CREATE AN INTERACTIVE EDUCATIONAL CHATBOT

USING STREAMLIT AND GOOGLE GEMINI LLM

Project Report Submitted to

RAJALAKSHMI EDUVERSE

Submitted by

SHEHINA K

CERTIFICATION IN DATA SCIENCE

Under the Guidance of

Mr.Rahul Ramesh
(Master of Engineering in Mechatronics,
Post Graduate Program in Analytics and Artifical Intelligence)

RAJALAKSHMI EDUVERSE

Chennai – 600004
(2023-2024)

Acknowledgements

First of all , I thank the Almighty and my parents from the bottom of my heart for bestowing their blessings on me throughout the course of this project.

I would like to express my sincere gratitude to my trainer, Mr. Rahul Ramesh, for his invaluable guidance and support throughout the development of my Generative AI project. His expertise in data science was instrumental in helping me to understand the complexities of the technology and apply it effectively in my project.

From the very beginning, Mr. Rahul has been a constant source of inspiration and motivation. His comprehensive lectures and practical sessions provided me with a solid foundation in machine learning, data preprocessing, and model evaluation techniques. His ability to simplify complex concepts and his patience in addressing all my queries have significantly enhanced my understanding and skills.

Abstract

This project introduces an interactive educational chatbot aimed at assisting high school and college students in overcoming obstacles related to accessing and understanding intricate information from academic texts. The chatbot is specifically designed to facilitate user interaction with Environmental Science content. Using Streamlit, the application allows students to upload Environmental Science PDFs, which are then processed to extract and segment text into manageable sections.

Text segments undergo embedding using Google Generative AI Embeddings and are stored in a FAISS vector store for efficient semantic search capabilities. Powered by the "gemini-pro" conversational AI model, the chatbot provides tailored responses to user queries based on the content extracted from the uploaded PDFs. This intuitive interface supports seamless PDF uploads and dynamic querying, aiming to enhance students' comprehension and engagement with Environmental Science academic materials.

CONTENT

CHAPTER I
INTRODUCTION

1.1 Large Language Model:

LLM, or Large Language Models, refer to advanced artificial intelligence models like GPT (Generative Pre-trained Transformer) that are trained on vast amounts of text data to understand and generate human-like language.

The Process of Developing and Utilizing Large Language Models (LLMs):

1. Data Collection and Preprocessing

Data Collection:

- **Source Identification:** Identify diverse and extensive sources of text data, such as books, articles, websites, and scientific papers.
- **Volume:** Collect large volumes of data to provide the model with a rich and varied linguistic foundation.

Preprocessing:

- **Cleaning:** Remove noise from the data, including HTML tags, special characters, and irrelevant information.
- **Normalization:** Standardize text by converting it to lowercase, removing punctuation, and handling contractions.
- **Tokenization:** Split the text into tokens (words or subwords) that the model can process.

2. Model Architecture Design

Choosing the Architecture:

- **Transformer Model:** Most LLMs are based on the transformer architecture, which excels at handling sequential data and capturing long-range dependencies.
- **Variants:** Select specific transformer variants like GPT (Generative Pre-trained Transformer), BERT (Bidirectional Encoder Representations from Transformers), or others based on the application.

Configuring Hyperparameters:

- **Layer Size:** Determine the number of layers, heads, and hidden units in the model.
- **Training Parameters:** Set learning rates, batch sizes, and optimization algorithms.

3. Training the Model

Pre-training:

- **Unsupervised Learning:** Train the model on large text corpora to learn language patterns, grammar, and context without specific task instructions.
- **Compute Resources:** Utilize powerful hardware, such as GPUs or TPUs, to handle the extensive computational demands.

Fine-tuning:

- **Supervised Learning:** Fine-tune the pre-trained model on task-specific datasets (e.g., question answering, summarization) to adapt it to particular applications.
- **Evaluation:** Continuously evaluate the model's performance on validation sets to avoid overfitting.

4. Model Deployment

Serving Infrastructure:

- **Scalability:** Deploy the model on scalable infrastructure to handle varying loads of user queries.
- **APIs:** Develop APIs to facilitate easy integration of the LLM into applications.

Latency and Throughput:

- **Optimization:** Optimize the model for faster inference times to ensure responsiveness.
- **Caching:** Implement caching mechanisms to reduce redundant computations for frequently asked queries.

5. Utilization in Applications

Integration:

- **Chatbots:** Integrate LLMs into chatbots to provide intelligent and context-aware responses.
- **Content Generation:** Use LLMs for generating articles, reports, and other written content.
- **Translation:** Employ LLMs in translation services to offer accurate and fluent translations across languages.

User Interaction:

- **Customization:** Allow users to customize and interact with the LLM to suit their specific needs.
- **Feedback Loop:** Collect user feedback to continuously improve the model's performance.

6. Continuous Monitoring and Maintenance

Performance Monitoring:

- **Metrics:** Monitor key performance metrics such as accuracy, latency, and user satisfaction.
- **Error Analysis:** Analyze errors and failures to understand the model's limitations and areas for improvement.

Regular Updates:

- **Retraining:** Periodically retrain the model with new data to keep it updated with the latest information and language usage trends.
- **Bug Fixes:** Address and fix any bugs or issues that arise during deployment.

CHAPTER II

2.1 Workflow Architecture for a Chatbot

1. PDF File Upload and Processing

- **File Upload**: Users can upload multiple PDF files through the Streamlit sidebar.
- **Text Extraction**: Extracts text from the uploaded PDF files using PdfReader from PyPDF2.
- **Text Chunking**: Splits the extracted text into manageable chunks using RecursiveCharacterTextSplitter to handle large documents efficiently.

2. Vector Store Creation

- **Embedding Generation**: Uses GoogleGenerativeAIEmbeddings to create embeddings for the text chunks.
- **Vector Store**: Creates a FAISS vector store from these embeddings, enabling efficient similarity searches. The vector store is saved locally. During the deserialization process of the FAISS vector store, if an error occurs, the parameter allow_dangerous_deserialization=True is added to ensure proper loading and processing.

3. Conversational Chain

- **Prompt Template**: Defines a prompt template to instruct the AI model on how to respond to questions using the context provided.
- **Model Configuration**: Configures ChatGoogleGenerativeAI to use the gemini-pro model with a specified temperature setting to control response variability.
- **QA Chain**: Loads a question-answering chain with the model and prompt template, facilitating the interaction between user questions and the extracted PDF content.

4. User Interaction

- **Question Input**: Users can input their questions in a text box.
- **Similarity Search**: Upon receiving a question, the application performs a similarity search in the FAISS vector store to retrieve relevant text chunks.
- **Answer Generation**: The retrieved chunks and the user question are fed into the conversational chain to generate a detailed answer.

5. Streamlit Interface

- **Main Page**: Contains the question input field and displays responses.
- **Sidebar**: Allows PDF file uploads and initiates the processing of uploaded files.

CHAPTER III
CODING

Platform: Visual Studio and python code

3.1).env file

```
GOOGLE_API_KEY = "AIzaSyBqeBo99qjfL_c8To0bfzTJ4lnEsX_NU6Q"
```

3.2)requiremet.txt file

```
streamlit
google-generativeai
python-dotenv
langchain
PyPDF2
faiss-cpu
fastapi
langchain_google_genai
```

3.3)app.py file

```
# Import necessary libraries and modules
import streamlit as st
from PyPDF2 import PdfReader
from langchain.text_splitter import RecursiveCharacterTextSplitter
import os
import hashlib
from langchain_google_genai import GoogleGenerativeAIEmbeddings



import google.generativeai as genai
from langchain.vectorstores import FAISS
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.chains.question_answering import load_qa_chain
from langchain.prompts import PromptTemplate
from dotenv import load_dotenv

# Load environment variables
```

```
load_dotenv()
api_key = os.getenv("GOOGLE_API_KEY")
if api_key:
    genai.configure(api_key=api_key)
else:
    st.error("Google API key not found. Please check your .env file.")

# Function to extract text from PDF
def get_pdf_text(pdf_docs):
    text = ""
    for pdf in pdf_docs:
        pdf_reader = PdfReader(pdf)
        for page in pdf_reader.pages:
            text += page.extract_text()
    return text

# Function to split text into chunks
def get_text_chunks(text):
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=10000, chunk_overlap=1000)
    chunks = text_splitter.split_text(text)
    return chunks

# Function to create and save the FAISS vector store
def get_vector_store(text_chunks):
    embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")
    vector_store = FAISS.from_texts(text_chunks, embedding=embeddings)
    vector_store.save_local("faiss_index")


# Function to get the conversational chain
def get_conversational_chain():
    prompt_template = """
    Answer the question based on your comprehensive knowledge of science,
    incorporating insights from the trained PDF documents.
    Provide a detailed and accurate explanation without errors. Include
    relevant examples or definitions to enhance understanding.
    If needed, ask clarifying questions to ensure the response is precise and
    Informative.
    If the answer is not in the provided context just say, "answer is not
    available in the context",
    don't provide the wrong answer\n\n
    Context:\n{context}?\n
    Question:\n{question}\n
    Answer:
    """

    model = ChatGoogleGenerativeAI(model="gemini-pro", temperature=0.3)
    prompt = PromptTemplate(template=prompt_template,
```

```
    input_variables=["context", "question"])
    chain = load_qa_chain(model, chain_type="stuff", prompt=prompt)
    return chain

# Function to handle user input and query the vector store
def user_input(user_question):
    embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")
    new_db = FAISS.load_local("faiss_index", embeddings,
    allow_dangerous_deserialization=True)
    docs = new_db.similarity_search(user_question)

    chain = get_conversational_chain()
    response = chain(
        {"input_documents": docs, "question": user_question},
         return_only_outputs=True )

    st.write("Reply: ", response["output_text"])

# Main function to handle the Streamlit app
def main():
    st.set_page_config(page_title="Chat PDF")
    st.header("Environmental Science 📑📚")

    user_question = st.text_input("Ask a Question from the PDF Files")

    if user_question:
        user_input(user_question)

    with st.sidebar:
        st.title("Menu:")
        pdf_docs = st.file_uploader("Upload your PDF Files and Click on the
         Submit & Process Button", accept_multiple_files=True)

      if st.button("Submit & Process"):
            if pdf_docs:
                with st.spinner("Processing..."):
                    raw_text = get_pdf_text(pdf_docs)
                    text_chunks = get_text_chunks(raw_text)
                    get_vector_store(text_chunks)
                    st.success("Processing completed!")
            else:
                st.warning("Please upload at least one PDF file.")

if __name__ == "__main__":
    main()
```

CHAPTER – IV

DESCRIPTION ABOUT CODE

4.1 Detail Description about Code

1)Importing Libraries and Setting Up Environment:

1. **streamlit**:
   - A Python library for building interactive web applications. It simplifies the process of creating and deploying data-driven applications.
2. **PdfReader from PyPDF2**:
   - Allows reading and extracting text from PDF files. PdfReader is part of the PyPDF2 library, commonly used for PDF manipulation in Python.
3. **RecursiveCharacterTextSplitter**:
   - This utility splits large text into manageable chunks based on characters. It aids in preprocessing text for further analysis or embedding.
4. **os**:
   - The standard Python module for interacting with the operating system. It provides functions to manipulate file paths, environment variables, and execute system commands.
5. **GoogleGenerativeAIEmbeddings**:
   - Utilizes Google's Generative AI to generate embeddings (numeric representations) for text chunks. Embeddings are crucial for tasks like semantic similarity and text understanding.
6. **genai**:
   - This module likely configures settings or interacts with Google's Generative AI capabilities. It provides an interface to configure and use Google's AI services effectively.
7. **FAISS**:
   - A library designed for efficient similarity search and clustering of dense vectors. FAISS is often used in conjunction with embeddings to perform fast similarity searches over large datasets.
8. **ChatGoogleGenerativeAI**:
   - Provides an interface to interact with Google's Generative AI specifically for conversational purposes. It likely encapsulates functionalities for dialogue generation or interaction.
9. **load_qa_chain**:
   - A function that loads a question-answering chain from langchain.chains.question_answering. It sets up a pipeline for answering questions based on input context and questions.

10. **PromptTemplate**:
    - A template framework from langchain.prompts for constructing prompts or input templates. It helps structure how questions and context are presented to the conversational AI model.
11. **load_dotenv**:
    - A utility to load environment variables from a .env file into the script's environment. It enhances security by keeping sensitive information like API keys separate from the codebase.

2)Environment Variables

- **GOOGLE_API_KEY**:
    - Stored securely in a .env file, the Google API key (GOOGLE_API_KEY) is accessed using os.getenv("GOOGLE_API_KEY"). This key is essential for authenticating and accessing Google's services such as Generative AI.

1. **Text Extraction from PDFs**:

    - The get_pdf_text(pdf_docs) function extracts text from uploaded PDF documents (pdf_docs). It iterates through each page of each PDF using PdfReader from PyPDF2 and concatenates the text into a single string.

3)Text Chunking:

- After extracting text from PDFs, get_text_chunks(text) splits the concatenated text into smaller chunks using RecursiveCharacterTextSplitter from langchain. Chunks are created to be 10,000 characters long with a 1,000 character overlap to facilitate efficient processing.

4)Creating and Saving FAISS Vector Store:

- get_vector_store(text_chunks) creates embeddings for the text chunks using GoogleGenerativeAIEmbeddings from langchain_google_genai. These embeddings are then stored in a FAISS vector store (vector_store) using FAISS from langchain.vectorstores, and saved locally as "faiss_index".

5)Setting Up Conversational AI Chain:

- get_conversational_chain() defines a conversational chain using ChatGoogleGenerativeAI from langchain_google_genai. It initializes a model (gemini-pro) and sets a prompt template (prompt_template) for generating responses based on user questions and the embedded text chunks.

6)Handling User Input and Querying:

- user_input(user_question) processes user queries by first loading the FAISS vector store (new_db) created earlier. It performs a similarity search (similarity_search) on the vector store using the user's question (user_question).
- The conversational chain (chain) defined earlier is then used to generate a response based on the retrieved documents (docs) and the user's question. The response is displayed using Streamlit's st.write.

7)Main Streamlit Application Setup:

- main() sets up the Streamlit application interface. It configures the page title and header, allowing users to input questions related to Environmental Science PDFs.
- The sidebar provides options to upload PDF files (pdf_docs) and initiate the processing pipeline (Submit & Process button). Upon submission, PDFs are processed for text extraction, chunking, embedding, and storing in FAISS.

8)Execution:

- The script checks if it's being run directly (if __name__ == "__main__":) and calls main() to start the Streamlit application, where users can interact with the chatbot interface to ask questions based on the processed Environmental Science PDFs.

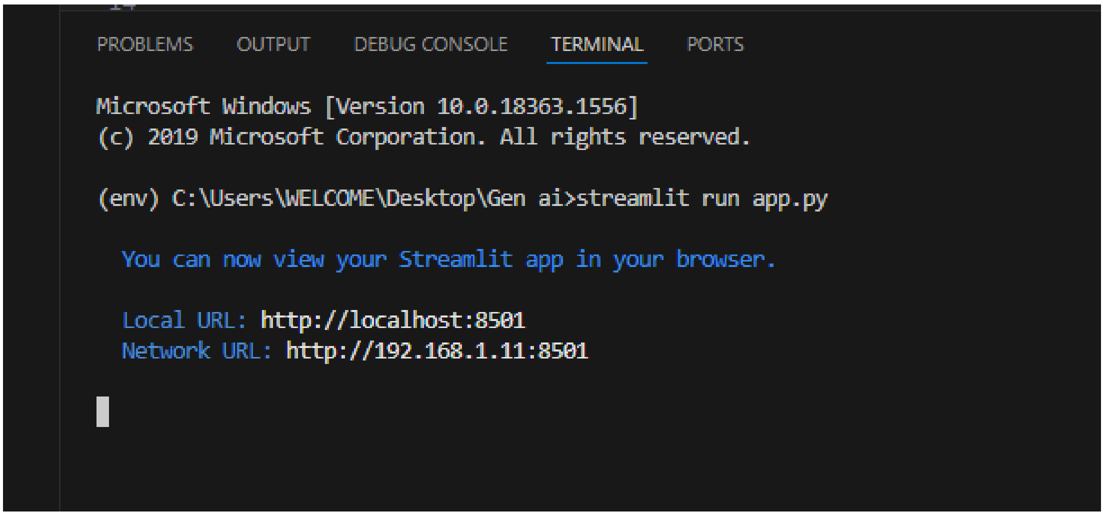
CHAPTER – V

OUTPUT

5.1 Results and Outputs

Streamlit Command and Local Server:

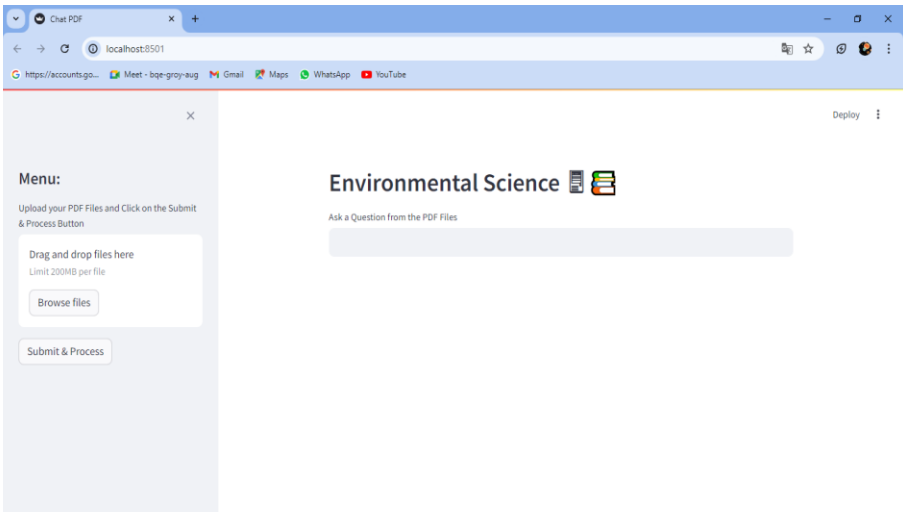
- streamlit run app.py initiates the Streamlit development server.
- This server processes your Python code containing Streamlit functions that define the app's layout, content, and interactivity.
- The server creates a temporary web application based on our code.



Web Browser Access:

- The server provides a URL (usually http://localhost:8501 by default) where the Streamlit app is hosted.
- Opening this URL in our web browser displays the interactive web application.

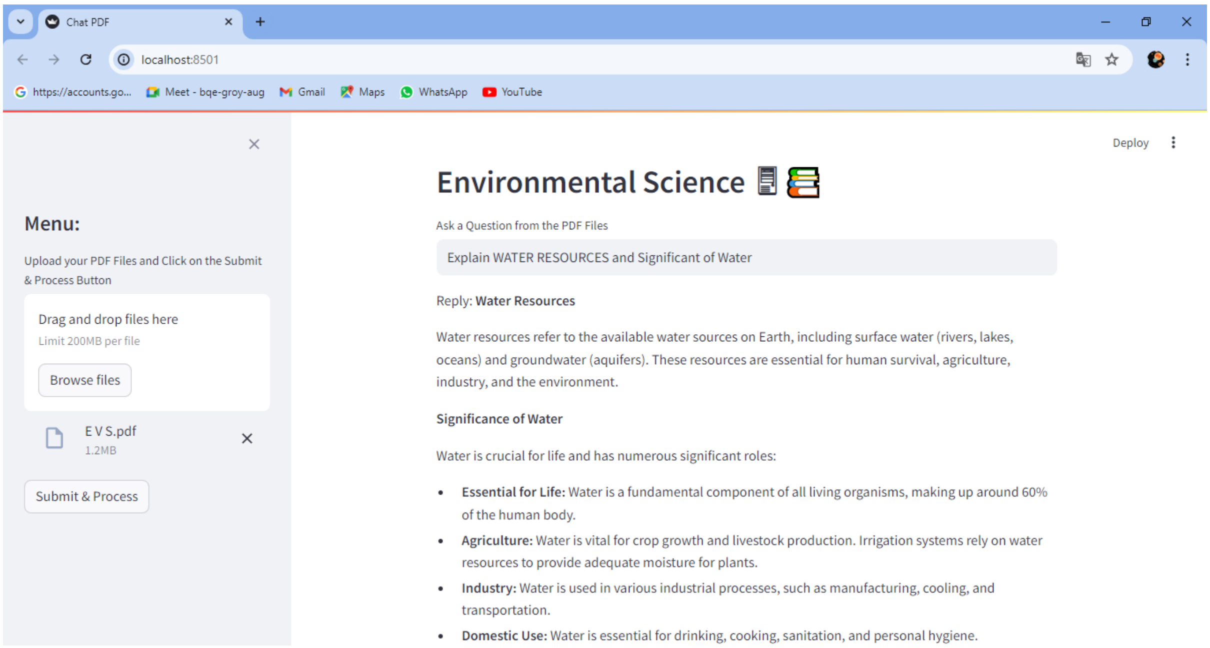
Interactive Educational Chatbot



Overall Output:

The combined effect is an interactive web application that allows users to:

- **Provide input:** Users can enter natural language queries or upload PDFs for analysis.
- **Interact with AI:** The app leverages Google's Generative AI to process the input.
- **Receive responses:** The app displays summaries, insights, or other relevant information derived from the processed PDF content, creating a chatbot-style dialogue.



CHAPTER – VI

Evaluation Method:

To evaluate the performance of the interactive educational chatbot in the context of Environmental Science, I followed a systematic approach:

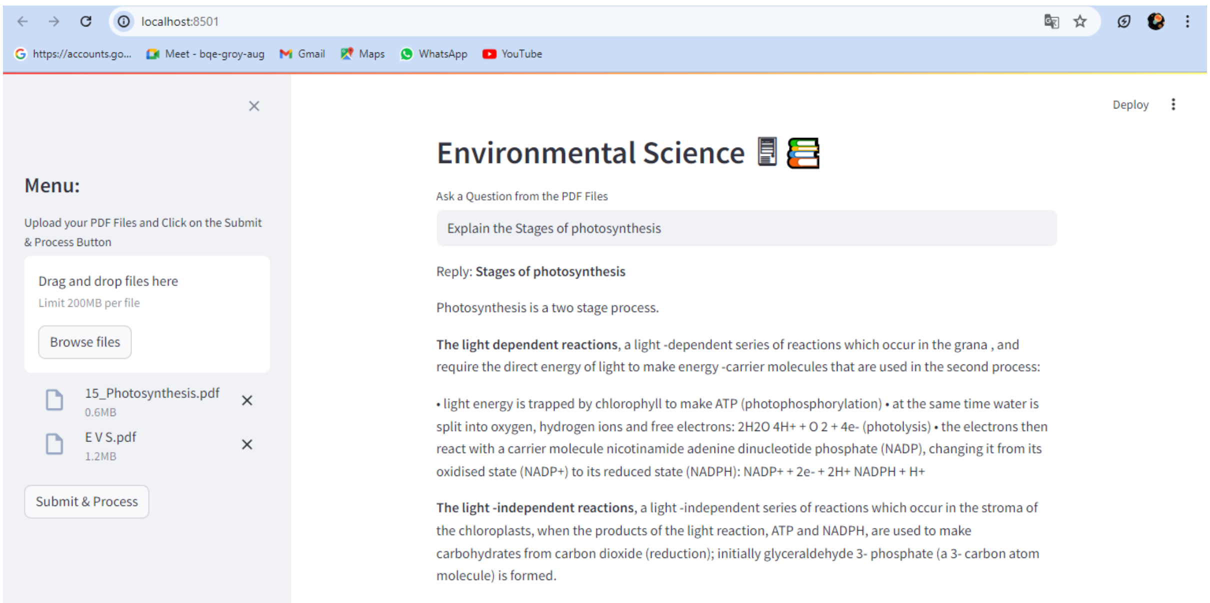
1. **Random Selection of Questions:**
    - I randomly selected questions from an uploaded PDF document that covered a broad range of topics within Environmental Science, including Water Resource Management, Natural Resources, Ecosystems, Biodiversity and Conservation, Pollution and its Factors, Social Issues, and Human Population.

2,Assessment Criteria:

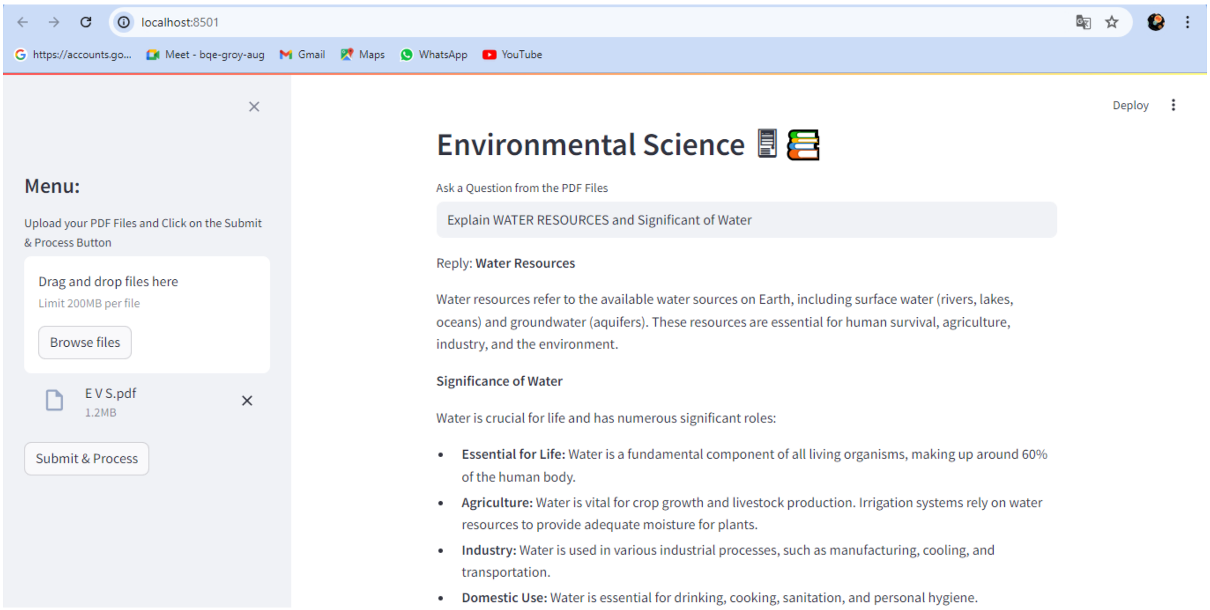
- The chatbot's responses were assessed based on two main criteria:
    - **Accuracy:** The correctness of the information provided by the chatbot.
    - **Comprehensiveness:** The thoroughness and depth of the information provided, ensuring it adequately covered the topics in question.

Interactive Environmental Science Chatbot Interface:

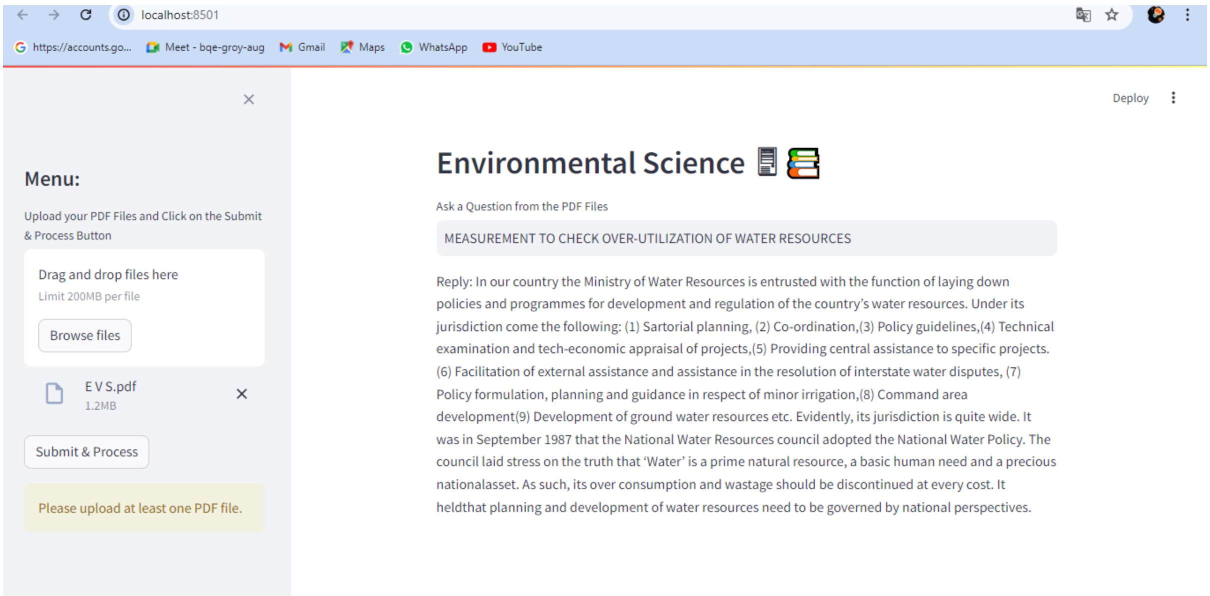
1.Explain the Stages of Photosynthesis?



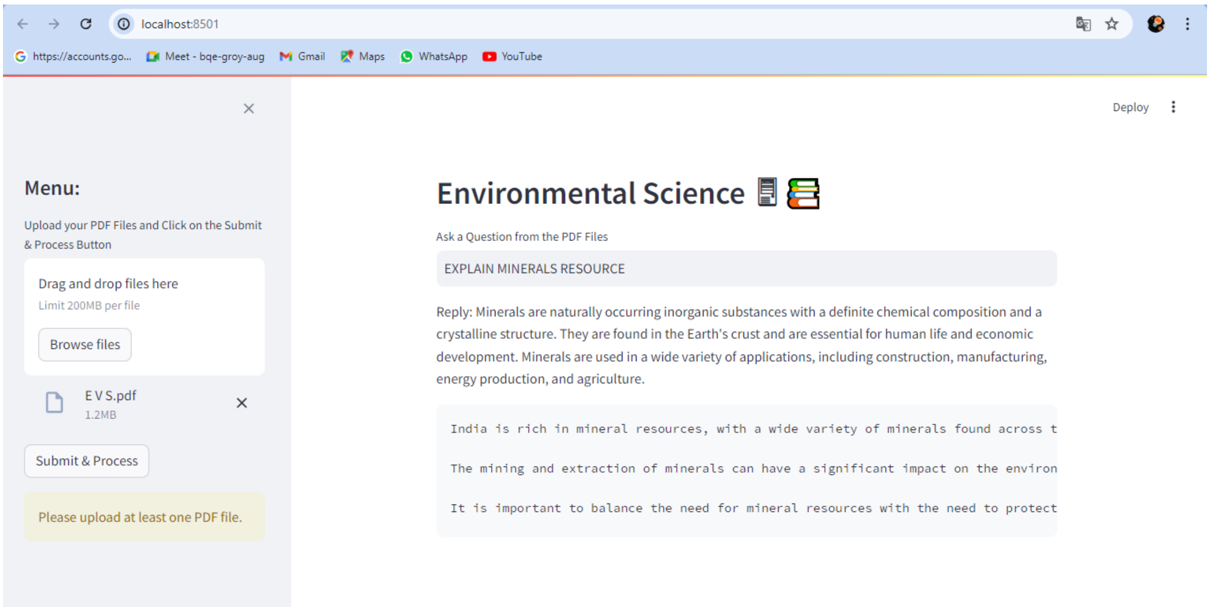
2. Explain Water resources and Significant of water?



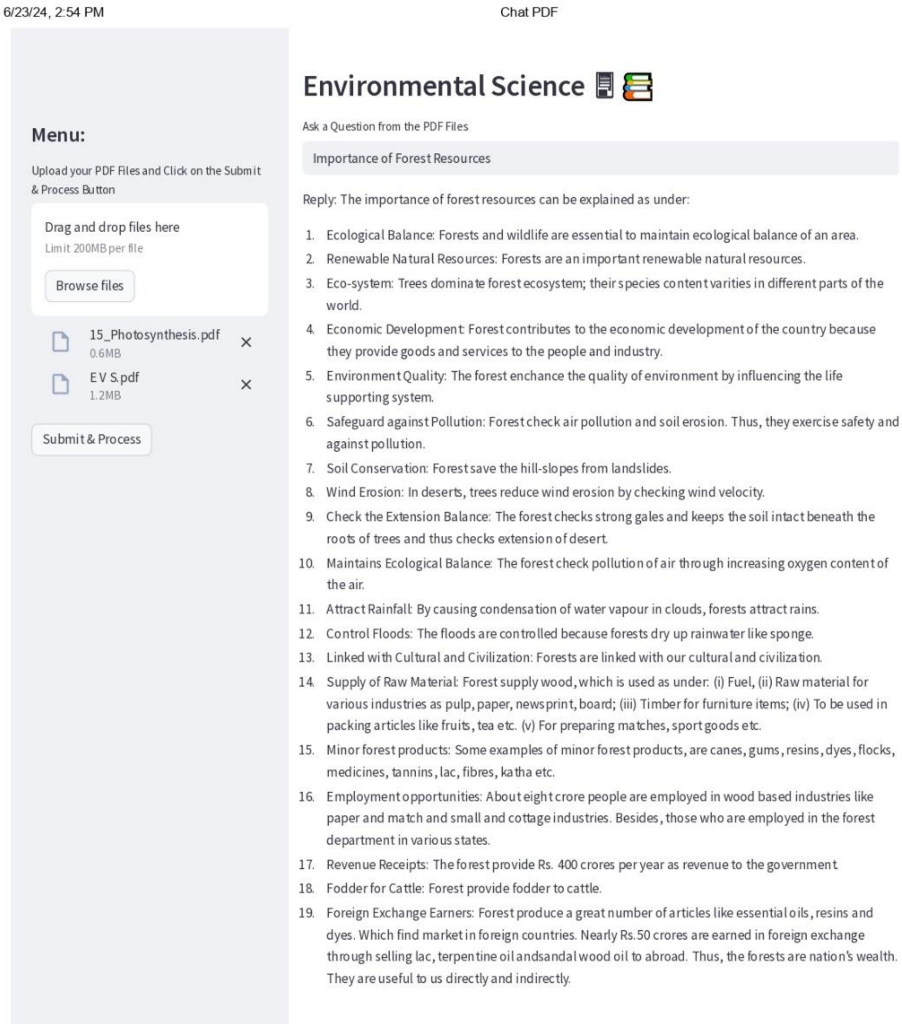
3.Measures to check the utilisation of water resources?



4.Explain mineral resources?



5. Importance of forest resources?



CHAPTER – VII

Applications and Limitations of Large Language Models (LLMs)

Applications:

1. **Natural Language Processing (NLP):**
    - **Text Generation:** LLMs can generate coherent and contextually relevant text, useful in creative writing, content creation, and automated storytelling.
    - **Text Summarization:** They can summarize large documents or articles, providing concise information quickly.
    - **Translation:** LLMs can translate text between multiple languages, enhancing communication and accessibility.
    - **Sentiment Analysis:** They can analyze and interpret the sentiment of a given text, useful in market analysis, customer feedback, and social media monitoring.
2. **Conversational AI:**
    - **Chatbots:** LLMs power intelligent chatbots that can engage in meaningful and context-aware conversations with users, providing customer support, information retrieval, and personal assistance.
    - **Virtual Assistants:** They enhance the capabilities of virtual assistants like Siri, Alexa, and Google Assistant by improving their understanding and response accuracy.
3. **Education:**
    - **Tutoring Systems:** LLMs can provide personalized tutoring, answer questions, and explain complex topics in various subjects.
    - **Language Learning:** They aid in language learning by providing practice exercises, translations, and conversation practice.
4. **Healthcare:**
    - **Medical Documentation:** LLMs assist in generating and summarizing medical reports, improving the efficiency of healthcare documentation.
    - **Patient Interaction:** They can interact with patients through chatbots to provide preliminary medical advice and information.
5. **Research:**
    - **Literature Review:** LLMs can assist researchers in summarizing academic papers and extracting relevant information from large datasets.
    - **Data Analysis:** They can help in interpreting complex data and generating hypotheses.
6. **Business:**
    - **Customer Service:** LLMs improve customer service by providing quick and accurate responses to customer inquiries.
    - **Market Analysis:** They analyze market trends and customer sentiment from
    - various sources, aiding in strategic decision-making

.

Limitations:

1. **Accuracy and Reliability:**
    - **Misinformation:** LLMs can sometimes generate inaccurate or misleading information, which can be problematic in critical applications like healthcare or legal advice.
    - **Bias:** They can inherit and propagate biases present in the training data, leading to biased or unfair outcomes.
2. **Context Understanding:**
    - **Limited Understanding:** LLMs might not fully grasp the context or nuances of complex topics, leading to oversimplified or incorrect responses.
    - **Ambiguity:** They may struggle with ambiguous queries or those requiring deep understanding and reasoning.
3. **Ethical Concerns:**
    - **Privacy:** The use of LLMs in applications involving sensitive data raises privacy concerns.
    - **Manipulation:** They can be used to generate persuasive but misleading content, contributing to the spread of misinformation.
4. **Resource Intensive:**
    - **Computational Resources:** Training and deploying LLMs require significant computational power and resources, making them expensive and environmentally taxing.
    - **Maintenance:** Continuous updating and fine-tuning are necessary to keep the models relevant and accurate, which adds to the maintenance cost.
5. **Scalability:**
    - **Data Handling:** Handling and processing large volumes of data can be challenging and may require specialized infrastructure.
    - **Integration:** Integrating LLMs into existing systems and workflows can be complex and resource-intensive.
6. **Human-AI Interaction:**
    - **Dependence:** Over-reliance on LLMs can lead to reduced human oversight and critical thinking.
    - **Miscommunication:** Misunderstandings between humans and AI can occur, especially if the AI's responses are misinterpreted.

CHAPTER – VIII

Conclusion

In this project, I have successfully developed an interactive educational chatbot using Streamlit and Google Gemini LLM to address the challenges faced by students studying Environmental Science. The chatbot effectively assists in navigating complex information, scientific terminology, and extensive data, providing clear and concise answers to queries related to water resource management, natural resources, ecosystems, biodiversity, pollution factors, social issues, and human population. The key functionalities implemented, such as textual search, question answering, and summarization, have demonstrated their effectiveness in supporting academic studies and saving valuable time. This project represents a significant contribution to enhancing educational resources in Environmental Science and promoting efficient learning methods through technology.