**Trabajo Final Procesamiento del Lenguaje Natural**

**2023**

**Tomás Navarro Miñón**

Introducción:

La Generación Aumentada por Recuperación (RAG) es el concepto de proporcionar a los Modelos de Lenguaje de Gran Escala (LLMs) información adicional proveniente de una fuente de conocimiento externa. Esto les permite generar respuestas más precisas y contextuales, reduciendo las alucinaciones.

En términos simples, RAG es para LLMs lo que un examen con libro abierto es para los seres humanos. En un examen con libro abierto, se permite a los estudiantes llevar materiales de referencia, como libros de texto o apuntes, que pueden utilizar para buscar información relevante y responder a una pregunta. La idea detrás de un examen con libro abierto es que el examen se centra en las habilidades de razonamiento de los estudiantes en lugar de su capacidad para memorizar información específica.

Teniendo esto en cuenta nuestro sistema RAG lo hicimos en base a un autor de teoría política conocido como Nicolas Maquiavelo, él mismo fue autor de muchos libros que hoy en día se aplican en el campo de la política. Algunas de sus obras más conocidas son “El Príncipe”, “Discursos Sobre la Primera Década De Tito Livio”, entre otros.

¿Por qué realizar un sistema RAG de un autor de teoría política? Bueno tiene que ver con algo personal. Anteriormente yo estudiaba Relaciones Internacionales en la FCPOLIT. Algo que en lo personal no me gustaba de la carrera era que a la hora de rendir, tenía que leer absolutamente todo el autor, por más que muchas veces la mayoría de las cosas no eran importantes. Pero algo todavía peor es que si no entendía algo, a la única persona que le podía preguntar era al profesor en clases de consultas, por lo que si no podías ir, seguramente no apruebes. Es por esto que viendo en google drive las cosas que tenía me incline a ayudar a los alumnos de aquella carrera que, como yo, no podrian ir a clases de consultas porque vivían lejos… De esta manera llego el Chatbot RAG: “Nicolás MaquiavRAGlo” (jaja malisimo el nombre)

**Documentación TP NLP**

1- Lo primero que hacemos es instalar las librerias necesarias para llevar a cabo el proyecto:  
llama\_index sentence-transformers pypdf langchain python-decouple PyMuPDF gdown chromadb fpdf SPARQLWrapper

Luego el paso siguiente será la importación de todas las librerías a utilizar:

from langchain.embeddings.huggingface import HuggingFaceEmbeddings

from llama\_index.embeddings import LangchainEmbedding

from llama\_index import ServiceContext

from llama\_index import VectorStoreIndex, SimpleDirectoryReader

from jinja2 import Template

from decouple import config

import chromadb

import os

from llama\_index.node\_parser.text import SentenceSplitter

from llama\_index.schema import TextNode

from llama\_index.vector\_stores import ChromaVectorStore

from llama\_index.storage.storage\_context import StorageContext

import requests

from bs4 import BeautifulSoup

from fpdf import FPDF

import gdown

import shutil

2- Extracción de datos para la creación del sistema RAG

Link a los archivos del Google Drive: <https://drive.google.com/drive/folders/1AEDfhcL9aLA5vLllLmrzeYl6zaL_yHur?usp=sharing>

En este punto lo que hacemos es importar los PDFs que nos serviran para un posterior scrapping de los mismos y poder entrenar a nuestro chatbot especialista en Nicolas Maquiavelo.

Tambien en este punto realizamos web scraping con beautifulsoup a la pagina oficial de wikipedia para obtener mas informacion: <https://es.wikipedia.org/wiki/Nicol%C3%A1s_Maquiavelo>

A su vez también obtenemos información realizando una consulta a una base de datos online SQL.

3- Obtenemos el modelo de LLM que vamos a utilizar desde hugging face en este caso LLama Index.

Y tambien cargamos el modelo de LangChain de Embeddings: sentence-transformers/paraphrase-multilingual-mpnet-base-v2

def zephyr\_instruct\_template(messages, add\_generation\_prompt=True):

# Definir la plantilla Jinja

template\_str = "{% for message in messages %}"

template\_str += "{% if message['role'] == 'user' %}"

template\_str += "<|user|>{{ message['content'] }}</s>\n"

template\_str += "{% elif message['role'] == 'assistant' %}"

template\_str += "<|assistant|>{{ message['content'] }}</s>\n"

template\_str += "{% elif message['role'] == 'system' %}"

template\_str += "<|system|>{{ message['content'] }}</s>\n"

template\_str += "{% else %}"

template\_str += "<|unknown|>{{ message['content'] }}</s>\n"

template\_str += "{% endif %}"

template\_str += "{% endfor %}"

template\_str += "{% if add\_generation\_prompt %}"

template\_str += "<|assistant|>\n"

template\_str += "{% endif %}"

# Crear un objeto de plantilla con la cadena de plantilla

template = Template(template\_str)

# Renderizar la plantilla con los mensajes proporcionados

return template.render(messages=messages, add\_generation\_prompt=add\_generation\_prompt)

# Aquí hacemos la llamada el modelo

def generate\_answer(prompt: str, max\_new\_tokens: int = 768) -> None:

try:

# Tu clave API de Hugging Face

api\_key = config('HUGGINGFACE\_TOKEN')

# URL de la API de Hugging Face para la generación de texto

api\_url = "https://api-inference.huggingface.co/models/HuggingFaceH4/zephyr-7b-beta"

# Cabeceras para la solicitud

headers = {"Authorization": f"Bearer {api\_key}"}

# Datos para enviar en la solicitud POST

# Sobre los parámetros: https://huggingface.co/docs/transformers/main\_classes/text\_generation

data = {

"inputs": prompt,

"parameters": {

"max\_new\_tokens": max\_new\_tokens,

"temperature": 0.7,

"top\_k": 50,

"top\_p": 0.95

}

}

# Realizamos la solicitud POST

response = requests.post(api\_url, headers=headers, json=data)

# Extraer respuesta

respuesta = response.json()[0]["generated\_text"][len(prompt):]

return respuesta

except Exception as e:

print(f"An error occurred: {e}")

# Esta función prepara el prompt en estilo QA

def prepare\_prompt(query\_str: str, nodes: list):

TEXT\_QA\_PROMPT\_TMPL = (

"La información de contexto es la siguiente:\n"

"---------------------\n"

"{context\_str}\n"

"---------------------\n"

"Dada la información de contexto anterior, y sin utilizar conocimiento previo, responde la siguiente pregunta.\n"

"Pregunta: {query\_str}\n"

"Respuesta: "

)

# Construimos el contexto de la pregunta

context\_str = ''

for node in nodes:

page\_label = node.metadata["page\_label"]

file\_path = node.metadata["file\_path"]

context\_str += f"\npage\_label: {page\_label}\n"

context\_str += f"file\_path: {file\_path}\n\n"

context\_str += f"{node.text}\n"

messages = [

{

"role": "system",

"content": "Eres un asistente útil que siempre responde con respuestas veraces, útiles y basadas en hechos.",

},

{"role": "user", "content": TEXT\_QA\_PROMPT\_TMPL.format(context\_str=context\_str, query\_str=query\_str)},

]

final\_prompt = zephyr\_instruct\_template(messages)

return final\_prompt

Con el codigo anterior configuramos la llamada al modelo de LLM (llama index) para poder realizar prompts y obtener una respuesta. Tambien lo configuramos para que tome un rol, en este caso que sea un asistente.

4- Creamos la base de datos vectorial para almacenar los respectivos nodos creados que van a ayudar a encontrar información y etiquetar el contexto para nuestro sistema RAG

chroma\_client = chromadb.EphemeralClient()

chroma\_collection = chroma\_client.create\_collection("quickstart")

Creamos la colección de ChromaDB

text\_parser = SentenceSplitter(

chunk\_size=1024,

)

text\_chunks = []

doc\_idxs = []

for doc\_idx, doc in enumerate(documents):

cur\_text\_chunks = text\_parser.split\_text(doc.text)

text\_chunks.extend(cur\_text\_chunks)

doc\_idxs.extend([doc\_idx] \* len(cur\_text\_chunks))

nodes = []

for idx, text\_chunk in enumerate(text\_chunks):

node = TextNode(

text=text\_chunk,

)

src\_doc = documents[doc\_idxs[idx]]

node.metadata = src\_doc.metadata

nodes.append(node)

for node in nodes:

node\_embedding = embed\_model.get\_text\_embedding(

node.get\_content(metadata\_mode="all")

)

node.embedding = node\_embedding

En las líneas de código vistas anteriormente, a la data extraída de nuestros textos le realizamos Text Split, Tokenización, la transformamos en Nodos para poder añadirla a nuestra base de datos vectorial y realizamos su transformación a embeddings.

vector\_store = ChromaVectorStore(chroma\_collection=chroma\_collection)

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

service\_context = ServiceContext.from\_defaults(embed\_model=embed\_model,llm=None)

index = VectorStoreIndex(

nodes, storage\_context=storage\_context, service\_context=service\_context

)

En esta celda lo que realizamos es añadir todos los Nodos con sus respectivos embeddings en la base de datos vectorial

retriever = index.as\_retriever(similarity\_top\_k=2)

Creamos el retriever que va ayudar a crear el modelo RAG (justamente esto lo que va a ser es encontrar la similaridad del coseno que más se aproxime a nuestra consulta transformada a embedding. Entonces el sistema con esto va a encontrar la respuesta y a su vez contextualizar la pregunta y la respuesta.

5- Ponemos en funcionamiento nuestro sistema RAG

print('Realizando llamada a HuggingFace para generar respuestas...\n')

while True:

queries = input("Ingrese su pregunta para el Chat especialista en Nicolas Maquiavelo: (si desea salir ingrese 'salir')")

if queries=="salir":

break

nodes = retriever.retrieve(queries)

final\_prompt = prepare\_prompt(queries, nodes)

print('Pregunta:', queries)

node\_metadata = node.metadata

file\_name = node\_metadata['file\_name']

# Ahora puedes imprimir el nombre del archivo

print(f"La respuesta se puede encontrar en el archivo: {file\_name}")

print('Respuesta:')

print(generate\_answer(final\_prompt))

print('-------------------------------------------------------')

Para finalizar realizamos un bucle while donde nuestro chatbot completamente configurado va a responder todas las preguntas que le hagamos como si fuese un gran asistente virtual especialista en Nicolas Maquiavelo!