Introducción a RAG

La generación aumentada por recuperación (RAG) es el proceso de optimización de la salida de un modelo de lenguaje de gran tamaño, de modo que haga referencia a una base de conocimientos autorizada fuera de los orígenes de datos de entrenamiento antes de generar una respuesta. Los modelos de lenguaje de gran tamaño (LLM) se entrenan con volúmenes de datos amplios y usan miles de millones de parámetros para generar resultados originales en tareas como responder preguntas, traducir idiomas y completar frases. La RAG extiende las ya poderosas capacidades de los LLM a dominios específicos o a la base de conocimientos interna de una organización, todo ello sin la necesidad de volver a entrenar el modelo. Se trata de un método rentable para mejorar los resultados de los LLM de modo que sigan siendo relevantes, precisos y útiles en diversos contextos.

¿Cuáles son los beneficios de la generación aumentada por recuperación?

La tecnología RAG aporta varios beneficios a los esfuerzos de la IA generativa de una organización.

- **Implementación rentable:** El desarrollo de chatbots normalmente comienza con un modelo fundacional. Los modelos fundacionales (FM) son LLM accesibles para API entrenados en un amplio espectro de datos generalizados y sin etiquetar. Los costos computacionales y financieros de volver a entrenar a los FM para obtener información específica de la organización o del dominio son altos. La RAG es un enfoque más rentable para introducir nuevos datos en el LLM. Hace que la tecnología de inteligencia artificial generativa (IA generativa) sea más accesible y utilizable.

- **Información actual:** Incluso si los orígenes de datos de entrenamiento originales para un LLM son adecuados para sus necesidades, es difícil mantener la relevancia. La RAG les permite a los desarrolladores proporcionar las últimas investigaciones, estadísticas o noticias a los modelos generativos. Pueden usar la RAG para conectar el LLM de manera directa a redes sociales en vivo, sitios de noticias u otras fuentes de información que se actualizan con frecuencia. El LLM puede entonces proporcionar la información más reciente a los usuarios.

- **Mayor confianza de los usuarios:** La RAG le permite al LLM presentar información precisa con la atribución de la fuente. La salida puede incluir citas o referencias a fuentes. Los usuarios también pueden buscar ellos mismos los documentos de origen si necesitan más aclaraciones o más detalles. Esto puede aumentar la confianza en su solución de IA generativa.

- **Más control para los desarrolladores:** Con la RAG, los desarrolladores pueden probar y mejorar sus aplicaciones de chat de manera más eficiente. Pueden controlar y cambiar las fuentes de información del LLM para adaptarse a los requisitos cambiantes o al uso multifuncional. Los desarrolladores también pueden restringir la recuperación de información confidencial a diferentes niveles de autorización y garantizar que el LLM genere las respuestas adecuadas. Además, también pueden solucionar problemas y hacer correcciones si el LLM hace referencia a fuentes de información incorrectas para preguntas específicas. Las organizaciones pueden implementar la tecnología de IA generativa con mayor confianza para una gama más amplia de aplicaciones.

Implementación con Langhain

1. Cargamos el documento con **PyPDFLoader** y lo convertimos en un iterable. Luego recorremos ese iterable y guardamos cada page en una variable.

```
from langchain_community.document_loaders import PyPDFLoader
# cargar el documento
document = PyPDFLoader("LANGCHAIN.pdf")

# creamos el iterador
loader = document.lazy_load()

text = ""
# recorremos el contenido del loader
for page in loader:
    text += page.page_content
```

2. Importamos nuestro separador de texto, lo implementamos y creamos los documentos que cargaremos posteriormente en nuestra base de datos vectorial:

```
from langchain_text_splitters import CharacterTextSplitter
# creamos nuestro separador de texto
text_splitter = CharacterTextSplitter(
    chunk_size=2000,
    chunk_overlap=100,
    separator="\n"
)
# creamos los documentos para cargar nuestra base de datos
vectorial
texts = text_splitter.create_documents([text])
```

3. Importamos OllamaEmbedding desde langchain_ollama para utilizar el modelo de embedding

```
from langchain_ollama import OllamaEmbeddings
# creamos la comunicacion con nuestro modelo de embedding
embedding = OllamaEmbeddings(
    model="nomic-embed-text"
)
```

4. Ahora, importamos Chroma desde langchain_chroma y creamos nuestra vector store

```
from langchain_chroma import Chroma

# declaramos la base de datos vectorial
vector_store = Chroma(
    collection_name="test",
    embedding_function=embedding,
    persist_directory="./vectorstore-nomic"
)
```

5. Cargamos los documentos que creamos en el punto 2

```
# creamos los embeddings y lo guardamos en la base de datos
vectorial
vector_store.add_documents(texts)
```

6. En este punto, ya tenemos documentos en nuestra base de datos vectorial, por lo tanto, vamos a crear nuestro llm que responderá preguntas utilizando la información que guardamos

```
from langchain_google_genai import ChatGoogleGenerativeAI
from dotenv import load_dotenv
import os
load_dotenv()

api_key = os.getenv("API_KEY")

llm = ChatGoogleGenerativeAI(
    api_key=api_key,
    model = "gemini-2.5-flash",
    temperature=0.5
)
```

7. Creamos un prompt_template para nuestro llm con ChatPromptTemplate

```
from langchain_core.prompts import ChatPromptTemplate
ai_msg = [""]
human_msg = []

prompt_template = ChatPromptTemplate.from_messages([
    ("system","""
    Eres un asistente encargado de responder preguntas sobre
Langchain.
                  Responde solo si  {context} posee contenido. Si
el contexto esta vacio,
                  responde "No tengo suficiente informacion para
reponder esa pregunta"
    """),
    ("ai","{ai_msg}"),
    ("human","{human_msg}")
])
```

8. Casi terminamos, ahora vamos a declarar una variable donde utilizaremos un input, para probar el funcionamiento de nuestro rag.

```
# solicitud del usuario
input_user = input("Human: ")
human_msg.append(input_user)
```

9. La solicitud del punto 8, utilizaremos para hacer una búsqueda vectorial con similitary_search. Luego, pasaremos un diccionario a nuestro prompt_template para rellenar las variables que está esperando.

```
docs = vector_store.similarity_search(input_user, k=10)

prompt = prompt_template.invoke( {
    "context": docs,
    "ai_msg": ai_msg,
    "human_msg": human_msg
})
```

10. Invocamos el modelo con el método **invoke**

```
response = llm.invoke(prompt)

print(response.content)
```

Resultado obtenido:

```
┌── ~\Desktop\langchain
└──  Human: que son los prompt templates?
AI: Los Prompt Templates en Langchain son clases que permiten
crear plantillas de mensajes reutilizables. Estas plantillas
pueden definir variables (encerradas entre llaves `{}`) que se
llenarán con valores específicos al momento de invocar el modelo.
Existe también la clase `ChatPromptTemplate`, que a diferencia de
`PromptTemplate`, espera un diccionario con las variables
declaradas en el template.
```

Actividades

Todo el contenido de la actividad, debe ser entregado en un júpiter notebook, con sus correspondientes celdas de markdown para la documentación y celdas de código.

1. Implementación Práctica de RAG:

- Recrea el ejemplo dado en un Jupyter Notebook o entorno de desarrollo similar, utilizando las librerías langchain_community.document_loaders, langchain_text_splitters, langchain_ollama, langchain_chroma, y langchain_google_genai.
- Verifica el funcionamiento del sistema RAG respondiendo a preguntas relacionadas con el contenido de un documento X (deben proporcionar uno ustedes).

2. Exploración de Modelos de Embedding y LLM:

- Experimenta con diferentes modelos de embedding disponibles en langchain_ollama (además de "nomic-embed-text") y analiza cómo impactan en la calidad de la recuperación de documentos.
- Prueba con otros modelos de lenguaje de gran tamaño (LLM) compatibles con langchain_google_genai o langchain_ollama y compara sus respuestas y rendimiento.

3. Optimización del Separador de Texto:

- Modifica los parámetros chunk_size y chunk_overlap del CharacterTextSplitter y observa cómo afectan a la creación de los documentos y, consecuentemente, a la precisión de las respuestas del LLM.
- Investiga otros tipos de separadores de texto disponibles en Langchain y evalúa su idoneidad para diferentes tipos de documentos.

4. Gestión de Bases de Datos Vectoriales:

- Explora las funcionalidades de Chroma para gestionar colecciones, persistencia de datos y realizar búsquedas más avanzadas (por ejemplo, filtrado de resultados).
- Investiga otras bases de datos vectoriales compatibles con Langchain y considera sus ventajas y desventajas para diferentes casos de uso.

5. Refinamiento de Prompts:

- Experimenta con diferentes prompt_template para el LLM, ajustando las instrucciones del sistema y los mensajes de IA/humano para mejorar la calidad y relevancia de las respuestas.
- Implementa técnicas de ingeniería de prompts para guiar al LLM a proporcionar respuestas más precisas y útiles, incluyendo la atribución de fuentes cuando sea posible.