
- Integration with CMS: Integrate the system with content management systems (CMS) to facilitate automatic processing and querying of newly uploaded documents.

CONCLUSION:

The DocQuest project demonstrates a robust solution for transforming unstructured PDF documents into searchable and analyzable content. Leveraging state-of-the-art technologies like Optical Character Recognition (OCR), vector databases and this system offers a comprehensive pipeline for text extraction, chunking, embedding, and querying. It is designed with user accessibility in mind, providing a straightforward interface that allows users to upload multiple PDFs, extract text, and perform complex searches with ease. This project successfully bridges the gap between static, unsearchable documents and dynamic, searchable content. It opens up new possibilities for document management, data retrieval, and knowledge discovery, making it a valuable tool for businesses, researchers, and any domain where document processing is essential. Continued development and enhancement of this system will further solidify its utility, ensuring it remains at the cutting edge of document processing technology.

REFERENCES:

1. Kartik Joshi, Harshal Arolkar, Comparative Analysis of Outcomes of Tesseract OCR for Different Languages.2024 5th International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)
2. Deepa Rani; Rajeev Kumar; Naveen Chauhan,Study and Comparison of Vectorization Techniques Used in Text Classification, 2022 13th International Conference on Computing Communication and Networking Technologies (ICCCNT)

3. Mohammad Khorasani ,Mohamed Abdou ,Javier Hernández Fernández, Web Application Development with Streamlit.
4. Sentence Transformers - https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
5. Ly Pichponreay; Jin-Hyuk Kim; Chi-Hwan Choi; Kyung-Hee Lee; Wan-Sup Cho Smart answering Chatbot based on OCR and Overgenerating Transformations.