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**

Proyecto Final - Avance 1

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: 03 de mayo de 2024

Desarrollo de la interfaz de usuario:

Para la interfaz de usuario utilizamos ipywidgets, también conocidos como jupyter-widgets, los cuales sirvieron para incorporar la interfaz dentro de la misma plataforma donde hicimos el backend, la cual es colab.

Definiciones e importaciones de librería a código:

```
from langchain.chains import RetrievalQA
from ipywidgets import widgets, Layout, VBox, HBox
from IPython.display import display
import time
import threading
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
```

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 procesen 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 la lógica de presentación e Integración de componentes visuales:

La interfaz trabaja con el formato que anteriormente se definió para la realización de preguntas al chatbot, el cual es mostrado en la barra donde se ingresan estas.

Lo que pasa en la función ejecutar consulta es lo siguiente:

- Para ejecutar las consultas primero se lee la pregunta ingresada como texto.
- Se pausa el widget del botón de consultas que se representa con el texto “Preguntar” para que no se pueda hacer otra consulta mientras el chatbot piensa en la respuesta.
- Se inicia un contador de tiempo para que se muestre cuanto se tarda en recibir una respuesta a partir de que se escribe la pregunta en el chat.
- Una vez se termina la consulta y se recibe la respuesta del chatbot, se reactiva el boton y la barra de consultas.

```
# Función para manejar la consulta del usuario y ejecutarla
def ejecutar_consulta(b):
    query = input_text.value
```

```
    if query.lower() == "!salir" or query.lower() == "!SALIR" or query.lower()
== "!Salir":
        output_text.value += "\nANSWERBOOK Dice ADIOS..."
    else:
        input_text.disabled = True  # Desactivar el widget de entrada mientras
se procesa la consulta
        output_text.value += "\n\nANSWERBOOK: Generando respuesta más
adecuada..."
        start_time = time.time() # Iniciar contador de tiempo
        qa = RetrievalQA.from_chain_type(llm=hf_llm, chain_type="refine",
retriever=retriever)
        query_results = qa.run(query)
        end_time = time.time() # Finalizar contador de tiempo
        tiempo_respuesta = round(end_time - start_time, 2) # Calcular tiempo
de respuesta
        output_text.value += "\n\nYOU: " + f"{query} \n" + "\nANSWERBOOK: " +
str(query_results) + f"\n\nANSWERBOOK: ¿Quieres saber algo más? Realiza otra
pregunta...\n"
        input_text.disabled = False  # Reactivar el widget de entrada después
de recibir la respuesta
        # Limpiar el widget de pregunta después de mostrar la respuesta
        input_text.value = ""
```

La función actualizar_contador se encarga de, como dice su nombre, actualizar el contador de tiempo de respuesta del chatbot, tomando en cuenta el tiempo transcurrido desde que se escribe la pregunta y el tiempo de la pregunta anterior:

```
# Función para actualizar el contador en tiempo real
def actualizar_contador():
    while True:
        if not input_text.disabled:
            tiempo_transcurrido = time.time() - tiempo_inicio
            tiempo_texto.value = f'Tiempo transcurrido:
{round(tiempo_transcurrido, 2)} segundos'
        time.sleep(0.1)  # Actualizar cada 0.1 segundos
```

Al momento en que se da click en el botón de preguntar que se crea en esta parte, se llama a la función ejecutar_consulta y esta realiza el proceso de llamar al modelo del chatbot de libros.

```
# Crear widgets de entrada y salida
input_text = widgets.Text(placeholder='Pregunta algo a AnswerBook...',
layout=Layout(width='98%'))
output_text = widgets.Textarea(layout=Layout(width='98%', height='200px'),
disabled=True)
tiempo_texto = widgets.Text(value='Tiempo transcurrido: 0 segundos',
layout=Layout(width='98%'))


# Crear botón para enviar la pregunta
```

```
submit_button = widgets.Button(description='Preguntar',
button_style='success', layout=Layout(width='20%'))
submit_button.on_click(ejecutar_consulta)
```

Formato de letra para la interfaz:

```
# Estilo de salida para que se vea más como un chat
output_text.style.description_width = 'initial'
output_text.style.font_weight = 'bold'
output_text.style.font_family = 'Arial, sans-serif'
output_text.style.color = 'black'
```

Acomodo de los widgets dentro de la interfaz:

```
# Colocar widgets en una caja vertical
chat_box = VBox([output_text, HBox([input_text, submit_button],
layout=Layout(width='98%')), tiempo_texto])
```

Aquí se establece el tamaño y el titulo de la ventana de la interfaz:

```
# Crear la ventana con título
accordion = widgets.Accordion(children=[chat_box], layout=Layout(width='35%'))
accordion.set_title(0, 'AnswerBook Chat')  # Establecer el título de la
ventana

# Mostrar la ventana con título
display(accordion)

# Iniciar el contador de tiempo
tiempo_inicio = time.time()

# Iniciar el hilo para actualizar el contador en tiempo real
thread_contador = threading.Thread(target=actualizar_contador)
thread_contador.start()
```

Pruebas de usabilidad y experiencia del usuario:

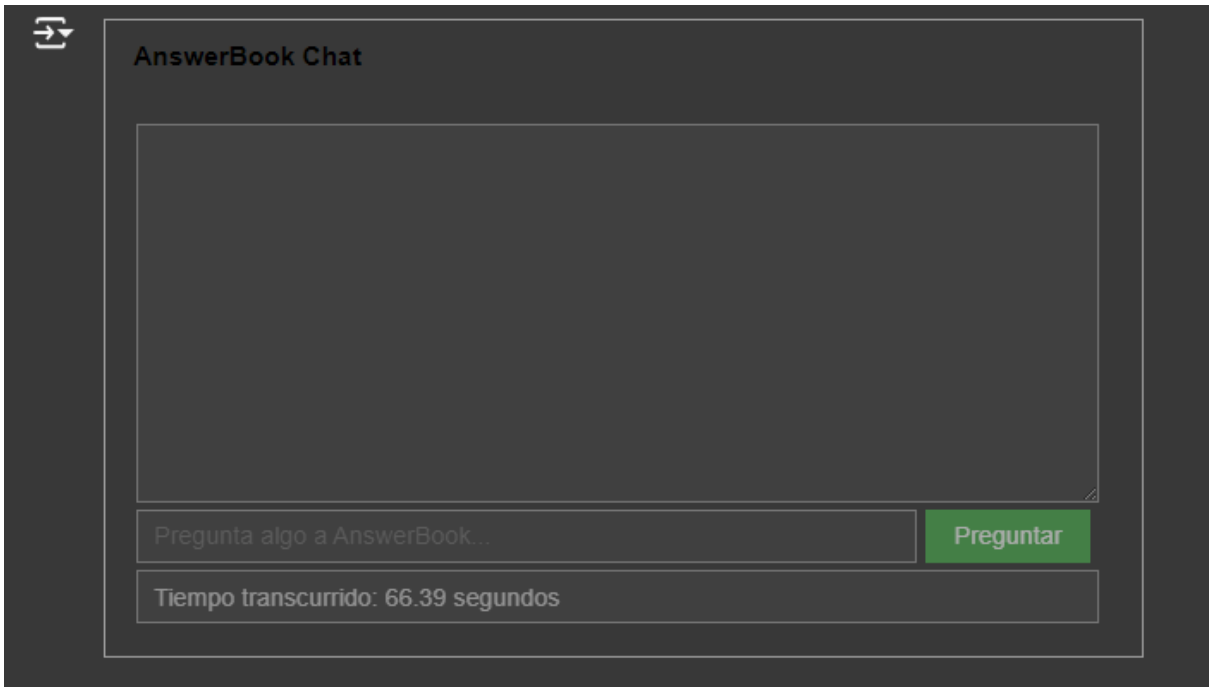

Figure 1 Interfaz en colab

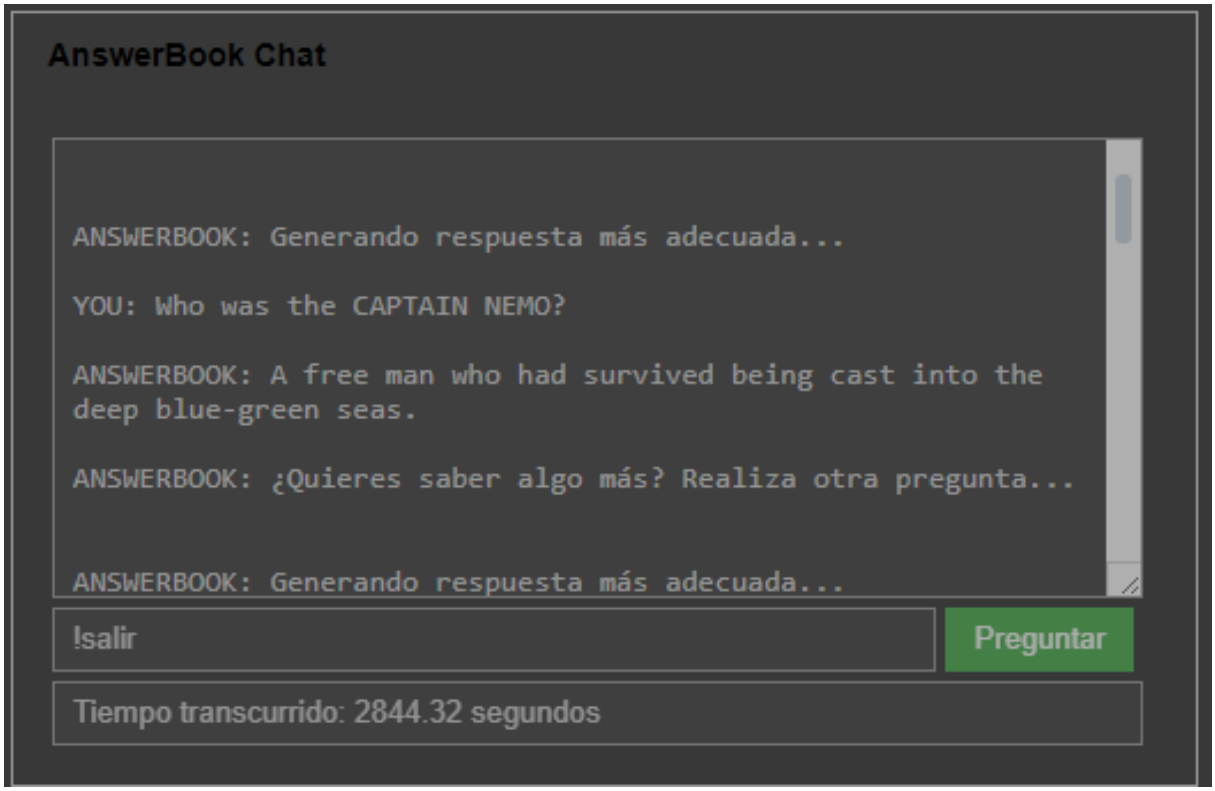

Figure 2 Pruebas realizadas dentro de la interfaz con ejemplo de las impresiones por el chatbot

Código completo de frontend con implementaciones relevantes:

```
# Todos
from langchain.chains import RetrievalQA
from ipywidgets import widgets, Layout, VBox, HBox
from IPython.display import display
import time
import threading
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,
    },
)

# Backend David
'''
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'''

# Implementacion Cristian

'''
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)'''

# Función para manejar la consulta del usuario y ejecutarla
def ejecutar_consulta(b):
    query = input_text.value
    if query.lower() == "!salir" or query.lower() == "!SALIR" or query.lower()
== "!Salir":
        output_text.value += "\nANSWERBOOK Dice ADIOS..."
    else:
        input_text.disabled = True  # Desactivar el widget de entrada mientras
se procesa la consulta
        output_text.value += "\n\nANSWERBOOK: Generando respuesta más
adecuada..."
        start_time = time.time() # Iniciar contador de tiempo
```

```
        qa = RetrievalQA.from_chain_type(llm=hf_llm, chain_type="refine",
retriever=retriever)
        query_results = qa.run(query)
        end_time = time.time() # Finalizar contador de tiempo
        tiempo_respuesta = round(end_time - start_time, 2) # Calcular tiempo
de respuesta
        output_text.value += "\n\nYOU: " + f"{query} \n" + "\nANSWERBOOK: " +
str(query_results) + f"\n\nANSWERBOOK: ¿Quieres saber algo más? Realiza otra
pregunta...\n"
        input_text.disabled = False  # Reactivar el widget de entrada después
de recibir la respuesta
        # Limpiar el widget de pregunta después de mostrar la respuesta
        input_text.value = ""

# Función para actualizar el contador en tiempo real
def actualizar_contador():
    while True:
        if not input_text.disabled:
            tiempo_transcurrido = time.time() - tiempo_inicio
            tiempo_texto.value = f'Tiempo transcurrido:
{round(tiempo_transcurrido, 2)} segundos'
        time.sleep(0.1)  # Actualizar cada 0.1 segundos

# Crear widgets de entrada y salida
input_text = widgets.Text(placeholder='Pregunta algo a AnswerBook...',
layout=Layout(width='98%'))
output_text = widgets.Textarea(layout=Layout(width='98%', height='200px'),
disabled=True)
tiempo_texto = widgets.Text(value='Tiempo transcurrido: 0 segundos',
layout=Layout(width='98%'))

# Crear botón para enviar la pregunta
submit_button = widgets.Button(description='Preguntar',
button_style='success', layout=Layout(width='20%'))
submit_button.on_click(ejecutar_consulta)

# Estilo de salida para que se vea más como un chat
output_text.style.description_width = 'initial'
output_text.style.font_weight = 'bold'
output_text.style.font_family = 'Arial, sans-serif'
output_text.style.color = 'black'

# Colocar widgets en una caja vertical
chat_box = VBox([output_text, HBox([input_text, submit_button],
layout=Layout(width='98%')), tiempo_texto])

# Crear la ventana con título
accordion = widgets.Accordion(children=[chat_box], layout=Layout(width='35%'))
```

```
accordion.set_title(0, 'AnswerBook Chat')  # Establecer el título de la
ventana

# Mostrar la ventana con título
display(accordion)

# Iniciar el contador de tiempo
tiempo_inicio = time.time()

# Iniciar el hilo para actualizar el contador en tiempo real
thread_contador = threading.Thread(target=actualizar_contador)
thread_contador.start()
```