Práctica 3.6

Procesamiento del lenguaje natural

Chatbot de consultas a documentos PDF

Enunciado

Objetivo

En esta práctica exploraremos aspectos no tratados acerca de Langchain para crear un asistente virtual que permita la consulta de información a documentos propios (como pueden ser documentos PDF).

Crear un chatbot de este tipo implica recurrir a los siguientes componentes/acciones de Langchain:

- **Cargador de documentos**: para cargar varios formatos de datos y crear objetos de documentos (aquí PDF)
- **Fragmentación**: fragmentar los documentos mediante divisores de texto.
- **Incrustación**: incrustar los fragmentos para generar vectores.
- **Almacén de vectores**: para almacenar e indexar documentos vectoriales (aquí usaremos el que proporciona la librería *Chroma db*)
- **LLM**: modelo de lenguaje para responder preguntas y resumir
- **Recuperador de documentos**: que recupera los fragmentos relevantes según la consulta del documento PDF

Articularemos los componentes y acciones previas a través de los siguientes pasos:

1. Instalación de librerías y claves
2. Acceso al PDF y creación del índice
3. Creación de una cadena con LLM
4. Creación del interfaz web para el chatbot

PASO 1 :: Instalación de librerías y claves

Realiza los siguientes pasos:

1. Crea un nuevo cuaderno dentro del directorio donde creaste el proyecto para el chatbot en la práctica anterior.

2. Instala con **poetry** las librerías: **pydantic**, **chromadb**, **sentence-transformers** y **gradio**

3. Importa las librerías anteriores:

```
import os
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import Chroma
from langchain import HuggingFaceHub
from langchain.chains import RetrievalQA
```

4. Ahora, indica tu clave de Huggingface:

```
#loading the API key
os.environ['HUGGING_FACE_HUB_API_KEY'] = "<token_hugginface>"
```

PASO 2 :: Acceso al PDF y creación del índice

5. El siguiente paso supone cargar el PDF con el que vamos a interactuar y crear el almacén de vectores (vector store)

```
path = input("Introduce la ruta al fichero PDF: ")
loader = PyPDFLoader(path)
pages = loader.load()
```

6. Comprueba cuántas páginas tiene el documento cargado y visualiza además la primera de ellas.

7. Antes de crear el índice (es decir, el almacén de vectores) necesitamos fragmentar el contenido del documento más allá de la división en páginas. O dicho de otro modo, el almacén de vectores funciona a un nivel inferior a la página donde lo que se vectoriza son fragmentos simples. La invocación a `RecursiveCharacterTextSplitter` que se hace a continuación tiene ese propósito. **¿Qué significado imaginas que tiene sus dos argumentos?**

```
splitter = RecursiveCharacterTextSplitter(chunk_size=500,
chunk_overlap=20)
docs = splitter.split_documents(pages)
```

Muestra cuántos fragmentos se han creado (variable "docs").

8. Para la creación del *vector store* recurriremos al objeto "Chroma", el cual recibe dos argumentos: los fragmentos junto a un vectorizador (de HugginFace):

```
embeddings = HuggingFaceEmbeddings()
doc_search = Chroma.from_documents(docs, embeddings)
```

La creación del vector store arroja una salida de este tipo:

/home/toni-pc/.cache/pypoetry/virtualenvs/llms-langchain-DoCdqqJ--py3.11/lib/python3.11/site-packages/tqdm/auto.py :21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
from .autonotebook import tqdm as notebook_tqdm
.gitattributes: 100%|| 1.18k/1.18k [00:00<00:00, 7.14MB/s]
1_Pooling/config.json: 100%|| 190/190 [00:00<00:00, 1.29MB/s]
README.md: 100%|| 10.6k/10.6k [00:00<00:00, 56.5MB/s]
config.json: 100%|| 571/571 [00:00<00:00, 3.72MB/s]
config_sentence_transformers.json: 100%|| 116/116 [00:00<00:00, 911kB/s]
data_config.json: 100%|| 39.3k/39.3k [00:00<00:00, 394kB/s]
pytorch_model.bin: 100%|| 438M/438M [00:08<00:00, 50.2MB/s]
sentence_bert_config.json: 100%|| 53.0/53.0 [00:00<00:00, 318kB/s]
special_tokens_map.json: 100%|| 239/239 [00:00<00:00, 1.72MB/s]
tokenizer.json: 100%|| 466k/466k [00:00<00:00, 1.50MB/s]
tokenizer_config.json: 100%|| 363/363 [00:00<00:00, 3.12MB/s]
train_script.py: 100%|| 13.1k/13.1k [00:00<00:00, 77.2MB/s]
vocab.txt: 100%|| 232k/232k [00:00<00:00, 1.18MB/s]
modules.json: 100%|| 349/349 [00:00<00:00, 1.03MB/s]

9. Finalmente, probaremos el funcionamiento del *vector store* realizando alguna consulta:

```
query = "<introduce aquí el texto con tu consulta al PDF>"
similar_docs = doc_search.similarity_search(query, k=3)
similar_docs
```

Introduce una nueva consulta con una pregunta (texto) diferente y observa el resultado. En ambos casos, el vector store nos devolverá los fragmentos asociados a cada una de ellas.

PASO 3 :: Creación de una cadena con LLM

La creación de la cadena va a requerir 2 pasos:

1) creamos primero el modelo:

```
repo_id = "tiiuae/falcon-7b"
llm = HuggingFaceHub(huggingfacehub_api_token =
"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",
                     repo_id=repo_id, model_kwargs={'temperature': 0.2,
'max_length':1000})
```

2) … y actor seguido, agregamos un recuperador (retrieval) que nos permita componer la cadena combinando el modelo con el *vectorstore* anterior:

```
retrieval_chain = RetrievalQA.from_chain_type(
    llm,
    chain_type='stuff',
    retriever=doc_search.as_retriever(),
)
```

Prueba la cadena realizando una consulta a un documento. Si, por ejemplo, subiéramos el enunciado de la práctica, la consulta podría ser:

```
query = "cuál es el objetivo de la práctica"
retrieval_chain.run(query)
```

Y una posible salida podría ser similar a esto:

```
'\n\nPara que los datos sean accesibles a los usuarios\n\nPara que los
datos sean accesibles a los usuarios y a los modelos\n\nPara que los datos
sean accesibles a los usuarios y a los modelos y a los modelos de
aprendizaje\n\nPara que los datos sean accesibles a los usuarios y a los
modelos y a los modelos de aprendizaje y a los modelos de
aprendizaje\n\nPara que los datos sean accesibles a los usuarios y a los
modelos y a los'
```

PASO 4 :: Creación del interfaz web para el chatbot

Crea un fichero con extensión "py" y traslada el siguiente código, sustituyendo el token de Huggingface por el asociado a tu cuenta:

```python
import chainlit as cl

@cl.on_chat_start
def main():
    retrieval_chain = RetrievalQA.from_chain_type(llm, chain_type='stuff',
retriever=doc_search.as_retriever())
    cl.user_session.set("retrieval_chain", retrieval_chain)

@cl.on_message
async def main(message:str):
    retrieval_chain = cl.user_session.get("retrieval_chain")
    res = await retrieval_chain.acall(message, callbacks=
                                      [cl.AsyncLangchainCallbackHandler()])

    await cl.Message(content=res["text"]).send()
```

¿Qué diferencia existe en este código respecto del que usamos para el chatbot en la práctica anterior?

Finalmente, ejecuta el script que lanzará el interfaz web con el asistente, que puedes probar haciendo consultas sucesivas a un PDF. Para ello, abre una consola, sitúate en el directorio donde tengas el script anterior, y lánzalo con el siguiente comando:

```
# chainlit run <nombre_fichero.py> -w –port 8080
```

Este comando provocará que se abra el navegador con el asistente.