Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías Departamento de Ciencias Computacionales

Asignatura**: ANÁLISIS DE ALGORITMOS**
Clave de Asignatura: **IL355**
NRC: **204843**
Sección: **D06**

Código fuente backend

Alumnos y roles en el equipo:

- Frontend y documentación: Ruíz González Mariana - 221978248
- Backend y documentación: Chávez Velasco Cristian - 218532484
- Backend y documentación: Valadez Gutierrez David - 217450107

Profesor: **LOPEZ ARCE DELGADO, JORGE ERNESTO**

Fecha: 15 de mayo de 2024

Desarrollo de la lógica de negocio

Manejo de Consultas:
- El usuario envía una consulta a través de la interfaz de usuario.
- La consulta se procesa mediante las técnicas de NLP (análisis gramatical, reconocimiento de entidades) para entender su significado.

Generación de Respuestas:
- El modelo de generación de respuestas utiliza la información extraída de la base de datos de libros para generar una respuesta adecuada.
- La respuesta se verifica para asegurar que sea coherente y relevante al contexto de la consulta.

Entrega de Respuestas:
- La respuesta generada se envía de vuelta al usuario a través de la interfaz de usuario.
- La interfaz de usuario presenta la respuesta de manera clara y comprensible.

Optimización y Eficiencia:
- Se realizan pruebas y ajustes continuos para mejorar la precisión y relevancia de las respuestas del chatbot.

Algoritmos

Large Language Models (LLM): Modelos de lenguaje estadísticos o basados en aprendizaje automático que están diseñados para manejar grandes cantidades de datos de texto y generar contenido coherente y relevante.

LangChain: Framework que nos permite interactuar con LLMs de manera fácil y rápida. El cual cuenta con dos funcionalidades principales: Integración de datos (llamado data-aware): conecta un LLM con otra fuente de datos, por ejemplo, todo el texto que tiene un PDF
Es posible que queramos que nuestra aplicación tenga un contexto determinado, o responda de una manera específica y solo cuando tenga respuestas específicas.

ConversationChain: Es una cadena más versátil diseñada para gestionar conversaciones. Genera respuestas basadas en el contexto de la conversación y no necesariamente depende de la recuperación de documentos.

ConversationalRetrievalChain: Está diseñado específicamente para responder preguntas basadas en documentos.
Útil cuando tiene documentos específicos que desea utilizar para generar respuestas.

Embeddings: Técnica de procesamiento de lenguaje natural que convierte el lenguaje humano en vectores matemáticos. Estos vectores son una representación del significado subyacente de las palabras, lo que permite que las computadoras procesan el lenguaje de manera más efectiva.

Permiten que las palabras sean tratadas como datos y manipuladas matemáticamente.

ChromaDB: Base de datos especializada en el almacenamiento y recuperación eficiente de información lingüística, incluyendo datos de texto, anotaciones semánticas y sintácticas.
Es particularmente útil para el almacenamiento y la gestión de grandes cantidades de datos de lenguaje natural, lo que permite a los desarrolladores aprovechar al máximo los avances en algoritmos de aprendizaje automático y análisis de texto.

RetrievalQA: Herramienta que combina técnicas de recuperación de información con procesamiento del lenguaje natural para responder preguntas formuladas en lenguaje natural sobre textos largos o documentos extensos. La idea detrás de este enfoque es que, en lugar de analizar todo el documento cada vez que se hace una pregunta, la herramienta primero busca en la base de datos o corpus relevante para encontrar los fragmentos más prometedores que podrían contener la respuesta.

Implementación de las funciones de procesamiento de datos

Carga y procesamiento de datos: Se utiliza GutenbergLoader para cargar el texto del libro desde Project Gutenberg.

```
# Obtener la informacion
loader = GutenbergLoader(
    "https://www.gutenberg.org/cache/epub/1268/pg1268.txt"
)

# Cargar informacion en memoria
document = loader.load()
```

División del texto: El texto cargado se divide en fragmentos manejables usando CharacterTextSplitter.

```
# Dividir o cortar el texto
# Chunk sizes of 1024 and an overlap of 256 (this will take approx.
10mins with this model to build our vector database index)
text_splitter = CharacterTextSplitter(
    chunk_size=1024,
    chunk_overlap=256
)
texts = text_splitter.split_documents(document)
```

Generación de embeddings: Se utilizan embeddings preentrenados de HuggingFaceEmbeddings para representar los textos.

```
model_name = "sentence-transformers/all-MiniLM-L6-v2"

# Guardar la informacion codificada en embeddings
embeddings = HuggingFaceEmbeddings(
    model_name=model_name,
```

```
    cache_folder=cache_dir # Modelo preentrenado
)  # Use a pre-cached model
```

Almacenamiento en base de datos vectorial: Los embeddings se almacenan en una base de datos vectorial usando Chroma.

```
#Guarda la informacion en chroma
vectordb = Chroma.from_documents(
    texts,
    embeddings,
    persist_directory=cache_dir
)
```

Configuración del recuperador: Se configura un recuperador de documentos similar a partir de la base de datos vectorial.

```
# We want to make this a retriever, so we need to convert our index.
# This will create a wrapper around the functionality of our vector
database
# so we can search for similar documents/chunks in the vectorstore and
retrieve the results:
retriever = vectordb.as_retriever()
```

Manejo de las solicitudes y respuestas del cliente

Configuración del modelo LLM: Se utiliza un modelo de lenguaje grande (google/flan-t5-large) de HuggingFace para interpretar el lenguaje.

```

# This chain will be used to do QA on the document. We will need
# 1 - A LLM to do the language interpretation
# 2 - A vector database that can perform document retrieval
# 3 - Specification on how to deal with this data

hf_llm = HuggingFacePipeline.from_model_id(
    model_id="google/flan-t5-large",
    task="text2text-generation",
    model_kwargs={
#         "temperature": 0,
        "do_sample":True,
        "max_length": 2048,
        "cache_dir": cache_dir,
    },
)
```

Creación de la cadena de preguntas y respuestas: Se configura una cadena de preguntas y respuestas (RetrievalQA) para manejar las consultas.

```
qa = RetrievalQA.from_chain_type(
    llm=hf_llm,
    chain_type="refine",
    retriever=retriever
)
```

Ejecución de una consulta: Se ejecuta una consulta ejemplo y se imprime el resultado.

```
query = "Who is the main character?"
query_results_venice = qa.run(query)
print("#" * 12)
query_results_venice
```

Pruebas unitarias y de integración

Pruebas Unitarias: Deben enfocarse en probar funciones individuales, como la carga de documentos, la generación de embeddings y la división del texto.

```
def test_load_document():
                                                    loader              =
GutenbergLoader("https://www.gutenberg.org/cache/epub/1268/pg1268.txt")
    document = loader.load()
    assert document is not None

def test_split_text():
             text_splitter    =    CharacterTextSplitter(chunk_size=1024,
chunk_overlap=256)
    texts = text_splitter.split_documents(["This is a sample text."])
    assert len(texts) > 0
```

Pruebas de Integración: Deben asegurar que todos los componentes del sistema (carga de documentos, generación de embeddings, recuperación de documentos y generación de respuestas) funcionen juntos de manera cohesiva.

```
qa = RetrievalQA.from_chain_type(
    llm=hf_llm,
    chain_type="refine",
    retriever=retriever
)
```

```
query = "Who is the main character?"
query_results_venice = qa.run(query)
print("#" * 12)
query_results_venice
```

```
while True:
    # Crear una instancia de RetrievalQA
    qa = RetrievalQA.from_chain_type(
        llm=hf_llm,
        chain_type="refine",
        retriever=retriever
    )

    # Pedir al usuario la consulta
       query = input("Ingrese su pregunta (o escriba 'salir' para
terminar): ")

    # Verificar si el usuario quiere salir del programa
    if query.lower() == "salir":
        print("Saliendo del programa...")
        break

    # Ejecutar la consulta
    query_results_venice = qa.run(query)

    # Imprimir los resultados
    print("#" * 12)
    print(query_results_venice)
```

Código fuente

```
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain, ConversationalRetrievalChain,
ConversationChain
from langchain.memory import ConversationBufferMemory
from langchain.schema import messages_from_dict, messages_to_dict
from langchain.memory.chat_message_histories.in_memory import
ChatMessageHistory
from langchain.agents import Tool
from langchain.agents import initialize_agent
from langchain.agents import AgentType
import pandas as pd
from langchain.document_loaders import GutenbergLoader
```

```
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import Chroma
import tempfile
from langchain.llms import HuggingFacePipeline
from langchain.chains import RetrievalQA

cache_dir = "/content/"

pd.set_option('display.max_column', None)
pd.set_option('display.max_rows', None)
pd.set_option('display.max_seq_items', None)
pd.set_option('display.max_colwidth', 500)
pd.set_option('expand_frame_repr', True)

# Obtener la informacion
loader = GutenbergLoader(
    "https://www.gutenberg.org/cache/epub/1268/pg1268.txt"
)

# Cargar informacion en memoria
document = loader.load()

extrait = ' '.join(document[0].page_content.split()[:100])
display(extrait + " .......")



# Dividir o cortar el texto
# Chunk sizes of 1024 and an overlap of 256 (this will take approx.
10mins with this model to build our vector database index)
text_splitter = CharacterTextSplitter(
    chunk_size=1024,
    chunk_overlap=256
)
texts = text_splitter.split_documents(document)

model_name = "sentence-transformers/all-MiniLM-L6-v2"

# Guardar la informacion codificada en embeddings
embeddings = HuggingFaceEmbeddings(
    model_name=model_name,
    cache_folder=cache_dir # Modelo preentrenado
)  # Use a pre-cached model
```

```
#Guarda la informacion en chroma
vectordb = Chroma.from_documents(
    texts,
    embeddings,
    persist_directory=cache_dir
)



# We want to make this a retriever, so we need to convert our index.
#  This will create a wrapper around the functionality of our vector
database
# so we can search for similar documents/chunks in the vectorstore and
retrieve the results:
retriever = vectordb.as_retriever()

# This chain will be used to do QA on the document. We will need
# 1 - A LLM to do the language interpretation
# 2 - A vector database that can perform document retrieval
# 3 - Specification on how to deal with this data

hf_llm = HuggingFacePipeline.from_model_id(
    model_id="google/flan-t5-large",
    task="text2text-generation",
    model_kwargs={
#         "temperature": 0,
        "do_sample":True,
        "max_length": 2048,
        "cache_dir": cache_dir,
    },
)



qa = RetrievalQA.from_chain_type(
    llm=hf_llm,
    chain_type="refine",
    retriever=retriever
)
query = "Who is the main character?"
query_results_venice = qa.run(query)
print("#" * 12)
query_results_venice
```