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**

Reporte técnico

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: 16 de mayo de 2024

Desarrollo del proyecto

El equipo adoptó una metodología de trabajo modular para el desarrollo del proyecto. Se dividió en tres áreas principales: BackEnd, FrontEnd y Documentación, cada una con responsabilidades específicas.

Responsable	Objetivo	Resultado	Entregable	Fecha	Status	
Cristian	Objetivo del proyecto, Delimitaciones y herramientas	Envio de correo electronico	Correo electronico	30/04/2024	Completo	**Objetivos individuales**
	Objetivos individuales	Tareas para cada integrante	Objetivos en cronograma	02/05/2024	Completo	comprobar que el chatboot responde de manera coherente
	Implementar interaccion con usuario	Commit de modelo en repositorio	Modelo funcionando con usuario	07/05/2024	Completo	comprender cómo funciona backend de manera harcodeado
	Probar funcionamiento de la ventana junto con bakcend	Correo electronico de errores	Reporte en correo sobre errores	09/05/2024	Completo	implementar la interacción del usuario y chat desde consola
	Proyecto terminado, Presentacion de proyecto terminada	Entrega final, Presentacion de proyecto	Proyecto final, Presentacion del proyecto	14/05/2024	Completo	ayudar al desarrollo del frontend

Responsable	Objetivo	Resultado	Entregable	Fecha	Status	
Mariana	Objetivo del proyecto, Delimitaciones y herramientas	Envio de correo electronico	Correo electronico	30/04/2024	Completo	**Objetivos individuales**
	Objetivos individuales	Tareas para cada integrante	Objetivos en cronograma	02/05/2024	Completo	comprender funcionamiento de backend
	Implementacion una ventana con tkinter	Commit de ventana en repositorio	Codigo ventana con tkinter	07/05/2024	Completo	Desarrollar ventana de tkinter
	Ventana tkinter junto con backend	Commit final back y front unidos	Codigo ventana tkinter junto con back	09/05/2024	Completo	implementar el funcionamiento de la interacción entre usuario y chatboot
	Proyecto terminado, Presentacion de proyecto terminada	Entrega final, Presentacion de proyecto	Proyecto final, Presentacion del proyecto	14/05/2024	Completo	probar el correcto funcionamiento de la interfaz

Responsable	Objetivo	Resultado	Entregable	Fecha	Status	
David	Objetivo del proyecto, Delimitaciones y herramientas	Envio de correo electronico	Correo electronico	30/04/2024	Completo	**Objetivos individuales**
	Objetivos individuales	Tareas para cada integrante	Objetivos en cronograma	02/05/2024	Completo	Objetivos individuales
	Implementacion de modelo	Commit de modelo en repositorio	Modelo funcionando	07/05/2024	Completo	Entrenar al modelo con un libro
	Resolver problemas con back y front	Commit final de proyecto terminado	Correcciones de codigo	09/05/2024	Completo	Obtener un modelo preentrenado
	Proyecto terminado, Presentacion de proyecto terminada	Entrega final, Presentacion de proyecto	Proyecto final, Presentacion del proyecto	14/05/2024	Completo	Probar que el chatboot realiza respuestas coherentes

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.

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.

Arquitectura del proyecto

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.

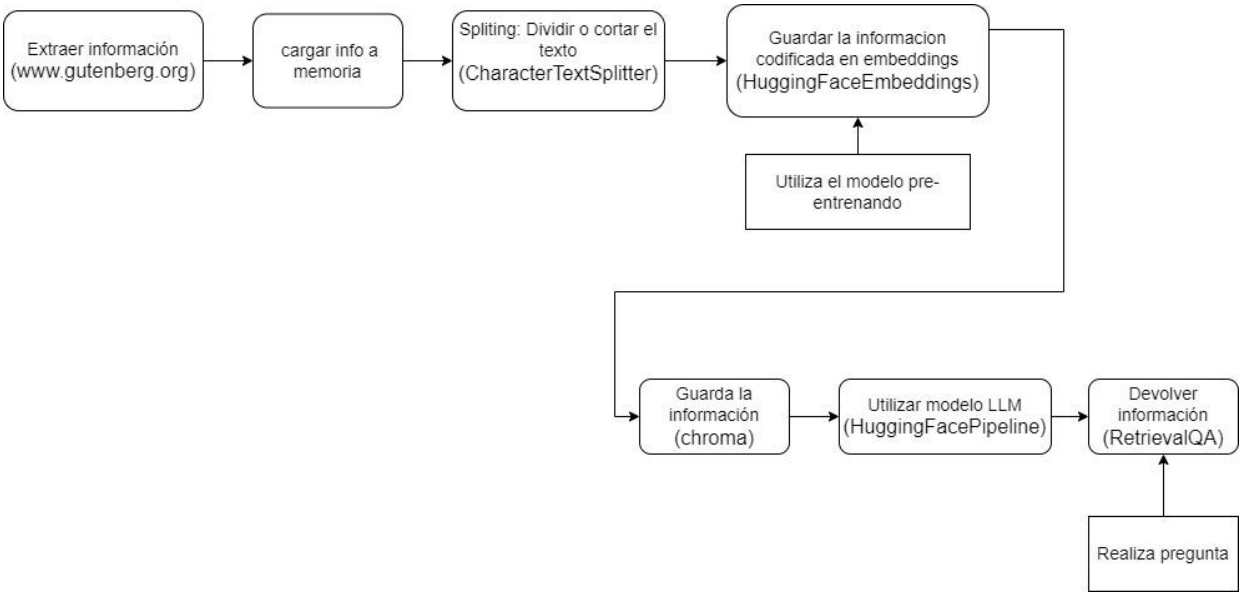
Generación de Respuestas:

- 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.

Diagrama del funcionamiento del proyecto



Problemas encontrados y las acciones tomadas para resolverlos.

Actualización de la función utilizada:
La función principal que estaba utilizando en el proyecto se actualizó. Esta actualización modificó la manera en que la función operaba, lo que resultó en que mi implementación previa ya no funcionara como estaba originalmente diseñada. Esta incompatibilidad generó problemas al implementar el proyecto de manera local y adaptarla a la interfaz de usuario (GUI).

Solución:
Utilizar la función implementada por el modelo tal como estaba originalmente y adapté la interfaz de usuario para trabajar con Google Colab. Esta adaptación permite mantener la funcionalidad sin necesidad de realizar cambios significativos en el código existente. Google Colab proporcionó un entorno accesible y colaborativo para ejecutar y probar la función, facilitando la integración con el modelo original.

Incompatibilidad entre versiones de librerías de python:
Al intentar actualizar las librerías de Python a sus versiones más recientes, surgió un problema de incompatibilidad. Las nuevas versiones de ciertas librerías no eran compatibles con otras librerías que también eran necesarias para el proyecto. Esta incompatibilidad provocó conflictos que impidieron que el entorno de desarrollo funcionara correctamente, obligándome a identificar versiones específicas de cada librería que pudieran coexistir sin problemas.

Solución:
Se optó por no actualizar a las versiones más nuevas de cada una de las librerías. En su lugar, utilicé el modelo tal como fue encontrado en Kaggle, lo que permitió mantener un entorno estable y funcional sin los conflictos de versiones. Esto implicó trabajar con las versiones específicas de las librerías que se utilizaron en la implementación original del modelo en Kaggle, asegurando así la compatibilidad y estabilidad del entorno de desarrollo.

Imposibilidad de implementar la función actualizada:
A pesar de los esfuerzos para adaptar el proyecto a la función actualizada, no fue posible implementarla de manera efectiva. La nueva versión de la función requería cambios significativos en la estructura del código y en la lógica de implementación que, debido a limitaciones de tiempo y recurso. Esto resultó en la necesidad de buscar alternativas o soluciones temporales para mantener el progreso del proyecto.

Solución:
Se adaptó una interfaz gráfica en Google Colab para utilizar la función tal como fue encontrada en el modelo original. Esta solución permitió mantener la funcionalidad del proyecto utilizando la versión original de la función sin necesidad de realizar cambios estructurales en el código. La interfaz gráfica en Google Colab facilitó la interacción con la función y proporcionó una plataforma intuitiva para la ejecución y visualización de los resultados.

Pruebas del proyecto

Los tres integrantes del equipo realizamos pruebas al proyecto, fueron un total de 55 preguntas en total, de las cuales 31 fueron correctas, con un porcentaje de acierto de 56% y un tiempo promedio de respuesta en 49 seg.

RESPUESTAS CORRECTAS: 31 / 55	PROMEDIO DE TIEMPO:
PORCENTAJE DE ACIERTOS: 56%	48.65.

Repositorio del proyecto

El código del proyecto se encuentra disponible en el repositorio: [https://github.com/davidvaladez09/Red-Neuronal-Generativa-para-Consulta-de-libros].

Explicación del código

```
import os
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
from ipywidgets import widgets, Layout, VBox, HBox
from IPython.display import display
```

```
import time
import threading
```

Se importan bibliotecas y módulos necesarios para cargar, procesar y manejar documentos textuales, integrar modelos de lenguaje y crear interfaces de usuario para AnswerBook. Utiliza herramientas de LangChain para gestión de conversaciones, memoria de chat, recuperación de documentos y creación de interfaces gráficas.

```
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)




cache_dir = "/content/"
```

Aquí configura opciones de visualización de pandas para mostrar todas las columnas, filas y elementos de secuencias sin truncamiento, y establece un ancho máximo de columna de 500 caracteres. Se define el directorio de caché en "/content/".

```
%%capture

!pip install chromadb==0.4.10 tiktoken==0.3.3 sqlalchemy==2.0.15
!pip install langchain==0.0.249
!pip install --force-reinstall pydantic==1.10.6
!pip install sentence_transformers
!pip install ipywidgets

os.environ["HUGGINGFACEHUB_API_TOKEN"] = "Fill"
```

Se realiza la instalación de varias bibliotecas necesarias, como “chromadb”, “tiktoken”, “sqlalchemy”, “langchain”, “pydantic”, “sentence_transformers” e “ipywidgets”, utilizando pip, y establece la variable de entorno “HUGGINGFACEHUB_API_TOKEN” con un valor que debe ser proporcionado por el usuario.

```
# 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 + " .......")
```

Carga el texto desde Project Gutenberg utilizando “GutenbergLoader”, guarda el contenido en la variable “document”, y luego extrae y muestra las primeras 100 palabras del texto.

```
# 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
```

Este fragmento utiliza “CharacterTextSplitter” para dividir el texto en fragmentos de tamaño 1024 con una superposición de 256 caracteres. Codifica estos fragmentos utilizando el modelo “sentence-transformers/all-MiniLM-L6-v2”, almacenando las representaciones en la variable “embeddings”.

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

Este fragmento de código guarda los fragmentos de texto codificados en una base de datos vectorial utilizando Chroma, con las representaciones embebidas previamente calculadas, y almacena los datos en el directorio especificado por “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,
    },
)
```

Este código crea un “retriever” a partir de la base de datos vectorial para buscar documentos similares, y configura una cadena de QA utilizando un modelo de lenguaje grande (LLM) de Hugging Face (“google/flan-t5-large”) para interpretar el lenguaje, realizar la recuperación de documentos y especificar cómo manejar estos datos.

```
# 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 = ""
```

Esta función maneja las consultas del usuario, desactiva la entrada mientras procesa la consulta, genera la respuesta utilizando “RetrievalQA” con el modelo de Hugging Face, mide el tiempo de respuesta, y muestra tanto la consulta del usuario como la respuesta generada por AnswerBook, reactivando la entrada después.

```
# 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()
```

Esta parte del código crea una interfaz de chat para AnswerBook. La función “actualizar_contador” actualiza continuamente el tiempo transcurrido desde el inicio del chat, mientras se muestran los mensajes de entrada y salida en un diseño de chat.

Código del proyecto

```
import os
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
from ipywidgets import widgets, Layout, VBox, HBox
from IPython.display import display
import time
import threading

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)




cache_dir = "/content/"

%%capture

!pip install chromadb==0.4.10 tiktoken==0.3.3 sqlalchemy==2.0.15
!pip install langchain==0.0.249
!pip install --force-reinstall pydantic==1.10.6
!pip install sentence_transformers
!pip install ipywidgets

os.environ["HUGGINGFACEHUB_API_TOKEN"] = "Fill"


# 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,
    },
)



# 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()
```