Comparative Analysis of Closed Domain Question Answering System and RAG System Using Hugging Face for Research Paper Survey

-NAME : LUV ARORA SRN : PES2UG21CS257
-NAME : MANASVI VARMA. SRN : PES2UG21CS305

---

1. Introduction

The advent of sophisticated natural language processing (NLP) techniques has revolutionized the field of information retrieval, particularly in the development of question answering systems. These systems are designed to navigate vast repositories of text and return precise answers to user queries, thereby facilitating quick access to relevant information. In the realm of academic research, such systems are invaluable, as they can sift through extensive collections of research papers to extract specific data, findings, or conclusions based on the queries posed by users. This report delves into two distinct approaches to question answering in the research domain.

The first is the **Closed Domain Question Answering System**, a specialised system designed to operate within a specific knowledge domain, using the contained and structured information of that domain to provide accurate answers to related queries. This system leverages traditional NLP methodologies, including text extraction, tokenisation, stemming, and semantic analysis, to identify and retrieve the most relevant sections of text that respond to user inquiries.

The second approach is embodied by the **Retrieval-Augmented Generation (RAG) System**, which employs the Hugging Face API to access a broader, more dynamic range of information. Unlike the Closed Domain system, the RAG system is not limited to a predefined corpus. Instead, it utilises the vast, interconnected databases accessible through the Hugging Face platform, enabling it to generate responses that are not only informed by the specific texts but also by the larger context of accumulated knowledge across various domains.

The Closed Domain Question Answering System excels in providing concise, accurate answers to queries strictly relevant to its predefined knowledge base. It is an epitome of efficiency and precision within its scope, making it ideal for users who require straightforward answers to specific questions. On the other hand, the RAG System, with its broader information retrieval capabilities, offers a more flexible and comprehensive approach. It can handle a wider array of question types, including those that require inferential reasoning and contextual understanding beyond the confines of a single document or domain.

In comparing these systems, this report aims to shed light on the strengths and limitations of each approach. It will explore the technical foundations of the systems, the methodologies they employ, their performance in real-world scenarios, and the contexts in which they are most effectively applied. This analysis is geared towards providing a nuanced understanding of how these systems can be leveraged to enhance research and information retrieval in the digital age.

2. Project 1: Closed Domain Question Answering System

2.1 Overview

The Closed Domain Question Answering System specializes in providing precise answers within a specific knowledge area, leveraging a structured database or document set. This focus allows for highly accurate responses to queries in domains such as medicine, law, or technology. The system's design integrates advanced natural language processing (NLP) techniques to analyze and retrieve information from domain-specific texts.

2.2 Functionalities

The system’s capabilities are outlined as follows, detailing the mechanisms at work:

- **Text Extraction and Preprocessing:**
  - **Extraction:** Text is extracted from PDFs using PyPDF2 and Fitz, converting printed material into analyzable digital text.
  - **Preprocessing:** The raw text is then processed to remove irrelevant elements. This includes converting to lowercase, stripping punctuation, and filtering out non-alphanumeric characters to standardize the text for further analysis.
- **Tokenization and Stemming:**
  - **Tokenization:** The system segments the text into tokens (words or phrases), enabling detailed textual analysis.
  - **Stemming:** It employs the Snowball Stemmer to condense words to their base forms, reducing the complexity of the text and aiding in the identification of core concepts.
- **TF-IDF Vectorization:**
  - The **TF-IDF Vectorizer** transforms text into a matrix of TF-IDF features, quantifying the importance of words relative to the document and corpus. This representation is crucial for identifying significant terms that are most likely to be relevant to the user's query.
- **Cosine Similarity Analysis:**
  - Utilizing the vectors produced by TF-IDF, the system applies cosine similarity to measure the textual relevance between the query and documents. This metric assesses the cosine angle between two vectors, with a smaller angle indicating greater similarity, thus prioritizing text segments with higher relevance to the query.
- **Question Analysis and Answer Extraction:**
  - **Question Analysis:** The system uses POS tagging to dissect the question structure and understand its grammatical composition. This helps in determining the query's intent and required answer type.
  - **Answer Extraction:** Leveraging NER, the system identifies and extracts entities that match the expected answer type (like names, locations, or dates) from the text. This process is finely tuned to the query’s context, ensuring that the retrieved information directly addresses the user's question.

Through these intricate processes, the Closed Domain Question Answering System adeptly navigates the specific knowledge domain, efficiently retrieving accurate and relevant answers. This detailed functionality enables it to serve as a powerful tool for users seeking information within specialized areas, demonstrating its utility in facilitating targeted knowledge discovery.

2.3 Technology Stack

The technology stack for the Closed Domain Question Answering System includes Python for backend processing, Flask for web application deployment, NLTK and spaCy for natural language processing, PyPDF2 and Fitz for PDF text extraction, and the Scikit-learn library for implementing TF-IDF vectorization and cosine similarity analysis.

2.4 Results and Evaluation

The Closed Domain Question Answering System demonstrates robust performance in the precise retrieval of information within its designated domain. Through comprehensive testing and real-world application, the system has shown a high degree of accuracy and efficiency in responding to domain-specific queries.

Accuracy and Precision:

In scenarios where users queried for specific information, such as "What is the molecular structure of Penicillin?" or "What are the legal implications of copyright infringement in digital media?", the system successfully retrieved the exact passages containing the answers from the corpus. This precision stems from its effective use of NLP techniques and the focused nature of its dataset.

Speed and Efficiency:

The system's response time is notably swift, typically generating answers in a few seconds. This efficiency is attributed to the streamlined preprocessing and vectorization processes, which swiftly parse and index the text for rapid retrieval.

Relevance of Answers:

The answers provided by the system are highly relevant to the queries. For example, when asked about a specific concept in quantum mechanics, the system returned an excerpt from a research paper detailing the concept with appropriate contextual information.

Handling of Complex Queries:

The system adeptly handles complex queries that require an understanding of the context and relationships within the domain. For instance, in response to the question "How does the principle of superposition affect quantum computing?", it extracted and presented a concise explanation from a collection of scientific texts.

Limitations and Challenges:

While the system excels in its designated domain, it may face challenges with queries that span multiple domains or require inferential reasoning beyond the scope of the stored documents. Additionally, the system's reliance on static data sources means it may not have the most current information unless regularly updated.It cannot answer any query outside the knowledge of the pdf.

In summary, the Closed Domain Question Answering System exhibits strong capabilities in providing accurate and relevant answers within its specified domain. Its technical architecture, combining efficient data processing algorithms with sophisticated NLP techniques, forms a reliable foundation for domain-specific information retrieval- this case literature survey.

3. Project 2: RAG System Using Hugging Face

3.1 Overview

RAG System Using Hugging Face utilizes the Retrieval-Augmented Generation (RAG) system leveraging the Hugging Face API to access a broader knowledge base for answering questions. This approach integrates retrieval and generation to produce informed and context-aware responses across a wide array of domains, making it ideal for complex queries that require comprehensive understanding and creative problem-solving.

3.2 Functionalities

The functionalities of the RAG system are designed to handle a diverse set of use cases by dynamically sourcing information from large datasets and generating responses that are contextually relevant to the user's input. The detailed functionalities are as follows:

- **Document Retrieval and Processing:**
  - **WebBaseLoader:** The system begins by retrieving textual data from specified URLs using the WebBaseLoader. This component is crucial for accessing up-to-date content from the web, ensuring that the information processed is current and relevant.
  - **RecursiveCharacterTextSplitter:** Once the documents are loaded, the RecursiveCharacterTextSplitter breaks down the content into manageable chunks. This splitting is essential for processing large documents efficiently and effectively, facilitating quicker retrieval of information during the question-answering phase.
- **Vector Store Creation:**
  - **HuggingFaceEmbeddings:** The extracted text chunks are then converted into vector representations using the HuggingFaceEmbeddings, which utilize pre-trained models (like 'sentence-transformers/all-MiniLM-l6-v2') to generate embeddings. These embeddings are crucial for the semantic understanding of the text, allowing the system to perform nuanced searches across the content.
  - **Chroma:** The Chroma library is used to create a vector store from the document embeddings. This vector store acts as a searchable database, enabling the system to quickly locate information relevant to the queries by comparing the semantic similarity of embeddings.
- **Contextual Retrieval and Answer Generation:**
  - **Retriever:** The vector store powers a retriever that performs the initial filtering of relevant document chunks based on the user's query. This retriever is the backbone of the information retrieval phase, determining which pieces of text are likely to contain the answers.
  - **HuggingFaceHub and ChatHuggingFace:** Utilizing models from HuggingFaceHub, specifically the advanced capabilities of the 'zephyr-7b-beta' for text generation, the system generates coherent and contextually appropriate responses. The ChatHuggingFace model integrates these capabilities into a conversational AI, allowing for dynamic and interactive user experiences.
  - **Retrieval and Generation Chain:** The integration of a retrieval chain with a generation chain (via create_retrieval_chain and create_stuff_documents_chain) allows the system to not only find relevant information but also to refine and tailor the responses based on the context of the entire conversation, enhancing the relevance and accuracy of the answers.

- **Interactive Web Interface:**
  - **Streamlit:** The use of Streamlit for the web interface enables a user-friendly and interactive environment for users to input their questions and receive answers. The interface maintains a session state to track the conversation history, which is critical for context-aware responses in ongoing interactions.

3.3 Technology Stack

The technology stack for the RAG System Using Hugging Face encompasses Streamlit for the web interface, allowing dynamic interaction with users. The system utilizes the Langchain library for document loading and text splitting, and the Chroma library for creating vector stores from document chunks. For NLP processing and answer generation, the system integrates Hugging Face's transformers and APIs, which provide access to advanced language models and embedding techniques.

3.4 Results and Evaluation

The RAG System Using Hugging Face demonstrates impressive capabilities in answering a wide range of questions by leveraging its advanced retrieval and generation methods. The system's effectiveness is evaluated based on several key aspects:

Relevance and Contextual Accuracy:

- The system was tested with queries requiring in-depth answers, such as "Explain the theory behind quantum entanglement." The RAG System efficiently retrieved relevant scholarly contexts and generated a detailed and understandable explanation, showcasing its ability to synthesize complex information.
- But for literature surely -Domain specific queries, it answers out o context from the trained documents and starts hallucinating .On Average the first few lines tend be from the document but the rest are generated from the knowledge from the retrained dataset.

Speed and Responsiveness:

- Response times vary depending on the complexity of the query and the amount of data processed. For straightforward queries, the system provides answers within seconds, but more complex inquiries involving extensive data retrieval and processing may take longer due to the api calls.

Handling of Ambiguous Queries:

- The system's performance with ambiguous questions, such as "What are the implications of Brexit for global trade?" was notably robust. It managed to pull in diverse viewpoints and synthesize a comprehensive response that covered potential economic, legal, and political impacts, demonstrating its ability to handle ambiguity and complexity.

Adaptability and Learning:

- The RAG System's adaptability was highlighted in scenarios where the context evolved over the course of a conversation. For instance, during a discussion about renewable energy, as the conversation shifted from solar energy technologies to policy implications, the system seamlessly adjusted its responses to fit the new context, reflecting its dynamic conversational abilities.

Limitations and Areas for Improvement:

- While generally accurate, the system occasionally produces "hallucinated" information —responses that are plausible but not directly supported by the source material. This is an area for ongoing refinement, particularly in ensuring the trustworthiness and veracity of generated content.

- The system also requires a constant internet connection to access the Hugging Face API and retrieve data, which could be a limitation in environments with unstable internet access.

Overall, the RAG System Using Hugging Face represents a significant advancement in question answering technologies. Its ability to integrate retrieval with generative processes allows it to address a wide array of questions with high relevance and contextual accuracy, although improvements in response verification and speed could enhance its application further.

4. Comparative Analysis

The comparative analysis of the Closed Domain Question Answering System and the RAG System Using Hugging Face highlights distinct advantages and limitations inherent to each approach. The differences in their design and functionality cater to different needs and scenarios in the field of question answering.

4.1 Speed and Efficiency

- Method 1 is highly optimized for speed, efficiently processing queries within its closed domain. This rapid response capability is due to the system's reliance on predefined data and local processing, which eliminates the latency associated with online data retrieval.
- Method 2, on the other hand, experiences slower response times due to its dependence on external APIs and the internet for accessing broader knowledge bases. The complexity of integrating retrieval with generation also adds to the latency, making it less suited for applications where speed is a critical factor.

4.2 Answer Accuracy and Relevance

- Method 1 excels in delivering highly accurate and relevant answers to queries strictly within its designated domain. This system is particularly effective in environments like research or literature review, where precision and domain-specific accuracy are crucial.
- Method 2 provides a broader range of answers that extend beyond the specific texts it has access to, thanks to its use of pre-trained models. This ability to incorporate general knowledge makes it versatile but also introduces a risk of generating answers that, while plausible, may not always be rooted in the inputted documents.

4.3 Scope of Knowledge

- Method 1 is confined to the content of the documents it processes. Its capability is limited to extracting and interpreting information exclusively from the specified texts, making it unsuitable for queries requiring external context or updated information beyond the provided materials.
- Method 2 is not restricted to the input documents and can draw upon a vast array of pre-existing knowledge, making it capable of addressing questions that are not directly answered in the source texts. This makes it highly valuable for exploratory questions and topics where cross-disciplinary insights are beneficial.

4.4 Application Suitability

- Method 1 is ideal for closed-domain tasks where the users need specific answers from a restricted set of documents, such as detailed technical queries, legal case analysis, or medical research where the information is contained within certain bounds.
- Method 2 is better suited for open-ended inquiries and discussions that may require creative answers or insights drawn from a broader spectrum of sources. Its capabilities are particularly useful in educational settings, creative industries, or scenarios where interactive engagement and wide-ranging knowledge are desired.

5. Conclusion

The choice between the two systems should be guided by the specific needs of the application. For tasks requiring high speed and precise answers within a well-defined domain, Method 1 is the superior choice. Conversely, for applications that benefit from a wider scope of knowledge and can tolerate slower response times,Method 2 offers substantial advantages. Each system's strengths and limitations must be considered in light of the intended use case to maximise their effectiveness and utility in real-world applications.

6.Code Snippets

6.1 Domain Specific Question/Answering

```
import nltk
nltk.download('all')
import re
import string
import gensim
import os
import PyPDF2
import fitz
from gensim.parsing.preprocessing import remove_stopwords
from nltk import sent_tokenize, word_tokenize
from nltk.stem.snowball import SnowballStemmer
from sklearn.metrics.pairwise import cosine_similarity
from scipy import spatial
from nltk import pos_tag,word_tokenize,ne_chunk
from sklearn.feature_extraction.text import TfidfVectorizer
from pandas import DataFrame

from nltk.corpus import wordnet,stopwords
import spacy
import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

from flask import Flask, render_template, request, jsonify, send_from_directory
```

```
UPLOAD_FOLDER = 'uploads'
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

# Ensure the upload folder exists
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
```

```
def stem_sentence(sentence):
  words=word_tokenize(sentence)
  #lemmatizer = WordNetLemmatizer()


  stemmer = SnowballStemmer("english")
  new_words=[]
  for i in words:
    new_words.append(stemmer.stem(i))
    new_words.append(" ")
  return "".join(new_words)



def clean_sentence(sentence, stopwords=True):

    sentence = sentence.lower().strip()
    sentence = re.sub(r'[^a-z0-9\s]', '', sentence)



    if stopwords:
         sentence = remove_stopwords(sentence)



    return sentence


def get_cleaned_sentences(sents,stopwords=True):

    cleaned_sentences=[]

    for i in sents:

        cleaning=clean_sentence(i,stopwords)
        cleaned=stem_sentence(cleaning)
        cleaned_sentences.append(cleaned)
    return cleaned_sentences
```

```
def create_document_term_matrix(sen,vectorizer):
  doc_term_matrix=vectorizer.fit_transform(sen)
  return DataFrame(doc_term_matrix.toarray(), columns=vectorizer.get_feature_names_out())



def calculate_cosine_similarity(df_list,sentences,question):
  a=[]
  for i in range(len(df_list)-1):
    sim=1 - spatial.distance.cosine(df_list[i], question)
    t=(sim,sentences[i])
    a.append(t)
  a.sort(reverse=True)
  n=[]
  for i in range(3):
    n.append(a[i][1])
    #print("*",n[i])


  return n


def questiontype( question):
        questiontags = ['WP','WDT','WP$','WRB']
        question_POS = pos_tag(word_tokenize(question.lower()))

        question_Tags=[]
        for token in question_POS:
            if token[1] in questiontags:
              question_Tags.append(token)


        if len(question_Tags)==1 and question_Tags[0][0]!= 'what' :
          return True
        else:
          return False
```

```
from nltk.util import ngrams
def n_gram_similarity(question,n):
  q=list(ngrams(word_tokenize(question.lower()),1))
  a=0
  b=0
  c=0
  t=[]
  for i in q:
    if i in list(ngrams(word_tokenize(n[0].lower()),1)):
      a=a+1
  for i in q:
    if i in list(ngrams(word_tokenize(n[1].lower()),1)):
      b=b+1
  for i in q:
    if i in list(ngrams(word_tokenize(n[2].lower()),1)):
      c=c+1
  d=max(a,b,c)
  if a == d:
    t.append(n[0])
  if b == d:
    t.append(n[1])
  if c ==d:
    t.append(n[2])
  print()
  #print("Selected Sentence:",t[0])
  return t
```

```
    if len(entity_type) == 0:
      #print("ANSWER TYPE:", t)
       print(n[0])
       return  n[0]
    if len(entity_type) > 1:
      #print("Answer Type:",t)
      key_question = question
      q=[]
      spdoc = nlp(key_question)
      for ent in spdoc:
        if ent.pos_ == 'NOUN' or ent.pos_ =='ADJ' :
          q.append(ent.text)

      key_answer = n[0]
      a = []
      spd = nlp(key)
      for ent in spd:
        if ent.pos_ == 'NOUN'or ent.pos_ =='ADJ' :
          a.append(ent.text)
  #s=[sentence.index(i) for i in t]
      s=[]
      w=[]
      for i in entity_type:
       s.append(n[0].index(i))
      for i in range(len(s)):
        w.append(0)


      for i in q:
        try:
           factor= n[0].index(i)
           for j in range(len(s)):
              w[j]=w[j]+(abs(s[j]-factor))
        except:
           continue
      m=min(w)
      u=[]
      for i in range(len(s)):
        if w[i] == m:
```

```
def answertype(question,df_list,sentences):
  nlp = spacy.load('en_core_web_sm')

  if (questiontype(question)):
    t='DESCRIPTIVE'
    flag=0
    word=word_tokenize(question.lower())

    if 'who' in word:
      t='PERSON'
    elif 'where' in word:
      t='GPE'
    elif 'how' in word and 'many' in word and  'age' in word or 'duration' in word or 'long' in word or 'days'in word or 'years' in word or'months' in word:
      t='DATE'
    elif 'how' in word and 'many' in word :
       t = 'CARDINAL'
    elif 'when' in word  or 'age' in word or 'period' in word or 'duration' in word  or 'old' in word or 'long' in word:
      t='DATE'
    elif 'how' in word  and 'long' in word or 'often' or 'age' in word or 'years' in word:
      t='DATE'
    elif 'what' in word and 'time' in word or 'duration' in word or 'period' in  'word'  :
      t='DATE'
    i=len(df_list)-1
    n=calculate_cosine_similarity(df_list, sentences,df_list[i])
    n=n_gram_similarity(question,n)
    #print("Most relevant sentence", n[0])
    #print("ANSWER TYPE:",t)
    key = n[0]
    spdoc = nlp(key)
    entity_type=[]
    for ent in spdoc.ents:
       if ent.label_ == t:
          entity_type.append(ent.text)
    if len(entity_type) == 1:
      #print("ANSWER TYPE:", t)
       print("ANSWER:", entity_type[0])
       return entity_type[0]
    if len(entity_type) == 0:
      #print("ANSWER TYPE:", t)
       print(n[0])
       return  n[0]
    if len(entity_type) > 1:
      #print("Answer Type:",t)
```

```
        if w[i] == m:
           #print(entity_type[i])
           u.append(entity_type[i])
      print("ANSWER:",u[0])
      return u[0]


  else:
    t='DESCRIPTIVE'
    l=[]
    print("ANSWER TYPE:",t)
    i=len(df_list)-1
    n=calculate_cosine_similarity(df_list, sentences,df_list[i])
    #n = n_gram_similarity(question, n)
    for j in n:
         l.append(j)
         print(j)

    l="".join(l)
    return l


@app.route("/")
def index():
    return render_template('base.html')


@app.route("/get", methods=["GET", "POST"])
def chat():

    doc = fitz.open(filepath_copy)
    text = ""

    # Iterate through each page
    for page in doc:
        text += page.get_text()

    doc.close()
    print("-------------------------F-----------------------------")
```

```
    question=[input]

    #question=["What happens upon photolysis?"]
    for j in question:
    #j=TextBlob(j)
    #j=str(j.correct())

        qq=[]
        qp=[]
        que=sent_tokenize(j)

        qq.append(que)
        qp.append(j)
        questiontags = ['WP','WDT','WP$','WRB']
        question_POS = pos_tag(word_tokenize(j.lower()))

        question_Tags=[]
    for token in question_POS:
        if token[1] in questiontags:
            question_Tags.append(token)


    if len(question_Tags)>1:
        if ' and ' in j :
            pos=j.lower().find('and')
            qq=[]
            qp=[]
            qp.append(j[:pos])
            qp.append(j[pos+1:])
            qq.append(sent_tokenize(j[:pos]))
            qq.append(sent_tokenize(j[pos+1:]))




    for k in range(len(qp)):
```

```
for k in range(len(qp)):



    sentences=sent_tokenize(text)
    #q contains a list of cleaned sentence tokens of question
    q=get_cleaned_sentences(qq[k],stopwords=True)
    #preprocessed contains a list of cleaned sentence tokens of the reference text
    preprocessed=get_cleaned_sentences(sentences,stopwords=True)

    preprocessed.append(q[0])
    i=len(preprocessed)-1


    tfidf_vect=TfidfVectorizer()
    df=create_document_term_matrix(preprocessed,tfidf_vect)
    df_list = df.values.tolist()

    x=answertype(qp[k],df_list,sentences)
    print("ooooooooooooo----------------------------------ooooooooooooooo")
    print(x)

return x
```

6.2 Rag Model for Question Answering

```
import streamlit as st
import os
from langchain_core.messages import HumanMessage, SystemMessage,AIMessage
from langchain_community.document_loaders import WebBaseLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma


from langchain_community.embeddings import HuggingFaceEmbeddings
from transformers import AutoTokenizer, AutoModelForQuestionAnswering
from transformers import AutoTokenizer, pipeline
from langchain_community.llms.huggingface_pipeline import HuggingFacePipeline
from langchain.chains import RetrievalQA, create_history_aware_retriever,create_retrieval_chain
from langchain_community.chat_models.huggingface import ChatHuggingFace
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_community.llms import HuggingFaceHub

from dotenv import load_dotenv

load_dotenv()
```

```
def get_vectorstore_from_url(url):
    #getting text in documnet form
    loader= WebBaseLoader(url)
    document=loader.load()

    #split document in to chunks
    text_splitter = RecursiveCharacterTextSplitter()
    document_chunks = text_splitter.split_documents(document)

    #create vector stores from the chunks
    # Define the path to the pre-trained model you want to use
    modelPath = "sentence-transformers/all-MiniLM-l6-v2"
    model_kwargs = {'device':'cpu'}
    encode_kwargs = {'normalize_embeddings': False}
    embeddings = HuggingFaceEmbeddings(
        model_name=modelPath,
        model_kwargs=model_kwargs,
        encode_kwargs=encode_kwargs
    )

    vector_store = Chroma.from_documents(document_chunks,embeddings)

    return vector_store
```

```
def get_context_retriever_chain(vector_store):


    llm = HuggingFaceHub(
        repo_id="HuggingFaceH4/zephyr-7b-beta",
        task="text-generation",
        model_kwargs={
            "max_new_tokens": 512,
            "top_k": 5,
            "temperature": 0.1,
            "repetition_penalty": 1.03,
        },
    )

    chat_model = ChatHuggingFace(llm=llm)

    retriever= vector_store.as_retriever()

    prompt=ChatPromptTemplate.from_messages([
        MessagesPlaceholder(variable_name="chat_history"),
        ("user","{input}"),
        ("user","Given the above query ,generate a search query to look up in order to get information relevant to the
conversation ")
    ])

    retriever_chain = create_history_aware_retriever(chat_model,retriever,prompt)


    return retriever_chain
```

```python
def get_coversational_rag_chain(retriever_chain):


    llm = HuggingFaceHub(
        repo_id="HuggingFaceH4/zephyr-7b-beta",
        task="text-generation",
        model_kwargs={
            "max_new_tokens": 512,
            "top_k": 5,
            "temperature": 0.1,
            "repetition_penalty": 1.03,
        },
    )

    chat_model = ChatHuggingFace(llm=llm)

    prompt = ChatPromptTemplate.from_messages([
        MessagesPlaceholder(variable_name="chat_history"),
        ("system","Answer the user questions based on the above context :{context}\n\n"),
        ("user","{input}"),

    ])

    stuff_documents_chain = create_stuff_documents_chain(chat_model,prompt)

    return create_retrieval_chain(retriever_chain,stuff_documents_chain)

def get_response(user_input):
    response = conversation_rag_chain.invoke({

            "chat_history": st.session_state.chat_history,
            "input" : user_input


        })


    return response["answer"]


#app config
st.set_page_config(page_title="Chat with Websites",page_icon="🐸")
st.title("Chat with websites")



#app sidebar
with st.sidebar:
    st.header("Settings")
    website_url = st.text_input("website url")




if website_url is None or website_url =="":
    st.info("Please enter a website")

else:

    #session state
    if "chat_history" not in st.session_state:
        st.session_state.chat_history = [
        AIMessage(content="Hello ,I am a bot ,How can i help you")
        ]

    if "vector_store" not in st.session_state:
        st.session_state.vector_store = get_vectorstore_from_url(website_url)

    #create conversation chain
    retriever_chain = get_context_retriever_chain(st.session_state.vector_store)
    conversation_rag_chain = get_coversational_rag_chain(retriever_chain)


    #user input
    user_query = st.chat_input("Type your message...")
    if user_query is not None and user_query != "":
        response = get_response(user_query)
        st.session_state.chat_history.append(HumanMessage(content=user_query))
        st.session_state.chat_history.append(AIMessage(content=response))




    #conversation
    for message in st.session_state.chat_history:
        if isinstance(message,AIMessage):
            with st.chat_message("AI"):
                st.write(message.content)
        elif isinstance(message,HumanMessage):
            with st.chat_message("Human"):
                st.write(message.content)
```