

Descripción de la Solución

Objetivo: Mi objetivo fue desarrollar un Asistente Conversacional para responder consultas relacionadas con recursos humanos, usando datos en tiempo real obtenidos de la API de LinkedIn. Implementé un sistema basado en Generación Aumentada por Recuperación (RAG), que permite que, cuando una pregunta no se pueda responder directamente con los datos entrenados, el sistema consulte una base de datos vectorial y obtenga respuestas utilizando un modelo de embeddings (Cohere) y FAISS.

Tecnologías y Herramientas Utilizadas:

- **LangChain:** Utilicé LangChain para construir la lógica de procesamiento de consultas e integrarme con el modelo de embeddings (Cohere) y el motor de búsqueda (FAISS).
- **Streamlit:** Utilicé Streamlit para crear la interfaz de usuario interactiva.
- **API de LinkedIn (RapidAPI):** Utilicé esta API para obtener datos en tiempo real sobre personas, empleos y empresas.
- **FAISS:** Utilicé FAISS para realizar búsquedas semánticas dentro de la base de conocimiento.

Base de Datos Vectorial

Construí la base de datos vectorial usando FAISS, que me permitió realizar búsquedas rápidas basadas en similitudes semánticas. FAISS indexa los embeddings generados a partir de las consultas de los usuarios y luego ordena los resultados según la similitud con la consulta.

Código de FAISS: La función `sort_with_langchain_faiss()` maneja el proceso de indexado y recuperación de resultados usando FAISS.

Fuente de Conocimiento: La base de conocimiento proviene de los datos obtenidos de la API de LinkedIn y se complementa con la capacidad de búsqueda semántica proporcionada por FAISS.

Modelos utilizados:

- **EMBEDDINGS:** embed-multilingual-v3.0 de Cohere.
- **MML:** command-xlarge-nightly de Cohere.
- **API:** RAPIDAPI.

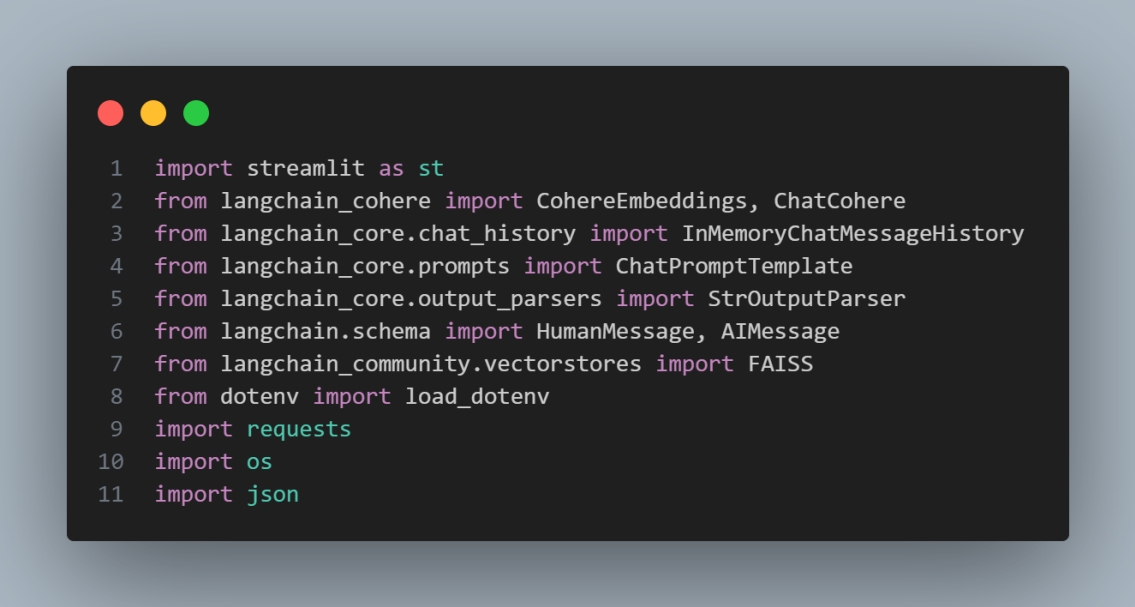
Proceso General

1. **Carga de Documentos:** Se cargan documentos o datos externos (como los de la API de LinkedIn) para alimentar la base de conocimiento.
2. **Creación de Embeddings:** Los documentos se convierten en embeddings utilizando el modelo de Cohere.
3. **Almacenamiento en Vector Store:** Los embeddings se almacenan en FAISS para permitir búsquedas rápidas.
4. **Consulta del Usuario:** Cuando el usuario hace una consulta, esta se convierte en un embedding utilizando Cohere.
5. **Recuperación por Similitud:** El embedding de la consulta se compara con los embeddings almacenados en FAISS.
6. **Creación del Prompt:** Con los documentos recuperados, se crea un prompt que se utiliza para generar la respuesta.
7. **Generación de la Respuesta:** Cohere genera la respuesta usando el prompt.
8. **Salida Formateada:** La respuesta es presentada al usuario a través de la interfaz de Streamlit.

Descripción del Código

1. Importación de Librerías y Configuración Inicial

Librerías Importadas

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays 11 lines of Python code for importing various libraries.

```
1 import streamlit as st
2 from langchain_cohere import CohereEmbeddings, ChatCohere
3 from langchain_core.chat_history import InMemoryChatMessageHistory
4 from langchain_core.prompts import ChatPromptTemplate
5 from langchain_core.output_parsers import StrOutputParser
6 from langchain.schema import HumanMessage, AIMessage
7 from langchain_community.vectorstores import FAISS
8 from dotenv import load_dotenv
9 import requests
10 import os
11 import json
```

Propósito de cada librería:

1. **streamlit**: Se utiliza para construir la interfaz de usuario interactiva.
2. **langchain_cohere**:
 - CohereEmbeddings: Convierte texto en vectores para búsquedas semánticas.
 - ChatCohere: Maneja el modelo de chat basado en Cohere.
3. **langchain_core**: Proporciona herramientas para la gestión de historial de chat, plantillas de prompts y parsing de respuestas.
4. **requests**: Realiza solicitudes HTTP para interactuar con la API de LinkedIn.
5. **dotenv**: Carga variables sensibles (como claves API) desde un archivo .env.
6. **os** y **json**: Manejo del entorno y formatos de datos.
7. **time**: Controla las pausas necesarias en caso de límites de solicitudes.

2. Funciones de Configuración y Variables

```
1 # %% Cargar variables de entorno
2 load_dotenv()
3
4 # %% Configurar claves API
5 RAPIDAPI_KEY = os.getenv("RAPIDAPI_KEY")
6 COHERE_API_KEY = os.getenv("COHERE_API_KEY")
7 BASE_URL = "https://linkedin-data-api.p.rapidapi.com"
```

Explicación: En esta sección, cargo las claves API necesarias para interactuar con la API de LinkedIn y con Cohere, utilizando el archivo .env para mantener estas claves privadas.

3. Inicialización del Modelo de Embeddings

```
1 # %% Configurar Cohere y embeddings
2 cohere = ChatCohere(model="command-xlarge-nightly", api_key=COHERE_API_KEY, temperature=0.7)
3 embeddings = CohereEmbeddings(model="embed-multilingual-v3.0")
4
```

Explicación: Aquí inicializo el modelo de chat de Cohere, que se utiliza para generar las respuestas del asistente, y también configuro los embeddings de Cohere para convertir el texto en vectores que luego se pueden indexar y buscar.

4. Headers para importación de API

```
1 HEADERS = {
2     "x-rapidapi-key": RAPIDAPI_KEY,
3     "x-rapidapi-host": "linkedin-data-api.p.rapidapi.com",
4     "Accept": "application/json"
5 }
```

5. Funciones para Interacción con la API de LinkedIn

```
1 def search_people(**kwargs):
2     """Realiza una búsqueda de personas utilizando los filtros proporcionados."""
3     params = {k: v for k, v in kwargs.items() if v is not None}
4     return safe_api_call(f"{BASE_URL}/search-people", params)
5
6 def search_jobs(**kwargs):
7     """Realiza una búsqueda de trabajos utilizando los filtros proporcionados."""
8     params = {k: v for k, v in kwargs.items() if v is not None}
9     return safe_api_call(f"{BASE_URL}/search-jobs-v2", params)
```

Explicación: Estas funciones permiten realizar consultas a la API de LinkedIn para buscar personas, obtener detalles de perfiles y empresas, y realizar búsquedas de trabajos. Utilizo el método `safe_api_call()` para manejar errores y asegurarme de que las respuestas de la API sean procesadas correctamente.

6. Función de Análisis de Intenciones (LangChain)

```
1 def create_chain(question):
2     """Crea una cadena LangChain para interpretar la consulta del usuario."""
3     template = """
4     Actúas como un asistente de recursos humanos especializado en interpretar consultas en lenguaje natural.
5     Dado el texto de entrada, interpreta la intención del usuario para determinar qué tipo de búsqueda desea realizar.
6
7     Si la consulta está relacionada con buscar personas, empleados, desarrolladores o similares en ese momento, dirás "Intención: Personas".
8
9     Si la consulta se refiere a trabajos, empleos, ofertas laborales o similares en ese momento, dirás "Intención: Trabajos".
10
11    Si la consulta no se ajusta a ninguno de los casos anteriores, responderás con un mensaje genérico.
12    Si la consulta pide de alguna forma explicar que hacer, responderás con un mensaje de ayuda.
13    Respeta parametros como Ubicacion o Localización, Habilidades, Experiencia, etc.
14    No interpretes saludos o preguntas como búsquedas.
15    Entrada del usuario:
16    {question}
17    """
18    prompt = ChatPromptTemplate.from_template(template)
19    output_parser = StrOutputParser()
20    chain = prompt | cohere | output_parser
21    return chain.invoke({"question": question})
```

Explicación: Esta función utiliza LangChain para procesar las consultas del usuario. El prompt genera un template específico para interpretar la intención de la consulta, como si la pregunta es sobre personas, trabajos o empresas. LangChain ejecuta el flujo de trabajo que integra el modelo de Cohere para generar la respuesta y luego la parsea para obtener el resultado final.

7. Manejo de Errores y Llamadas a la API



```
1 def safe_api_call(url, params, max_retries=3, wait_time=2):
2     response = requests.get(url, headers=HEADERS, params=params)
3     response.raise_for_status() # Genera excepción para códigos de error HTTP
4     return response.json()
```

Explicación: La función `safe_api_call` maneja las llamadas a la API de LinkedIn de forma segura. Se encarga de manejar errores de HTTP y otros posibles fallos de red, asegurando que el sistema no falle si hay problemas con la conexión, en este caso no hay validaciones ni depuraciones porque las quite al final ya que no había errores de conexión de API.

8. Base de Datos Vectorial con FAISS



```
1 def sort_with_langchain_faiss(response_json, key="name"):
2
3     # Extraer los textos que se desean ordenar
4     items = response_json.get("items", [])
5     texts = [item.get(key, "") for item in items]
6
7     # Verificar si hay datos válidos
8     if not texts:
9         return items # Si no hay textos, regresar la lista original sin cambios
10
11     # Crear el índice FAISS utilizando los textos
12     faiss_store = FAISS.from_texts(texts, embeddings)
13
14     # Consultar para obtener los textos en orden de similitud
15     # Usamos un texto representativo para buscar ("clave genérica")
16     query = "orden semántico"
17     search_results = faiss_store.similarity_search(query, k=len(texts))
18
19     # Ordenar los elementos originales según los resultados de FAISS
20     sorted_items = [items[texts.index(result.metadata["text"])] for result in search_results]
21
22     return sorted_items
```

Explicación: En esta función utilizo FAISS para indexar y buscar los resultados semánticamente más relevantes en función de la consulta del usuario.


8. Memoria y Historial

1. Uso de `st.session_state` en Streamlit

`st.session_state` permite mantener datos persistentes durante una sesión en Streamlit. Se utiliza para almacenar el historial de mensajes de la conversación.

Código:

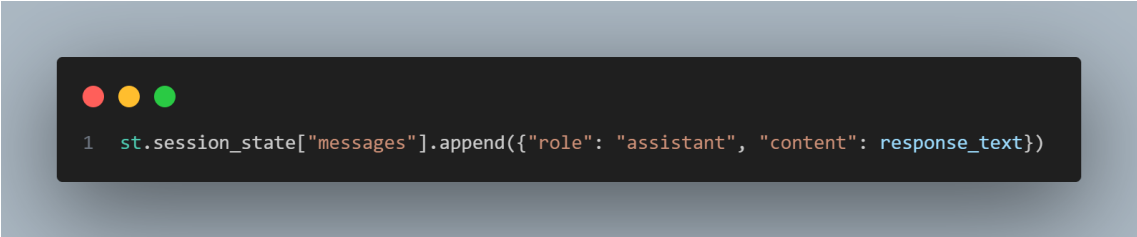
- **Inicialización del historial:**



```
1 def initialize_chat():
2     """Inicializa los mensajes en la sesión."""
3     if "messages" not in st.session_state:
4         st.session_state.messages = []
```

Explicación: Si no existe la clave "messages", se crea una lista vacía.

- **Almacenamiento de respuestas de la IA:**



```
1 st.session_state["messages"].append({"role": "assistant", "content": response_text})
```

Explicación: La respuesta de la IA se guarda como un diccionario con el rol "assistant".

- **Sincronización entre LangChain y `st.session_state`:**



```
1 def update_chat_history(chat_history):
2     """Actualiza el historial de mensajes desde chat_history a session_state."""
3     for message in chat_history.messages:
4         role = "user" if isinstance(message, HumanMessage) else "assistant"
5         if {"role": role, "content": message.content} not in st.session_state.messages:
6             st.session_state.messages.append({"role": role, "content": message.content})
```


Explicación: Se sincroniza el historial de LangChain con `st.session_state`.

2. Gestión del Historial con `InMemoryChatMessageHistory`

`InMemoryChatMessageHistory` es una clase de LangChain que almacena los mensajes como objetos estructurados, lo que facilita la gestión del historial de manera más avanzada.

Código:

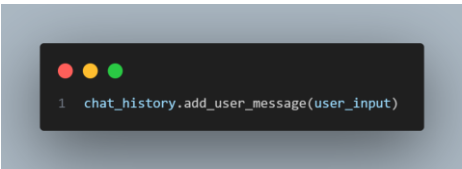
- **Inicialización del historial en LangChain:**



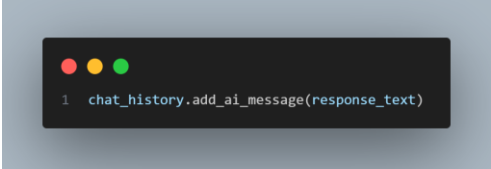
```
1 def get_session_history(session_id: str):
2     """Obtiene o inicializa el historial de mensajes de la sesión actual."""
3     if session_id not in st.session_state.store:
4         st.session_state.store[session_id] = InMemoryChatMessageHistory()
5     return st.session_state.store[session_id]
```

Explicación: Se crea un historial único para cada sesión identificado por `session_id`.

- **Agregar mensajes al historial de LangChain:**



```
1 chat_history.add_user_message(user_input)
```



```
1 chat_history.add_ai_message(response_text)
```

Explicación: Los mensajes del usuario y de la IA se agregan al historial utilizando métodos de LangChain.

10. Interfaz de Usuario con Streamlit

```
1  import streamlit as st
2
3  # Inicializa el historial de mensajes
4  def initialize_chat():
5      if "messages" not in st.session_state:
6          st.session_state.messages = []
7
8  # Maneja la entrada del usuario y la respuesta de la IA
9  def handle_user_input():
10     user_input = st.text_input("Escribe tu mensaje", key="user_input")
11     if st.button("Enviar") and user_input:
12         st.session_state.messages.append({"role": "user", "content": user_input})
13         response_text = get_ai_response(user_input)
14         st.session_state.messages.append({"role": "assistant", "content": response_text})
15         st.markdown(f"**IA**: {response_text}")
16
17 # Simula la respuesta de la IA
18 def get_ai_response(user_input):
19     return f"Respuesta simulada para: {user_input}"
20
21 # Muestra el historial de conversación
22 def display_chat_history():
23     for message in st.session_state.messages:
24         role = "Tú" if message["role"] == "user" else "IA"
25         st.markdown(f"**{role}**: {message['content']}")
26
27 # Reinicia el chat
28 def reset_chat():
29     if st.button("Recargar Chat"):
30         st.session_state.messages = []
31
32 # Función principal de la interfaz de usuario
33 def main():
34     st.title("Chat con la IA")
35     st.subheader("Habla con la IA y recibe respuestas instantáneas")
36     initialize_chat()
37     display_chat_history()
38     handle_user_input()
39     reset_chat()
40
41 if __name__ == "__main__":
42     main()
```

Explicación: Esta es la función principal que gestiona la interfaz de usuario. Utilizo Streamlit para permitir la entrada de texto, procesar la consulta y mostrar la respuesta.

Entregables

1. **Descripción de la solución:** Expliqué arriba.
2. **Fuente de Conocimiento:** Base de datos vectorial construida a partir de los embeddings generados y la API de LinkedIn.
3. **Código Fuente:** Repositorio de GitHub bien estructurado.
4. **Archivo `requirements.txt`:** Lista de dependencias necesarias.
5. **Comentarios y Buenas Prácticas:** Comentarios en el código y mejoras sugeridas.
6. **Uso de Herramientas:** CohereEmbeddings y FAISS para la búsqueda semántica.
7. **Acceso a Internet:** Integración con la API de LinkedIn.
8. **Funcionalidades Enriquecidas:** Agregar más funcionalidades según se necesite.

Resumen de Funcionalidades Clave

- Generación Aumentada por Recuperación (RAG) con Cohere y FAISS.
- Interfaz de usuario interactiva con Streamlit.
- Búsqueda semántica mediante FAISS.
- Gestión del historial con `st.session_state`.
- Integración con la API de LinkedIn para obtener datos en tiempo real.