TRABAJO PRÁCTICO FINAL PROCESAMIENTO DEL LENGUAJE NATURAL
Tecnicatura Universitaria en Inteligencia Artificial
FECIA - UNR

Leonel Palermo

Profesores:
D'Alessandro, Ariel.
Geary, Alan.
Leon Cavallo.
Andrea Carolina.
Manson, Juan Pablo.

Pautas generales:

El trabajo deberá ser realizado individualmente.
Deberá informar cuál es la url del repositorio con el que van a trabajar y las definiciones de en qué problematicas quisieran solucionar con un sistema multiagente en el siguiente formulario: https://docs.google.com/forms/d/e/1FAIpQLSdOZNGOzQ1gbf43caA4ygAbLx5tm5bU-s8RdfdftOzd_aXzhA/viewform?usp=pp_url
Se debe entregar un informe en el cual se incluya las justificaciones y un vínculo a los archivos que permitan reproducir el proyecto. No se acepta solamente código.

Temas deseables a cubrir en el tp

Recuperación de datos de bases de datos de grafos
Extracción de conocimiento de texto y posterior inserción en una base de datos de grafos
Agentes (estará cubierto en el Ejercicio 2)

Ejercicio 1 - RAG

Crear un chatbot experto en un tema a elección, usando la técnica RAG (Retrieval Augmented Generation). Como fuentes de conocimiento se utilizarán al menos las siguientes fuentes:

- Documentos de texto
- Datos numéricos en formato tabular (por ej., Dataframes, CSV, sqlite, etc.)
- Base de datos de grafos (Online o local)

El sistema debe poder llevar a cabo una conversación en lenguaje español. El usuario podrá hacer preguntas, que el chatbot intentará responder a partir de datos de algunas de sus fuentes. El asistente debe poder clasificar las preguntas, para saber qué fuentes de datos utilizar como contexto para generar una respuesta.

Requerimientos generales

- Realizar todo el proyecto en un entorno Google Colab
- El conjunto de datos debe tener al menos 100 páginas de texto y un mínimo de 3 documentos.
- Realizar split de textos usando Langchain (RecursiveTextSearch, u otros métodos disponibles). Limpiar el texto según sea conveniente.
- Realizar los embeddings que permitan vectorizar el texto y almacenarlo en una base de datos ChromaDB
- Los modelos de embeddings y LLM para generación de texto son a elección
- Para el desarrollo del “Clasificador” es posible utilizar diversas técnicas aprendidas en la materia, por ejemplo en Unidad 3 y Unidad 6

Ejercicio 2 - Agentes

Realice una investigación respecto al estado del arte de las aplicaciones actuales de agentes inteligentes usando modelos LLM libres.

Plantee una problemática a solucionar con un sistema multiagente. Defina cada uno de los agentes involucrados en la tarea. Es importante destacar con ejemplos de conversación, la interacción entre los agentes.

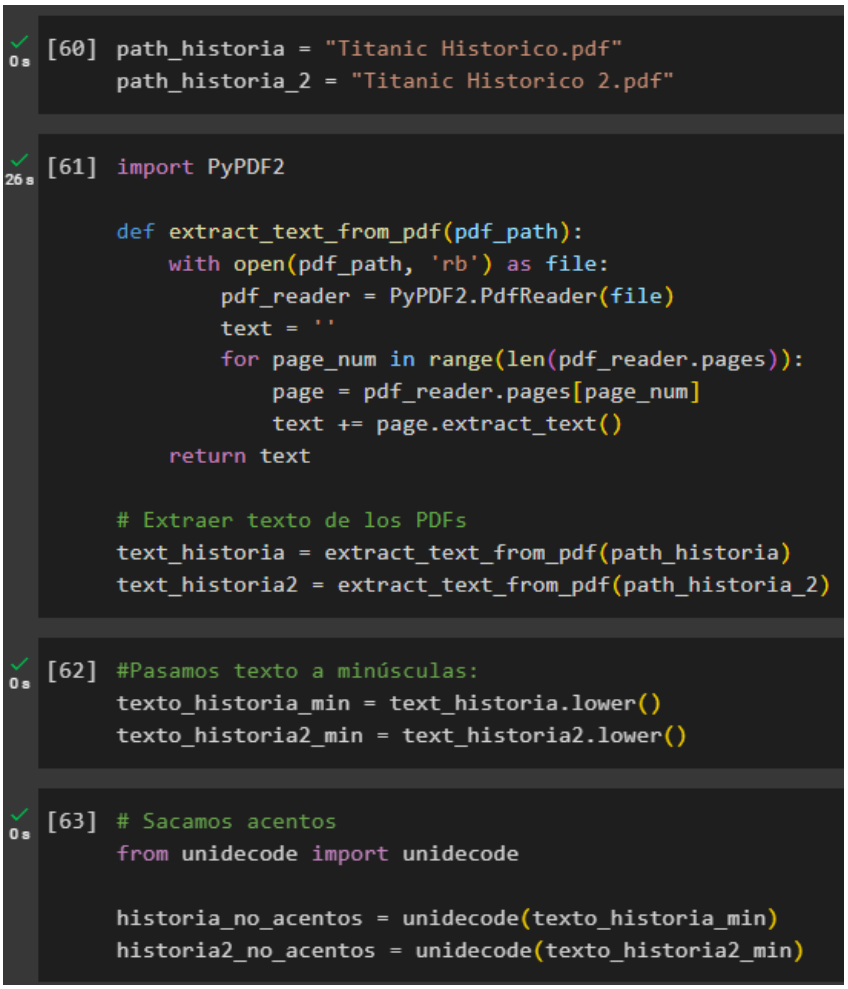
Realice un informe con los resultados de la investigación y con el esquema del sistema multiagente, no olvide incluir fuentes de información.

Opcional: Resolución con código de dicho escenario.

Paso a paso (puntos importantes)

Para la resolución del primer ejercicio seleccioné como tema, el RMS Titanic. Tanto la historia real que obtendré de mis archivos pdf, como datos de la película para los cuáles usaré la base de datos de grafos Wikidata. Además algunos datos sobre el precio de los pasajes y demás de un csv obtenido de kaggle.

1.1 Luego de instalar las librerías necesarias realicé el procesamiento de los textos, eliminación de stopwords, pasar a minúscula, eliminación de caracteres especiales, etc.



```
[60] path_historia = "Titanic Historico.pdf"
     path_historia_2 = "Titanic Historico 2.pdf"
```

```
[61] import PyPDF2

     def extract_text_from_pdf(pdf_path):
         with open(pdf_path, 'rb') as file:
             pdf_reader = PyPDF2.PdfReader(file)
             text = ''
             for page_num in range(len(pdf_reader.pages)):
                 page = pdf_reader.pages[page_num]
                 text += page.extract_text()
         return text

     # Extraer texto de los PDFs
     text_historia = extract_text_from_pdf(path_historia)
     text_historia2 = extract_text_from_pdf(path_historia_2)
```

```
[62] #Pasamos texto a minúsculas:
     texto_historia_min = text_historia.lower()
     texto_historia2_min = text_historia2.lower()
```

```
[63] # Sacamos acentos
     from unidecode import unidecode

     historia_no_acentos = unidecode(texto_historia_min)
     historia2_no_acentos = unidecode(texto_historia2_min)
```

1.2 Una vez resuelto eso obtenemos los fragmentos de texto. Usé word_tokenizer para transformar cada palabra en un token y luego uní esos tokens en grupos de 500, por último uní los elementos para que formaran una lista.

```
[65] import nltk
     from nltk.tokenize import word_tokenize

     # Descargar el tokenizador de palabras de NLTK si no está descargado
     nltk.download('punkt')

     # Función para dividir una lista de tokens en fragmentos cada 500 palabras
     def split_tokens_by_word_count(tokens, word_count=500):
         fragments = []
         current_fragment_words = []

         for word in tokens:
             current_fragment_words.append(word)

             if len(current_fragment_words) >= word_count:
                 fragments.append(current_fragment_words)
                 current_fragment_words = []

         if current_fragment_words:
             fragments.append(current_fragment_words)

         return fragments

     # Tokenizar los textos
     tokens_historia = word_tokenize(sin_especiales_historia)
     tokens_historia2 = word_tokenize(sin_especiales_historia2)

     # Dividir los tokens en fragmentos cada 500 palabras
     fragmentos_historia = split_tokens_by_word_count(tokens_historia, word_count=500)
     fragmentos_historia2 = split_tokens_by_word_count(tokens_historia2, word_count=500)

     # Imprimir los fragmentos resultantes
     print("Fragmentos de la novela:")
     for i, fragmento in enumerate(fragmentos_historia):
         print(f"Fragmento {i + 1}: {fragmento}")

     print("\nFragmentos del guion:")
     for i, fragmento in enumerate(fragmentos_historia2):
         print(f"Fragmento {i + 1}: {fragmento}")
```

```python
# Convertir cada fragmento de tokens en una cadena continua
fragmentos_historia = [' '.join(fragmento) for fragmento in fragmentos_historia]
fragmentos_historia2 = [' '.join(fragmento) for fragmento in fragmentos_historia2]

# Imprimir los fragmentos resultantes
print("Fragmentos de la historia:")
for i, fragmento in enumerate(fragmentos_historia):
    print(f"Fragmento {i + 1}: {fragmento}")

print("\nFragmentos del historia2:")
for i, fragmento in enumerate(fragmentos_historia2):
    print(f"Fragmento {i + 1}: {fragmento}")
```

Obteniendo algo como lo que se ve a continuación:

```
Fragmentos de la historia:
Fragmento 1: 1 titanic historias para despues de un naufragio fernando jose garcia
Fragmento 2: imposible unir los testimonios de los supervivientes para delimitar lo
Fragmento 3: muerte por hipotermia de 1517 personas de esos 1517 muertos 58 eran ni
Fragmento 4: largo el telegrafo no habia dejado de recibir en todo el dia avisos de
Fragmento 5: de que la sonda hubiera aumentado es que habian colisionado con algun
Fragmento 6: algunos trasatlanticos la tripulacion solia apagar las luces de los sa
Fragmento 7: de la navegacion podia ser requerido en cualquier momento por los ofic
Fragmento 8: hubiese aumentado parecia que una gran parte de el estaba fuera del ag
Fragmento 9: silueta del buque que se divisaba en el horizonte y se volvio hacia su
```

1.3 Utilizamos BERT para obtener los embeddings de dichos fragmentos.

```python
[74] from transformers import BertModel, BertTokenizer
import torch
import numpy as np

# Cargar el modelo BERT en español
model = BertModel.from_pretrained('dccuchile/bert-base-spanish-wwm-cased')
tokenizer = BertTokenizer.from_pretrained('dccuchile/bert-base-spanish-wwm-cased')

# Función que obtiene embeddings para cada texto
def obtener_embeddings(fragmentos):
    embeddings = []
    for fragmento in fragmentos:
        tokens = tokenizer(fragmento, truncation=True, padding=True, return_tensors='pt', max_length=512)
        outputs = model(**tokens)
        embedding_vector = outputs.last_hidden_state.mean(dim=1).squeeze()
        embeddings.append(embedding_vector.tolist())
    return embeddings

# Obtener embeddings para cada fragmento
embedding_historia = obtener_embeddings(fragmentos_historia)
embedding_historia2 = obtener_embeddings(fragmentos_historia2)

# Imprimir los resultados
print("Embeddings de los fragmentos de historia:", np.array(embedding_historia))
print("Embeddings de los fragmentos de historia2:", np.array(embedding_historia2))
```

1.4 Guardamos los embeddings en chromaDB

```
[79] import chromadb
     chroma_client = chromadb.Client()
```

```
[81] collection = chroma_client.get_collection(name="datos_titanic")
     if collection is not None:
         print("Collection already exists.")
     else:
         # Create the collection if it doesn't exist
         collection = chroma_client.create_collection(name="datos_titanic")
```

```
[89] #Agregamos los embeddings a chromaDB
     collection.add(
         embeddings=embeddings_titanic,
         documents=fragmentos_titanic,
         ids=ids_titanic
     )
```

2.1 Procesamos el csv

```
[90] import pandas as pd
```

```
[91] path_csv = 'Titanic.csv'
```

```
[92] # Cargar el archivo CSV en un DataFrame de pandas
     df_titanic = pd.read_csv(path_csv)
```

```
df_titanic.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	892	0	3	Kelly, Mr. James	male	34.5	0	0	330911	7.8292	NaN	Q
1	893	1	3	Wilkes, Mrs. James (Ellen Needs)	female	47.0	1	0	363272	7.0000	NaN	S
2	894	0	2	Myles, Mr. Thomas Francis	male	62.0	0	0	240276	9.6875	NaN	Q
3	895	0	3	Wirz, Mr. Albert	male	27.0	0	0	315154	8.6625	NaN	S
4	896	1	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.0	1	1	3101298	12.2875	NaN	S

2.2 lo convertimos a un string para luego poder usarlo en las respuestas

```python
def dataframe_to_string(df):
    result = ""
    for index, row in df.iterrows():
        # Reemplazamos los valores de "Survived" por "no" si es 0, o por "si" si es 1
        survived = "no" if row['Survived'] == 0 else "si"

        result += f"PassengerId: {row['PassengerId']}\n"
        result += f"Sobrevivio: {survived}\n"
        result += f"Clase: {row['Pclass']}\n"
        result += f"Nombre: {row['Name']}\n"
        result += f"Sexo: {row['Sex']}\n"
        result += f"Edad: {row['Age']}\n"
        result += f"Parch: {row['Parch']}\n"
        result += f"Boleto: {row['Ticket']}\n"
        result += f"Precio: {row['Fare']}\n"
        result += f"Cabina: {row['Cabin']}\n"
        result += f"Embarco: {row['Embarked']}\n"
        result += "\n"

    return result

# Convertimos el DataFrame en un string
result_string = dataframe_to_string(df_titanic)

# Imprimimos el resultado
print(result_string)
```

3.1 Nos conectamos a la base de datos de grafos, en este caso Wikidata

```python
def obtener_info_personaje(label_sujeto):
    # Iniciar sesión en Wikidata
    login_instance = wdi_login.WDLogin(wikidata_user, wikidata_password)

    # Probar que la conexión funciona con una consulta SPARQL
    sparql_query = f'''
    SELECT ?sujeto ?sujetoLabel
    WHERE {{
      ?sujeto rdfs:label "{label_sujeto}"@en.
      SERVICE wikibase:label {{ bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }}
    }}
    '''

    # Ejecutar la consulta SPARQL
    result = wdi_core.WDItemEngine.execute_sparql_query(sparql_query, as_dataframe=True)

    # Extracción del ID
    try:
        valor_sujeto = result['sujeto'].str.split('/').str[-1]
    except (KeyError, AttributeError, IndexError):
        # Manejar el error y retornar un string vacío en caso de cualquier excepción esperada
        return ''

    # Imprimir el resultado
    sujeto_id = valor_sujeto.values[0]

    description = f'{label_sujeto}  :\n'

    propiedades = ["P21", "P27", "P1559", "P1477", "P735", "P569", "P19", "P570", "P1196", "P509",
                   "P157", "P22", "P25", "P3373", "P3342", "P106", "P108", "P39", "P551", "P463"]
```

```python
for propiedad_id in propiedades:
    # Construir la consulta SPARQL para obtener los valores por sujeto y propiedad
    consulta_sparql = f"""
    SELECT ?property ?propertyLabel ?value ?valueLabel
    WHERE {{
      wd:{sujeto_id} wdt:{propiedad_id} ?value.
      ?property wikibase:directClaim wdt:{propiedad_id}.
      SERVICE wikibase:label {{ bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }}
    }}
    """

    # URL del punto de consulta SPARQL de Wikidata
    endpoint_url = "https://query.wikidata.org/sparql"

    # Encabezados HTTP necesarios para la consulta a Wikidata
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:100.0) Gecko/20100101 Firefox/100.0',
        'Accept': 'application/json'
    }

    # Parámetros de la consulta SPARQL
    params = {
        'query': consulta_sparql,
        'format': 'json'
    }

    # Realizar la solicitud HTTP GET
    respuesta = requests.get(endpoint_url, headers=headers, params=params)

    # Verificar si la solicitud fue exitosa (código de estado 200)
    if respuesta.status_code == 200:
        # Convertir la respuesta a formato JSON
        resultados = respuesta.json()
```

```
        # Obtener los valores asociados a la propiedad
        valores = [(item['propertyLabel']['value'], item['valueLabel']['value']) for item in resultados['results']['bindings']]

        for propiedad, valor in valores:
            description += f'{propiedad}: {valor} \n'
    else:
        description = ''

return description
```

4.1 Comenzamos copiando el código de la unidad 6 y agregándole un fewshots para que el LLM entienda qué tipo de pregunta le estamos haciendo

```
def zephyr_chat_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)
```

```python
# Aquí hacemos la llamada al modelo
def elegir_fuente(prompt: str, api_key, max_new_tokens: int = 768) -> None:
    messages: List[Dict[str, str]] = [
                                    {
                                      "role": "system",
                                      "content": "Sos un asistente bibliográfico, especializado en iden
                                    },
                                    {
                                      "role": "user",
                                      "content": "¿Quién fue Edward Smith?"
                                    },
                                    {
                                      "role": "assistant",
                                      "content": "Personaje: [Edward Smith]"
                                    },
                                    {
                                      "role": "user",
                                      "content": "¿Cuáles fueron las últimas palabras de Jack Dawson?"
                                    },
                                    {
                                      "role": "assistant",
                                      "content": "Personaje: [Jack Dawson]"
                                    },
                                    {
                                      "role": "user",
                                      "content": "¿Qué papel desempeñó Thomas Andrews?"
                                    },
                                    {
                                      "role": "assistant",
                                      "content": "Personaje: [Thomas Andrews]"
                                    },
                                    {
                                      "role": "user",
                                      "content": "¿Cuántas personas sobevivieron al choque?"
                                    },
                                    {
                                      "role": "assistant",
                                      "content": "Sobrevivientes"
```

```
                "role" : "user",
                "content": f'Responde de una de las siguientes maneras, sin usar información previ
                "Personaje: [Nombre de un personaje en específico mencionado en la pregunta]" Si e
                "Sobrevivientes" Si es una pregunta sobre las victimas o sobrevivientes del accide
                "Historia" Si es sobre la historia real del RMS Titanic.\n\
                ----------------------------------------\n\
                La pregunta o pedido es de la siguiente forma:\n\
                {prompt}'
            }
        ]


try:
    prompt_formatted: str = zephyr_chat_template(messages, add_generation_prompt=True)

    # 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_formatted,
        "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_formatted):]
    print(respuesta)
    return respuesta
```

4.2 Dado que las respuestas tienen información en un formato que no nos interesa, vamos a formatearlas para que sólo nos entreguen lo que necesitamos

```python
def formatear_respuesta(texto):
    palabras_clave = ["Personaje", "Sobrevivientes", "Historia"]

    for palabra in palabras_clave:
        if palabra in texto:
            if palabra == "Personaje":
                match = re.search(r'\[([^]]*)\]', texto)
                if match:
                    texto_entre_corchetes = match.group(1)
                    return [palabra, texto_entre_corchetes]
            else:
                return [palabra, None]

    return [None, None]

# Ejemplo de uso:
respuesta_limpia = formatear_respuesta(answer)
print(respuesta_limpia)
```

```
['Historia', None]
```

4.3 Elegimos el contexto adecuado para traer la respuesta

```python
def devolver_contexto(respuesta_LLM, prompt):
    # Si no se halló nada, no devolver contexto
    if respuesta_LLM in ([None, None], ["Personaje", None]):
        return ("", 'Error al identificar la fuente')

    # Si la pregunta o tarea es referida a un personaje
    elif respuesta_LLM[0] == "Personaje":
        propiedades_personaje = obtener_info_personaje(respuesta_LLM[1])
        return (propiedades_personaje, 'Base de datos de grafos')

    # Si la pregunta o tarea es referida a los sobrevivientes
    elif respuesta_LLM[0] == "Sobrevivientes":
        dataframe_string = dataframe_to_string(df_titanic)
        return (dataframe_string, 'Dataframe')

    # Si la pregunta o tarea es referida a la historia real
    elif respuesta_LLM[0] == "Historia":
        contexto = ''
        # Obtener el cliente Chroma
        chroma_client = chromadb.Client()
        # Obtener la colección
        collection = chroma_client.get_collection(name="datos_titanic")
        # Query
        query_embedding = obtener_embeddings(prompt)
        results = collection.query(query_embeddings= query_embedding, n_results=5)
        for result in results['documents'][0]:
          contexto += result

        return (contexto, "Base de datos vectorial de Embeddings")
```

4.4 Le decimos al modelo cómo debe responder, nuevamente basándonos en la unidad 6

```
[114] def pregunta_y_contexto(prompt: str, context: str, api_key, max_new_tokens: int = 768) -> None:
          messages: List[Dict[str, str]] = [
```

```
{
    "role": "system",
    "content": "Eres un asistente útil que siempre responde con respuestas veraces, útiles y basadas en hechos."
},
{
    "role": "user",
    "content": f"La información de contexto es la siguiente:\n\
    ---------------------\n\
    {context}\n\
    ---------------------\n\
    Dada la información de contexto anterior, y sin utilizar conocimiento previo, responde la siguiente pregunta.\n\
    Pregunta: {prompt}\n\
    Respuesta: "
}
```

```
        try:
            prompt_formatted: str = zephyr_chat_template(messages, add_generation_prompt=True)

            # 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_formatted,
                "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_formatted):]
            return respuesta

        except Exception as e:
            print(f"An error occurred: {e}")
```

5.1 Necesitamos ahora una función que llame a todas las anteriores y las combine

```python
def chatbot(prompt, api_key):
    # Identificar la fuente
    fuente = elegir_fuente(prompt , api_key=api_key)

    # Formatear respuesta del LLM para la fuente
    fuente_limpia = formatear_respuesta(fuente)

    # Obtener el contexto de la fuente indicada y el ID de la fuente
    contexto, id_source = devolver_contexto(fuente_limpia, prompt)

    # Obtener la respuesta con el prompt original más el contexto obtenido
    respuesta_final = pregunta_y_contexto(prompt, contexto, api_key=api_key)

    return (respuesta_final, id_source, contexto)
```

5.2 Terminamos con un ejemplo para asegurarnos de que el programa funcione:

```
[117] # Prompt:
      prompt = '¿Quien es Jack Dawson?'

      # Llamar a la función del chatbot
      respuesta, fuente, contexto = chatbot(prompt, api_key)

      # Imprimimos el resultado
      print('-----------------------------------------------------------------------')
      print(f'PREGUNTA:\n{prompt}')
      print('-----------------------------------------------------------------------')
      print(f'FUENTE ESCOGIDA: {fuente}')
      print('-----------------------------------------------------------------------')
      print(f'CONTEXTO BRINDADO:\n{contexto}')
      print('-----------------------------------------------------------------------')
      print(f'RESPUESTA:\n{respuesta}')
```

```
Personaje: [Jack Dawson] en la película Titanic (1997), interpretado por el actor L

En cuanto a la última pregunta, el personaje de Jack Dawson es un elemento ficticio
-----------------------------------------------------------------------
PREGUNTA:
¿Quien es Jack Dawson?
-----------------------------------------------------------------------
FUENTE ESCOGIDA: Base de datos de grafos
-----------------------------------------------------------------------
CONTEXTO BRINDADO:
Jack Dawson  :
sex or gender: male
country of citizenship: United States of America
given name: Jack
date of birth: 1889-01-01T00:00:00Z
place of birth: Chippewa Falls
date of death: 1912-04-15T00:00:00Z
manner of death: accidental death
cause of death: hypothermia
occupation: adventurer
occupation: drawer

-----------------------------------------------------------------------
RESPUESTA:
Jack Dawson fue un hombre estadounidense nacido el 1 de enero de 1889 en Chippewa F
```

EJERCICIO 2 - MULTIAGENTES

Vamos por partes, primero deberíamos entender qué es un agente inteligente:

Los agentes inteligentes son entidades autónomas o sistemas informáticos capaces de interactuar con su entorno para lograr objetivos específicos. Estos agentes pueden ser diseñados para procesar información, tomar decisiones y realizar acciones de manera autónoma, adaptándose dinámicamente a cambios en su entorno. Utilizan técnicas de inteligencia artificial, como el procesamiento del lenguaje natural (NLP) y el aprendizaje automático (ML), para comprender y responder a entradas complejas, colaborar con otros agentes y resolver tareas complejas que serían difíciles de lograr individualmente.

Ahora sabiendo lo que es un agente inteligente, pasamos a definir que es un sistema multiagente:

Un sistema multiagente es un conjunto de agentes inteligentes autónomos que interactúan entre sí para lograr objetivos complejos que serían difíciles de alcanzar individualmente. Estos sistemas están compuestos por múltiples agentes que pueden ser diseñados para procesar información, tomar decisiones y realizar acciones de manera independiente, colaborando entre sí para resolver problemas o realizar tareas específicas.

Desde la perspectiva del Procesamiento del Lenguaje Natural (NLP), los sistemas multiagente ofrecen oportunidades únicas para el análisis y la generación de lenguaje. Cada agente puede especializarse en distintos aspectos del lenguaje, como la comprensión de la semántica, la sintaxis o el contexto cultural y emocional, lo que permite una comprensión y generación de lenguaje más rica y matizada. Además, en un entorno multiagente, cada agente puede representar a un participante diferente en una conversación, facilitando la simulación de interacciones lingüísticas más dinámicas y naturales.

Ahora pasamos a buscar información sobre agentes inteligentes que usen LLM, o LLM agents:

Link del video que explica lo siguiente: https://www.youtube.com/shorts/ASP8DOs-qxc
Un agente LLM es una entidad de software capaz de razonar y ejecutar tareas de manera autónoma.
Hay una línea difusa entre lo que es y no es un agente LLM. La diferencia clave es la flexibilidad. Un LLM que resume un conjunto de documentos homogéneos utilizando el mismo patrón no es un agente.
Un LLM que selecciona la mejor fuente de datos para responder una pregunta es un agente.
Link: https://www.youtube.com/shorts/t0Ki-o7fgtk

Un Agente LLM es una entidad de software capaz de razonar y ejecutar tareas de forma autónoma.
Algún día, los LLMs pueden ser lo suficientemente inteligentes como para resolver mágicamente problemas complejos al examinar nuestros datos y brindarnos las respuestas que queremos.
Desafortunadamente, ese día no es hoy.
Sin orientación, los Agentes LLM realizarán actividades al azar, y corregirlos con otros Agentes hará que sus aplicaciones sean costosas y lentas.

Encontré más información general sobre LLM agents en este video:
https://www.youtube.com/watch?v=XV4IBaZqbps&ab_channel=Prolego

Ahora enfocándonos más en la consigna específica del tp:

Al leer la consigna se me presentó un problema, quisiera aclarar algo, y espero que no me resten puntos por corregirlos (además puedo estar equivocado), pero nos piden que investiguemos sobre el estado del arte, esto es si no me equivoco una traducción literal de “state of the art”, cuya traducción correcta sería “tecnología de punta” ya que “estado del arte” no significa nada en español, más que su significado literal que sería ver en qué estado se encuentra… el arte, no tendría mucho sentido en el contexto de este tp. Ahora bien, si entendí correctamente y lo que desean es que investiguemos sobre tecnología de punta que emplee LLM agents, esto fue lo que encontré:

CodeLlama:
CodeLlama es un agente de LLM que ha sido entrenado para escribir y generar código en una variedad de lenguajes de programación. Algunos de los lenguajes incluyen Python, Java y C++. Por supuesto, no es un reemplazo para los codificadores, lo que CodeLlama puede hacer es ser utilizado para generar código para una variedad de tareas, como la creación de aplicaciones web, el desarrollo de aplicaciones móviles y la escritura de scripts. Liberando tiempo valioso para que los desarrolladores se concentren en proyectos y planificación más complejos.

También se puede utilizar para generar código para propósitos específicos, como generar código para implementar un algoritmo específico o para resolver un problema específico. CodeLlama es una herramienta poderosa que puede ser utilizada tanto por programadores experimentados como por novatos.

Además encontré lo siguiente que si bien no es específicamente un agente, es un proyecto que podría hacer más fácil el desarrollo de agentes:

Open Interpreter:
Open Interpreter es un proyecto que tiene como objetivo crear un intérprete universal para modelos de lenguaje amplios. Esto permitiría que los LLMs se comuniquen entre sí y accedan a información de una variedad de fuentes, lo que les permitiría compartir información y colaborar en tareas con mayor eficiencia.

El proyecto aún está en sus etapas iniciales, pero tiene el potencial de revolucionar la forma en que se utilizan los LLMs de código abierto. Si tiene éxito, podría llevar a que los LLMs se utilicen en una amplia gama de nuevas aplicaciones, desde el servicio al cliente hasta el diagnóstico médico.

Este es el link de ambos títulos:
https://odsc.medium.com/9-open-source-llms-and-agents-to-watch-728049d77060

Consigna siguiente:
Plantee una problemática a solucionar con un sistema multiagente. Defina cada uno de los agentes involucrados en la tarea. Es importante destacar con ejemplos de conversación, la interacción entre los agentes.

Problemática: Coordinación de entrega de paquetes en una empresa de logística.

Descripción del problema: Una empresa de logística debe coordinar la entrega de paquetes desde un almacén central a diferentes destinos. Los paquetes varían en tamaño, peso y urgencia de entrega. La empresa desea optimizar la distribución de estos paquetes entre varios vehículos de entrega para minimizar los costos operativos y maximizar la eficiencia de la entrega.

Agentes involucrados:

Agente de Almacén Central: Este agente es responsable de recibir y organizar los paquetes para su entrega. Tiene acceso a la información sobre los paquetes disponibles, incluido su tamaño, peso y urgencia de entrega.

Agente de Vehículo de Entrega: Hay varios agentes de vehículos de entrega, cada uno asociado con un vehículo de entrega específico. Estos agentes están encargados de recoger los paquetes asignados en el almacén central y entregarlos a los destinos correspondientes. Tienen información sobre la capacidad de carga y la ubicación actual del vehículo.

Agente de Control de Rutas: Este agente se encarga de planificar las rutas óptimas para los vehículos de entrega. Analiza la ubicación actual de los vehículos, la carga disponible y la ubicación de los destinos para determinar la mejor ruta para cada vehículo.

Ejemplo de interacción entre los agentes:

Agente de Almacén Central: Informa que poseen 5 paquetes que deben ser entregados y solicita un informe del estado de los vehículos que hacen las entregas.

Agente de Control de Rutas: "El vehículo A está cerca del almacén y tiene espacio para dos paquetes medianos. El vehículo B está más lejos pero tiene capacidad para tres paquetes pequeños. Sugiero asignar un paquete grande al vehículo A y los otros paquetes al vehículo B para optimizar la entrega."

Agente de Vehículo de Entrega (Vehículo A): "Confirmo que estoy listo para recibir un paquete grande. ¿Cuál es la dirección de entrega?"

Agente de Vehículo de Entrega (Vehículo B): "Confirmo que tengo espacio para dos paquetes pequeños. ¿Cuáles son los destinos de entrega para estos paquetes?"