# Práctica 3.5

## Procesamiento del lenguaje natural

### Creación de un chatbot mediante LLM y Langchain

## Enunciado

**Objetivo**: En esta práctica nos apoyaremos en Langchain y en el uso de LLMs para crear una versión sencilla de asistente virtual.

La práctica tendrá tres partes:

1. Preparación de la cadena en **Lancghain**.
2. Creación de un interfaz tipo web que permita simular la interacción en una conversación, para lo cual usaremos el framework conocido como **Chainlit**.
3. Mejora del asistente añadiendo memoria a la conversación.

## PASO 1 :: Preparación de la cadena en Langchain

Realiza los siguientes pasos:

1. Crea un nuevo proyecto para esta práctica. Recuerda que resulta más que recomendable usar "poetry" para la instalación de los paquetes necesarios.

2. Instala las librerías: **langchain, huggingface_hub** y **chainlit**

3. Importa las librerías anteriores:

```
import os
import chainlit as cl
from langchain import HuggingFaceHub, PromptTemplate, LLMChain
```

4. Ahora, enlazaremos con un modelo LLM que aportará la base funcional de nuestro asistente:

```
model_id = "gpt2-medium"  #355M parameters
conv_model = HuggingFaceHub(
huggingfacehub_api_token = '<añade tu token de HuggingFace>',
repo_id = model_id,
model_kwargs = {"temperature":0.8, "max_new_tokens":200})
```

En la invocación del modelo en la línea, hay dos parámetros importantes incluidos en "model_kwargs" cuyo significado cabe tener presente

a) **temperature**: la temperatura indica el comportamiento del modelo, siendo **0** (**determinístico**, por tanto, misma salida para una misma entrada) y **1** (**probabilístico**, es decir, diferente salida antes una misma entrada de datos) .

b) **max_new_tokens**: número máximo de tokens que se aceptan como entrada.

Por otro lado, para este ejemplo se ha elegido la versión 2 de GPT, que por sus características, resulta adecuada al propósito de este práctica.

5. El siguiente paso supone crear una **plantilla** para la entrada del usuario. Introduce el siguiente código para ello:

```
template = """You are a helpful AI assistant that makes stories by
completing the query provided by the user
{query}
"""
prompt = PromptTemplate(template=template, input_variables=['query'])
```

En este caso, podemos observar cuál es la finalidad de plantilla: dar formato a la entrada de datos, permitiendo incluso el manejo de variables (aquí "query").

6. El siguiente paso corresponde a la definición de una cadena. Recordemos que el propósito de una cadena es aglutinar otros elementos que forman parte del *pipeline* de Langchain.

```
conv_chain = LLMChain(llm=conv_model,
                      prompt=prompt,
                      verbose=True)
print(conv_chain.run("Once upon a time in 1947"))
```

En este ejemplo, la cadena está formada por el modelo y el *prompt*. Una forma de probarla es pasarla un texto como entrada para que el modelo lo complete.

La salida debería ser algo parecido a esto:

**> Entering new LLMChain chain...**
Prompt after formatting:
**You are a helpful AI assistant that makes stories by completing the query provided by the user**

**Once upon a time in 1947**

**> Finished chain.**

The American public discovered that they had been lied to by a television network. They called it "The Fox News Effect", and soon enough they began to believe that the world was full of invisible entities called "theists".

Somehow these "theists" managed to capture the hearts of the populace, and began to become a kind of god, a sort of benevolent leader that would guide people through their daily lives, and answer questions about the world around them.

They did this by getting you to fill out an application, and they would ask you questions about the world around you, and then deliver a message that would tell your tale as to why you should care what the world thinks about you.

Since then, we humans have been running around the world, making headlines, being interviewed, and telling stories about what we think about the world. We even invented fake religion, because of course we do, and to tell the truth we have to think we know what people

## PASO 2 :: Creación del interfaz web interactivo

Dentro de un entorno tipo Jupyter, la capacidad de simular una conversación está más que limitada. En esta segunda parte, trataremos de crear un interfaz tipo web que posibilite la interacción entre el chatbot y el usuario. Para ello, vamos a recurrir a un framework llamado **Chainlit** que permite la creación de interfaces para asistentes conversacionales.

Los siguientes dos pasos permiten generar un entorno mínimo para la conversación:

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

```
import os
import chainlit as cl
from langchain import HuggingFaceHub, PromptTemplate, LLMChain
from getpass import getpass

model_id = "gpt2-medium"
conv_model = HuggingFaceHub(huggingfacehub_api_token=
                'HUGGINGFACEHUB_API_TOKEN',
                repo_id=model_id,
                model_kwargs={"temperature":0.8, "max_new_tokens":150})

template = """You are a story writer AI assistant that completes a story based on the
query received as input

{query}
"""


@cl.on_chat_start
def main():
  prompt = PromptTemplate(template=template, input_variables=['query'])
  conv_chain = LLMChain(llm=conv_model,
                prompt=prompt,
                verbose=True)

  cl.user_session.set("llm_chain", conv_chain)

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

```
#perform post processing on the received response here
#res is a dict and the response text is stored under the key "text"
await cl.Message(content=res["text"]).send()
```

La primera parte del código reproduce la funcionalidad que ya teníamos anteriormente. Sin embargo, en su parte final agrega dos funciones "main" precedidas ambas por un decorador:

1. el primero de ellos, **on_chat_start**, incorpora una cadena a la sesión, que asocia a la nueva sesión de usuario.
2. en el segundo, **on_message**, define un manejador que gestionará el intercambio asíncrono de mensajes relacionados con la sesión establecida anteriormente.

2) ahora ejecutaremos el script que lanzará el interfaz web con el asistente. 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.

## PASO 3 :: Mejora del asistente añadiendo memoria a la conversación

Más allá de la capacidad de mantener una respuesta adecuada a la entrada proporcionada (aspecto que depende de la idoneidad del modelo seleccionado), el asistente anterior tiene una problema: carece de memoria. Esto significa que no es capaz de retener lo dicho durante la conversación, haciendo que cada entrada y respuesta se den de forma aislada. En este último paso haremos alguna modificación que permita añadir esta característica.

Langchain cuenta actualmente con 2 tipos de módulos relacionados con la memoria:

a) **Conversation Buffer Memory**: permite al modelo guardar entradas previas. Tiene la contrapartida del costo computacional que genera procesar las entradas de forma acumulada, con lo cual existe una limitación en el número de tokens procesables en un tiempo razonable.

b) **Conversation Buffer Memory with Windows Size**: en este caso, se permite indicar el número de interacciones que se guardarán en la memoria.

Realicemos un ejemplo del primer caso:

1. Introduce el siguiente código

```
repo_id = 'lmsys/fastchat-t5-3b-v1.0'
llm = HuggingFaceHub(huggingfacehub_api_token =
os.environ['HUGGING_FACE_HUB_API_KEY'],
             repo_id=repo_id,
                    model_kwargs = {'temperature': 1e-10,
"max_length":32})

from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()
conversation_buf = ConversationChain(
    llm=llm,
    memory=memory)
```

Se ha usado el modelo **fastchat-t5-3b-v1.0** entre las diversas opciones disponibles en HuggingFace.

2. Haremos una prueba de funcionamiento del modelo:

```
conversation_buf.predict(input="Hi! My name is Pepe. I do have some
questions for you")
```

El modelo devolverá el saludo en su respuesta.

3. Ahora definiremos 3 preguntas que conformarán la conversación.

```
query1 = "I live in Spain. Who was the first President in democracy?"
```

```
query2 = "Summarize the history"
query3 = "what is my name?"
```

Y lanzamos la primera pregunta:

```
conversation_buf.predict(input=query1)
```

El resultado será algo así:

```
'<pad>`< pad>\n < pad> The first President in democracy in Spain was Juan Carlos I'
```

Podemos revisar que se ha guardado en memoria:

```
memory.load_memory_variables({})
```

```
{'history': "Human: Hi! My name is Pepe. I do have some questions for you\nAI: <pad> < pad>\n Hello Pepe! I'm here to help you with any questions you \nHuman: I live in Spain. Who was the first President in democracy?\nAI: <pad>`< pad>\n < pad> The first President in democracy in Spain was Juan Carlos I"}
```

4. Lanza ahora por tu cuenta las 2 siguientes preguntas y comprueba con la siguiente línea el estado de la memoria:

```
print(memory.buffer)
```

**¿Se ha guardado toda la conversación?**

## Ejercicio:

A continuación, prueba por tu cuenta las otra variante de memoria apuntada anteriormente, para lo cuál deberás consultar la página oficial de **Langchain**, concretamente el apartado de la documentación donde se refiere este aspecto.