Overview

This is a chatbot that allows users to upload PDF documents (particularly those containing Nepali text), process them into a searchable knowledge base, and query the content with responses provided in either English or Nepali. The application leverages natural language processing (NLP) tools, embeddings, vector stores, and generative AI to provide context-aware answers based on the uploaded document.

Key features include:

- Support for Nepali (Devanagari script) and English text extraction and querying.
- PDF text extraction with fallback mechanisms for robustness.
- Vector-based document retrieval using FAISS and Hugging Face embeddings.
- Bilingual response generation using Google's Gemini model and Google Translate.
- Persistent conversation history and reset functionality.

The application is designed for users who need to interact with Nepali-language documents in a conversational manner.

Dependencies

The application relies on the following Python libraries:

- **streamlit**: Web application framework for the UI.
- **langchain_huggingface**: Provides Hugging Face embeddings for text processing.
- **langchain_community.vectorstores**: Implements FAISS for vector-based document retrieval.
- **langchain.text_splitter**: Splits text into manageable chunks.
- **dotenv**: Loads environment variables from a .env file.
- **os**: Handles file system operations.
- **re**: Regular expressions for text cleaning.
- **google.generativeai**: Google's Gemini model for response generation.
- **PyPDF2**: Fallback PDF text extraction library.
- **deep_translator**: Translation between English and Nepali using Google Translate.
- **tempfile**: Temporary file handling for PDF processing.
- **pdfminer.six**: Primary PDF text extraction library with Unicode support.

Code Structure

1. Imports and Setup

```
import streamlit as st
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter
from dotenv import load_dotenv
import os
import re
import google.generativeai as genai
from PyPDF2 import PdfReader
from deep_translator import GoogleTranslator
import tempfile
```

2. API and Model Initialization

```
# Load environment variables
load_dotenv()
GOOGLE_API_KEY = os.getenv("GOOGLE_API_KEY")

# API Configuration
if not GOOGLE_API_KEY:
    st.error("Google API key is missing. Please configure it in the .env file.")
    st.stop()

genai.configure(api_key=GOOGLE_API_KEY)
model = genai.GenerativeModel("gemini-1.5-flash")
embedding_model = HuggingFaceEmbeddings(model_name="sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2")

# Initialize translators
nepali_translator = GoogleTranslator(source='auto', target='ne')
english_translator = GoogleTranslator(source='auto', target='en')
```

- Ensures the Google API key is available in the .env; otherwise, the app halts with an error.
    - Initializes the Gemini 1.5 Flash model for response generation and a multilingual MiniLM embedding model for text vectorization.

3. Initialising Translators and Session State

```python
# Initialize translators
nepali_translator = GoogleTranslator(source='auto', target='ne')
english_translator = GoogleTranslator(source='auto', target='en')

# Session state
if "conversation_history" not in st.session_state:
    st.session_state.conversation_history = []
if "vectorstore" not in st.session_state:
    st.session_state.vectorstore = None
if "language_preference" not in st.session_state:
    st.session_state.language_preference = "English"
```

- Sets up translators for Nepali and English from google translator.
- Initializes Streamlit session state to sotre conversation history, vector store, and language preference across interactions.

4. Utility Functions

- **Clean_ocr_text(text)**

```python
def clean_ocr_text(text):
    """Enhanced cleaning for Nepali OCR text"""
    # Fix hyphenated line breaks
    text = re.sub(r'-\s+([\u0900-\u097F])', r'\1', text)
    # Remove form feeds and special characters
    text = re.sub(r'\x0c', '', text)
    # Collapse whitespace
    text = re.sub(r'\s+', ' ', text)
    return text.strip()
```

  - Cleans text extracted from PDFs (e.g., removing hyphens, form feeds, and excessive whitespace).
  - Specifically handles Nepali text by reconnecting hyphenated Devanagari characters.

- **is_valid_devanagari_text(text, threshold=0.05)**

```python
def is_valid_devanagari_text(text, threshold=0.05):
    """Check if text contains sufficient Devanagari characters"""
    if not text:
        return False
    # Expanded Devanagari Unicode ranges
    devanagari_chars = re.findall(r'[\u0900-\u097F\u1CD0-\u1CFF\uA8E0-\uA8FF]', text)
    ratio = len(devanagari_chars) / len(text)
    return ratio > threshold
```

  - Validates if the text contains a sufficient proportion of Devanagari characters (Nepali script).
  - Uses a threshold (default 5%) to determine if the text is meaningfully Nepali.

- **extract_text_from_pdf(file)**

```python
def extract_text_from_pdf(file):
    """Improved PDF extraction with Unicode support"""
    try:
        # Primary extraction with pdfminer.six
        from pdfminer.high_level import extract_text as pdfminer_extract_text

        with tempfile.NamedTemporaryFile(delete=False) as tmp:
            tmp.write(file.getvalue())
            tmp_path = tmp.name

        result = pdfminer_extract_text(tmp_path)
        os.unlink(tmp_path)

        # Clean and validate
        cleaned_text = clean_ocr_text(result)
        if is_valid_devanagari_text(cleaned_text):
            return cleaned_text

        # Fallback to PyPDF2
        file.seek(0)
        reader = PdfReader(file)
        texts = []
        for page in reader.pages:
            page_text = page.extract_text() or ''
            texts.append(page_text)

        pyPDF_text = clean_ocr_text("\n".join(texts))
        combined = cleaned_text + "\n" + pyPDF_text
        final_text = clean_ocr_text(combined)

        if not is_valid_devanagari_text(final_text):
            st.warning("Some Nepali characters might not be recognized properly.")
        return final_text

    except Exception as e:
        st.error(f"PDF extraction error: {str(e)}")
        return ""
```

  - Extracts text from a PDF using pdfminer.six as the primary method (better Unicode support).
  - Falls back to PyPDF2 if the primary extraction lacks sufficient Devanagari content.
  - Combines results, cleans the text, and validates Nepali content, issuing warnings if recognition fails.

- **load_document_to_knowledge_base(file)**

```python
def load_document_to_knowledge_base(file):
    """Process PDF and create vector store"""
    text = extract_text_from_pdf(file)
    if text:
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=800,
            chunk_overlap=100,
            separators=[".\n", "।\n", "!\n", "?\n", "\n\n", "\n", " "]  # Nepali separators
        )
        chunks = text_splitter.split_text(text)
        st.session_state.vectorstore = FAISS.from_texts(chunks, embedding=embedding_model)
        st.success(f"{file.name} processed successfully!")
    else:
        st.error("Failed to extract text from PDF")
```

- Processes the extracted text into chunks using a text splitter with Nepali-specific separators (e.g., "।").
- Creates a FAISS vector store from the chunks using the embedding model.

- **retrieve_documents(query, k=2)**

```python
def retrieve_documents(query, k=2):
    """Document retrieval with Nepali support"""
    retriever = st.session_state.vectorstore.as_retriever(search_kwargs={"k": k})
    return retriever.invoke(query)
```

- Retrieves the top k (default 2) most relevant document chunks based on the query using FAISS.

- **translate_text(text, target_lang)**

```python
def translate_text(text, target_lang):
    """Text translation with error handling"""
    try:
        if target_lang == "Nepali":
            return nepali_translator.translate(text)
        return english_translator.translate(text)
    except Exception as e:
        st.error(f"Translation error: {str(e)}")
        return text
```

- Translates text to the target language (Nepali or English) with error handling.

5. Core Logic

```python
def generate_response(query):
    """Context-aware response generation"""
    if not st.session_state.vectorstore:
        return "Please upload a PDF first"

    # Retrieve context
    relevant_docs = retrieve_documents(query)
    context = "\n".join([doc.page_content for doc in relevant_docs])

    # Build prompt
    history = "\n".join(
        [f"User: {msg['query']}\nAI: {msg['response']}"
         for msg in st.session_state.conversation_history[-3:]]
    )

    prompt = f"""
    Answer in {st.session_state.language_preference} using this context:
    Nepali Context: {context}

    Conversation History:
    {history}

    Question: {query}
    Provide a detailed answer based on the document:
    """

    response = model.generate_content(prompt)
    raw_response = response.text

    # Translate if needed
    if st.session_state.language_preference == "Nepali":
        return translate_text(raw_response, "Nepali")
    return raw_response
```

- Generates a response by:
    1. Checking if a vector store exists.
    2. Retrieving relevant document chunks.
    3. Incorporating the last 3 conversation turns as history.

4. Constructing a prompt with context, history, and the query.
5. Using the Gemini model to generate a response.
6. Translating to Nepali if required.

6. Core Logic

```python
# Streamlit UI
st.title("Nepali Chatbot 📄")

# Language selection
lang = st.selectbox("Response Language:", ["English", "Nepali"])
st.session_state.language_preference = lang

# PDF upload
uploaded_file = st.file_uploader("Upload Nepali PDF", type=["pdf"])
if uploaded_file:
    load_document_to_knowledge_base(uploaded_file)

# Chat interface
user_input = st.text_input("Ask about the PDF content:")
if st.button("Submit") and user_input:
    response = generate_response(user_input)
    st.session_state.conversation_history.append({
        "query": user_input,
        "response": response
    })

# Display conversation
for entry in st.session_state.conversation_history:
    st.markdown(f"**You:** {entry['query']}")
    st.markdown(f"**Assistant:** {entry['response']}")
    st.markdown("---")

# Reset functionality
if st.button("Reset Chat"):
    st.session_state.conversation_history = []
    st.session_state.vectorstore = None
    st.rerun()

# Instructions
st.markdown("""
<style>
.stAlert { padding: 20px; border-radius: 10px; }
.stMarkdown { font-size: 16px; }
</style>
""", unsafe_allow_html=True)
```

- **Title**: Displays the app name, here its Nepali Chatbot
- **Language Selection**: Allows users to choose English or Nepali responses.
- **File Upload**: Accepts PDF uploads and processes them into the knowledge base.
- **Chat Interface**: Takes user queries, generates responses, and displays the conversation history.
- **Reset Button**: Clears the chat and vector store, refreshing the app.
- **CSS Styling**: Enhances the UI with custom styles.

7. Usage

1. **Setup**: Ensure dependencies are installed and the Google API key is configured in .env.
2. **Run**: Execute the script with streamlit run script.py.
3. **Interact**:
    - Select a response language (English or Nepali).
    - Upload a Nepali PDF.
    - Ask questions about the PDF content.
    - View the conversation history and reset if needed.