NLP INFORME:

Carátula

Título: Desarrollo de un Chatbot Experto en un Juego de Mesa Estilo Eurogame usando RAG

Nombre: Brisa Moresco

Curso: TUIA

Fecha: 18/12/2024

Introducción

En el presente informe se presenta el desarrollo de un chatbot experto basado en la técnica de **Retrieval Augmented Generation** (RAG), cuyo objetivo es asistir a los usuarios en un juego de mesa estilo **Eurogame**. El chatbot se diseñará para ser capaz de interactuar en español e inglés, proporcionando respuestas relevantes basadas en un conjunto de fuentes de datos, incluyendo documentos de texto, datos numéricos en formato tabular y una base de datos de grafos. Para lograr esto, el sistema implementará un clasificador capaz de identificar el tipo de pregunta del usuario y determinar qué fuente de datos utilizar como contexto para generar la respuesta adecuada.

El sistema se desarrollará en un entorno **Google Colab** y se garantizará que todos los recursos necesarios, como documentos, datasets y bases de datos, sean obtenidos de manera automática sin la intervención manual del usuario. Además, se llevará a cabo la segmentación de textos, la vectorización de los mismos mediante embeddings, y la consulta dinámica de datos, para garantizar un rendimiento eficiente y escalable. El chatbot también se someterá a una comparación entre dos versiones del clasificador: una basada en **LLM** y otra entrenada con ejemplos y embeddings.

Ejercicio 1: Resumen

El **Ejercicio 1** tiene como objetivo la creación de un chatbot experto que interactúe con los usuarios sobre un juego de mesa tipo **Eurogame**. El chatbot utilizará la técnica **RAG** para generar respuestas relevantes a partir de diferentes fuentes de conocimiento, como documentos de texto, bases de datos tabulares y bases de datos de grafos. Además, el sistema estará

diseñado para clasificar las preguntas de los usuarios y seleccionar la fuente de datos adecuada.

Desarrollo detallado de los pasos para la solución del problema:

1. Configuración inicial y autenticación en Hugging Face

El primer bloque de código se encarga de la configuración inicial, asegurando que el token de autenticación de Hugging Face esté correctamente configurado:

```
import os
os.environ['HF_TOKEN'] = 'hf_bBpEkjVygdzxbhKUFubbSBkcDHYpWW
maEl'
client_hf = os.environ.get('HF_TOKEN')
print(client_hf)
```

Justificación:

- Es importante almacenar el token de autenticación de manera segura y utilizarlo para autenticar el acceso a los servicios de Hugging Face. Este paso permite la autenticación y la utilización de sus modelos, como los utilizados en la generación de queries SPARQL más adelante.

2. Obtener texto de una página web

A continuación, se define una función `obtener_texto(url)` que obtiene el contenido de una página web usando `requests` y `BeautifulSoup` para analizar el HTML:

```
import requests
from bs4 import BeautifulSoup

def obtener_texto(url):
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')
    return soup
```

Justificación:

- se utiliza `requests` para obtener el contenido de la página web y `BeautifulSoup` para parsear el HTML. Esto es útil cuando se requiere extraer datos de una página en línea de manera automática.

3. Procesamiento del texto de la página web

El contenido relevante de la página web se extrae y se guarda en una lista de ítems:

```
texto_lista = """
Tablero Principal reversible (de cartón)
118 Cartas (68×45 mm.)
...
"""

items = texto_lista.strip().split("\n")
```

Justificación:

- El bloque de texto es preprocesado para extraer la lista de componentes, que luego se procesa línea por línea para obtener los elementos, números y descripciones.

4. Guardar los datos en un archivo CSV

A continuación, se crea un archivo CSV para almacenar los componentes extraídos:

```
filename = "viticulture_componentes.csv"
with open(filename, mode='w', encoding='utf-8', newline='')
as file:
    writer = csv.writer(file)
    writer.writerow(["ID", "Número", "Elemento", "Descripci
ón"])
    for i, item in enumerate(items, start=1):
        match = re.match(r'(\d+)\s*(.*)\s*\((.*)\)', item)
        if match:
            numero = match.group(1)
            elemento = match.group(2).strip()
            descripcion = match.group(3).strip()
            writer.writerow([i, numero, elemento, descripci
```

```
on])
        else:
            writer.writerow([i, "", item, ""])
```

Justificación:

- Esta parte convierte los datos procesados en un formato estructurado que puede ser fácilmente utilizado para crear embeddings más adelante. Utilizar un archivo CSV permite un manejo y procesamiento más sencillo de los datos. El uso de expresiones regulares ayuda a extraer el número, el nombre del componente y su descripción.

5. Generación de embeddings usando `SentenceTransformer`

Una vez que los datos están organizados, se generan embeddings de las descripciones de los componentes para permitir una búsqueda semántica eficiente:

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer("all-MiniLM-L6-v2")
embeddings = model.encode(df_componentes["Descripción"].fil
lna("").tolist())
```

Justificación:

- Los embeddings son representaciones vectoriales densas que permiten que el modelo entienda y compare semánticamente el texto. `SentenceTransformer` es una biblioteca eficaz para este tipo de tareas, y el modelo `all-MiniLM-L6-v2` es adecuado por su rendimiento y eficiencia.

6. Almacenamiento de embeddings en ChromaDB

Los embeddings generados se almacenan en una base de datos persistente de ChromaDB:

```
import chromadb
client = chromadb.PersistentClient(path=persist_directory)
collection = client.get_or_create_collection(name="componen
tes")
for idx, row in df_componentes.iterrows():
    collection.add(
        ids=[str(idx)],
```

```
        embeddings=[embeddings[idx]],
        metadatas=[{"Elemento": row["Elemento"], "Descripci
ón": row["Descripción"]}]
    )
```

Justificación:

- ChromaDB es una base de datos diseñada para almacenar y consultar embeddings. Usarla permite que el sistema realice búsquedas rápidas y eficientes basadas en similitudes semánticas. Esto es clave para la técnica de "Retrieval Augmented Generation", ya que permite la recuperación de información relevante.

7. Implementación de búsqueda híbrida con BM25

Luego se configura un modelo de búsqueda híbrida usando BM25, que combina la búsqueda por palabras clave y por semántica:

```
from rank_bm25 import BM25Okapi

tokenized_descriptions = [desc.split() for desc in df_compo
nentes["Descripción"].fillna("")]
bm25 = BM25Okapi(tokenized_descriptions)
```

Justificación:

- BM25 es un algoritmo basado en la frecuencia de términos, que es eficaz para la búsqueda de documentos relevantes en un conjunto de datos. Esta estrategia híbrida permite mejorar los resultados de búsqueda, combinando búsquedas por coincidencias exactas y por semántica.

8. Generación de consultas SPARQL dinámicas

La función `generar_query_sparql` se utiliza para generar consultas SPARQL dinámicas basadas en el input del usuario:

```
def generar_query_sparql(prompt):
    messages = [
        {
            "role": "system",
            "content": "Eres un generador de consultas en S
parQL ..."
```

```
        },
        {
            "role": "user",
            "content": prompt
        }
    ]
    response = client_hf.chat(
        messages=messages,
        model="Qwen/Qwen2.5-Coder-32B-Instruct",
        max_tokens=500
    )
    query = response["message"]["content"]
    return query
```

Justificación:

- Esta función permite generar consultas SPARQL que extraen información relevante de una base de datos de grafos. Utiliza un modelo de Hugging Face para generar las consultas dinámicamente basadas en el input del usuario, asegurando que se extraigan solo los datos relevantes.

9. Ejemplo de uso de la consulta SPARQL

Finalmente, el código muestra cómo usar la función para generar una consulta SPARQL:

```
prompt = "¿Cuántos trabajadores de madera hay en la base de datos?"
query_sparql = generar_query_sparql(prompt)
print("Query SPARQL generada:\n", query_sparql)
```

Justificación:

- Este es un ejemplo práctico de cómo se puede usar la consulta SPARQL generada para obtener información dinámica de una base de datos. El prompt del usuario se convierte en una consulta estructurada para que la base de datos recupere solo los datos relevantes.

---

1. Definición del Grafo y Namespaces

```
g = Graph()
EX = Namespace("http://example.org/terms/")
BGG = Namespace("http://boardgamegeek.com/")
MISUT = Namespace("https://misutmeeple.com/")

g.bind("ex", EX)
g.bind("bgg", BGG)
g.bind("misut", MISUT)
```

Explicación:

- Se crea un **grafo RDF** vacío (`g = Graph()`) que servirá para almacenar las tripletas que conforman la información extraída de las páginas web.
- Se definen **namespaces** (espacios de nombres) para organizar los términos dentro del grafo. Cada namespace tiene una URI única:
  - `EX`: Se usa para términos genéricos.
  - `BGG`: Para los datos relacionados con **BoardGameGeek**.
  - `MISUT`: Para los datos provenientes de **Misut Meeple**.
- Se registran estos namespaces en el grafo con `g.bind()`, lo que facilita la creación de consultas y evita la necesidad de usar URIs largas repetidamente.

2. URLs de las Fuentes

```
urls = [
    "https://misutmeeple.com/2015/03/resena-viticulture/",
    "https://boardgamegeek.com/boardgame/128621/viticultur
e"
]
```

Explicación:

- Se definen las **URLs** de las páginas de donde se extraerán los datos. La primera URL corresponde a **Misut Meeple** y la segunda a **BoardGameGeek**.
- Estas URLs serán utilizadas por las funciones `extract_bgg_data()` y `extract_misut_data()` para obtener la información específica de cada sitio.

3. Funciones de Extracción de Datos

a) Extraer datos de BoardGameGeek

```
def extract_bgg_data(url):
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')

    title = soup.find('meta', {'property': 'og:title'})['co
ntent']
    description = soup.find('meta', {'property': 'og:descri
ption'})['content']
    image = soup.find('meta', {'property': 'og:image'})['co
ntent']

    return title, description, image
```

Explicación:

- Esta función utiliza `requests` para hacer una solicitud HTTP a la URL de BoardGameGeek y obtener el contenido de la página.
- **BeautifulSoup** se utiliza para analizar el HTML de la página y extraer los metadatos (`og:title`, `og:description`, `og:image`) que contienen el título, la descripción y la imagen de la página.
- Se devuelve el **título**, **descripción** e **imagen**, los cuales luego se añadirán al grafo RDF.

b) Extraer datos de Misut Meeple

```
def extract_misut_data(url):
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')

    title = soup.title.text.strip()
    paragraphs = soup.find_all('p')
    content = " ".join(p.text.strip() for p in paragraphs[:
3])  # Primeros 3 párrafos como resumen
```

```
    return title, content
```

Explicación:

- Similar a la función anterior, esta función realiza una solicitud HTTP a la página de **Misut Meeple** y usa **BeautifulSoup** para analizar el HTML.
- Se extrae el **título** de la página y los primeros tres párrafos para crear un **resumen** del contenido.
- La información extraída será utilizada en el grafo.

4. Creación del Grafo y Adición de Datos

```
# Extraer datos de BoardGameGeek
bgg_title, bgg_description, bgg_image = extract_bgg_data(ur
ls[1])
bgg_uri = URIRef(BGG[quote(bgg_title.replace(" ", "_"))])
# Codificar URI

g.add((bgg_uri, EX.hasDescription, Literal(bgg_descriptio
n)))
g.add((bgg_uri, EX.hasImage, URIRef(bgg_image)))

# Extraer datos de Misut Meeple
misut_title, misut_content = extract_misut_data(urls[0])
misut_uri = URIRef(MISUT[quote(misut_title.replace(" ",
"_"))])  # Codificar URI

g.add((misut_uri, EX.hasSummary, Literal(misut_content)))

# Relacionar los datos entre ambas fuentes
g.add((bgg_uri, EX.relatedReview, misut_uri))
```

Explicación:

- Después de extraer los datos de las páginas, se **codifican las URIs** para los títulos de los artículos, usando `quote` para asegurar que no haya caracteres especiales en las URIs.
- Se añaden las **tripletas RDF** al grafo utilizando el método `add()`:

    - Para **BoardGameGeek**, se añade la descripción y la imagen.
    - Para **Misut Meeple**, se añade el resumen.
- Se crea una **relación** entre los dos recursos, asociando la reseña de Misut Meeple con la página de BoardGameGeek mediante el predicado `relatedReview`.

5. Serialización y Visualización del Grafo

```
# Serializar y mostrar el grafo en formato "turtle"
print(g.serialize(format="turtle"))
```

Explicación:

- El grafo RDF es **serializado** en formato **Turtle**, un formato comúnmente utilizado para representar grafos RDF de manera legible.
- Se imprime el grafo serializado, mostrando las tripletas en un formato estructurado, donde cada tripleta tiene un sujeto, un predicado y un objeto.

6. Verificación del Contenido del Grafo

```
print("Contenido del grafo (organizado):")
for s, p, o in g:
    print(f"🔹 Sujeto: {s}\n   🔸 Predicado: {p}\n   🔹 Obj
eto: {o}\n")
```

Explicación:

- Se recorre el grafo y se imprime el contenido de cada tripleta (sujeto, predicado, objeto) de forma estructurada.
- Esto permite verificar que el grafo contiene los datos correctos.

7. Verificación de los Namespaces

```
print("\nNamespaces registrados:")
for prefix, namespace in g.namespaces():
    print(f"{prefix}: {namespace}")
```

Explicación:

- Esta parte imprime los **namespaces** registrados en el grafo para asegurar que las URIs de los namespaces estén correctamente configuradas.

8. Generación de Consultas SPARQL Dinámicas

```
def generar_query_grafo(prompt):
    messages = [
        {
            "role": "system",
            "content": (
                "Eres un generador de consultas en SPARQL. 
Genera una consulta en base a los datos proporcionados en e
l grafo Turtle. "
                "El grafo contiene información sobre el jue
go Viticulture, incluyendo descripciones, imágenes y reseña
s. "
                "Puedes generar consultas para extraer info
rmación sobre las descripciones, imágenes, o reseñas relaci
onadas con Viticulture."
            )
        },
        {
            "role": "user",
            "content": prompt
        }
    ]
    response = client_hf.text_generation(
        prompt=str(messages),
        model="Qwen/Qwen2.5-Coder-32B-Instruct",
        max_new_tokens=500
    )
    return response
```

Explicación:

- Esta función **genera consultas SPARQL dinámicas** a partir de un prompt dado. Utiliza un modelo de **Hugging Face** para interpretar el prompt y generar una consulta SPARQL adecuada.

- El modelo se alimenta de un mensaje que describe el propósito del grafo (información sobre el juego Viticulture) y del contenido del **prompt** proporcionado por el usuario.

9. Ejemplo de Consulta SPARQL Generada

```
prompt = "Genera una consulta SPARQL que recupere la descri
pción del juego Viticulture."
query_grafo = generar_query_grafo(prompt)
print("Query SPARQL generada:\n", query_grafo)
```

Explicación:

- Se proporciona un **ejemplo de prompt** para generar una consulta SPARQL que recupere la descripción del juego "Viticulture".
- El resultado es una consulta SPARQL que extrae la información solicitada del grafo.

---

1. URLs de las Páginas a Analizar

```
url1 = "https://misutmeeple.com/2015/03/resena-viticultur
e/"
url2 = "https://stonemaiergames.com/games/viticulture/"
```

Explicación y Justificación:

- Se definen dos URLs, una para el sitio **Misut Meeple** y otra para **Stonemaier Games**, que contienen reseñas sobre el juego "Viticulture". Estos sitios servirán como fuentes de datos para el análisis.

2. Encabezado de Solicitud (User-Agent)

```
headers = {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x6
4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.0.0
Safari/537.36"
}
```

Explicación y Justificación:

- `User-Agent`: Este encabezado se incluye en la solicitud HTTP para simular que la solicitud proviene de un navegador web, evitando así bloqueos por parte de algunos sitios web que impiden solicitudes automáticas.

3. Creación de Sesión para Reutilización de Conexión

```
session = requests.Session()
session.headers.update(headers)
```

Explicación y Justificación:

- Se utiliza una **sesión de requests** para reutilizar la misma conexión TCP durante las solicitudes. Esto puede mejorar el rendimiento, especialmente si se realizan múltiples solicitudes a un mismo servidor.

4. Función para Obtener y Extraer el Texto de una Página

```
def obtener_texto(url):
    try:
        response = session.get(url, timeout=10)  # Establec
er un tiempo de espera de 10 segundos
        response.raise_for_status()  # Verifica si hubo un
error en la solicitud
        soup = BeautifulSoup(response.content, 'html.parse
r')
        texto = soup.get_text(separator=' ', strip=True)
        return texto
    except requests.exceptions.RequestException as e:
        print(f"Error al descargar {url}: {e}")
        return None
```

Explicación y Justificación:

- La función `obtener_texto` realiza una solicitud HTTP a la URL dada usando la sesión creada anteriormente.
- `timeout=10` asegura que la solicitud se cancele si no se recibe respuesta en 10 segundos.
- Si la solicitud es exitosa, el contenido HTML de la página se pasa a `BeautifulSoup` para extraer el texto.

- `get_text(separator=' ', strip=True)`: Este método extrae todo el texto de la página, eliminando etiquetas HTML y dejando solo el contenido textual, separado por espacios y limpiado de los espacios innecesarios.
- Si ocurre un error en la solicitud, se maneja con `try-except` y se devuelve `None`.

5. Descargar y Guardar los Archivos de Texto

```
texto_doc1 = obtener_texto(url1)
texto_doc2 = obtener_texto(url2)

if texto_doc1:
    with open("documento1.txt", "w", encoding="utf-8") as
f:
        f.write(texto_doc1)

if texto_doc2:
    with open("documento2.txt", "w", encoding="utf-8") as
f:
        f.write(texto_doc2)
```

Explicación y Justificación:

- Se llama a la función `obtener_texto` para extraer el contenido de las dos URLs y se guarda en variables (`texto_doc1` y `texto_doc2`).
- Si el texto se obtiene correctamente (es decir, no es `None`), el contenido se guarda en archivos de texto (`documento1.txt` y `documento2.txt`), utilizando codificación UTF-8 para manejar caracteres especiales correctamente.

6. Leer y Mostrar los Primeros 500 Caracteres de los Archivos

```
if texto_doc1:
    with open("documento1.txt", "r", encoding="utf-8") as f
ile1:
        texto_doc1_leido = file1.read()

if texto_doc2:
    with open("documento2.txt", "r", encoding="utf-8") as f
```

```
ile2:
        texto_doc2_leido = file2.read()

if texto_doc1:
    print("Texto del documento 1 (primeros 500 caractere
s):")
    print(texto_doc1_leido[:500])

if texto_doc2:
    print("\nTexto del documento 2 (primeros 500 caractere
s):")
    print(texto_doc2_leido[:500])
```

Explicación y Justificación:

- Una vez guardados los archivos, el código los lee y muestra los primeros 500 caracteres de cada archivo para hacer una **verificación preliminar** del contenido extraído.
- Esto permite asegurarse de que el texto que se descargó es el correcto y contiene los elementos esperados.

7. División del Texto en Fragmentos (Chunks) Usando LangChain

```
%pip install langchain
from langchain.text_splitter import RecursiveCharacterTextS
plitter
```

Explicación y Justificación:

- **LangChain** es una biblioteca que facilita el procesamiento y análisis de texto, y es particularmente útil cuando se trabaja con grandes volúmenes de texto que necesitan ser divididos o transformados.
- `RecursiveCharacterTextSplitter` es una clase de LangChain que divide un texto en fragmentos (chunks) más pequeños de manera recursiva. Esta segmentación es útil cuando se necesita procesar o analizar textos largos que no caben en la memoria o que deben ser analizados en fragmentos.

8. Segmentación Recursiva del Texto

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=8
0, chunk_overlap=10)
texts2 = text_splitter.split_text(texto_doc2_leido)
for txt in texts2:
    print(f'{len(txt)}: {txt}')

text_splitter = RecursiveCharacterTextSplitter(chunk_size=8
0, chunk_overlap=10)
texts1 = text_splitter.split_text(texto_doc1_leido)
for txt in texts1:
    print(f'{len(txt)}: {txt}')
```

Explicación y Justificación:

- `RecursiveCharacterTextSplitter(chunk_size=80, chunk_overlap=10)`: Este configurador divide el texto en fragmentos de tamaño 80 caracteres, pero con una superposición de 10 caracteres entre fragmentos consecutivos. La superposición puede ser útil para mantener el contexto entre fragmentos.
- `split_text()` divide el texto de cada documento en fragmentos (chunks) y los imprime con su longitud y contenido, lo que permite verificar la segmentación del texto.

Configuración de Modelos y ChromaDB

En esta sección, se importan las bibliotecas necesarias, se configuran los modelos de embeddings y se establece la conexión con ChromaDB.

```
# Importar las librerías necesarias
import shutil
import os
import chromadb
from sentence_transformers import SentenceTransformer
import numpy as np
import time

# Configuración de modelos de incrustación (embeddings)
model_gte = SentenceTransformer('Alibaba-NLP/gte-multilingu
al-base', trust_remote_code=True)
model_e5 = SentenceTransformer('intfloat/multilingual-e5-sm
```

```
all')

# Configuración de ChromaDB para almacenar los embeddings
persist_directory = "./chroma_db"
client = chromadb.PersistentClient(path=persist_directory)

# Crear o recuperar colecciones donde se almacenarán los em
beddings
gte_collection = client.get_or_create_collection("gte_embed
dings")
e5_collection = client.get_or_create_collection("e5_embeddi
ngs")
```

Explicación:

- **Importación de bibliotecas**: Se importan `chromadb` para interactuar con la base de datos de embeddings, `SentenceTransformer` para generar los embeddings, y otras librerías como `numpy` y `time`.
- **Modelos de embeddings**: Usamos dos modelos preentrenados de **SentenceTransformer**: uno llamado `gte-multilingual-base` y otro llamado `multilingual-e5-small`. Estos modelos se utilizan para generar representaciones vectoriales (embeddings) de los textos en varios idiomas.
- **ChromaDB**: Es una base de datos optimizada para almacenar y consultar embeddings. configuramos un cliente de ChromaDB y creamos dos colecciones: una para cada tipo de modelo (`gte_embeddings` y `e5_embeddings`).

Función para Generar Embeddings

En esta parte se define una función que genera los embeddings de un conjunto de textos utilizando un modelo dado.

```
# Función para calcular embeddings a partir de un modelo
def get_embeddings(model, texts):
    return model.encode(texts).tolist()  # Convertir a list
a de listas
```

Explicación:

- **Función** `get_embeddings`: Esta función toma un modelo y una lista de textos. Luego utiliza el modelo para generar los embeddings de cada texto, y convierte la salida en una lista de listas (lo que es necesario para almacenarlo en ChromaDB).

---

1. Importación de bibliotecas y configuración inicial

```
import shutil
import os
import chromadb
from sentence_transformers import SentenceTransformer
import numpy as np
import time
```

- **shutil, os**: Estas bibliotecas se usan para manipular archivos y directorios.
- **chromadb**: Se utiliza para interactuar con la base de datos ChromaDB, donde se almacenan los embeddings.
- **sentence_transformers**: Proporciona herramientas para trabajar con modelos de incrustación de oraciones (embeddings) como `SentenceTransformer`.
- **numpy**: Para operaciones numéricas, especialmente en el manejo de matrices.
- **time**: Para medir el tiempo de ejecución en algunas partes del código.

2. Cargar modelos de incrustación de oraciones

```
model_gte = SentenceTransformer('Alibaba-NLP/gte-multilingu
al-base', trust_remote_code=True)
model_e5 = SentenceTransformer('intfloat/multilingual-e5-sm
all')
```

Se cargan dos modelos preentrenados (`GTE` y `E5`) para generar los embeddings de los textos. Ambos modelos están diseñados para trabajar con varios idiomas.

3. Configuración de ChromaDB

```
persist_directory = "./chroma_db"
client = chromadb.PersistentClient(path=persist_directory)

gte_collection = client.get_or_create_collection("gte_embed
dings")
e5_collection = client.get_or_create_collection("e5_embeddi
ngs")
```

- Se establece un directorio persistente `./chroma_db` donde ChromaDB almacenará los datos.
- Se crea un cliente de ChromaDB para almacenar los embeddings.
- Se crean dos colecciones en ChromaDB: una para los embeddings generados por el modelo GTE y otra para los de E5.

4. Función para calcular embeddings

```
def get_embeddings(model, texts):
    return model.encode(texts).tolist()
```

Esta función toma un modelo y una lista de textos, y genera los embeddings correspondientes utilizando el modelo especificado. Los embeddings se devuelven como una lista de listas.

5. Función para agregar embeddings a ChromaDB

```
def add_embeddings_to_chroma(collection, embeddings, senten
ces, model_name, source_label):
    for idx, embedding in enumerate(embeddings):
        doc_id = f"{source_label}_{model_name}_{idx}"
        collection.add(
            ids=[doc_id],
            documents=[sentences[idx]],
            embeddings=[embedding],
            metadatas=[{'source': source_label, 'model': mo
del_name}]
     )
```

Esta función toma los embeddings generados, los textos originales y los agrega a la base de datos ChromaDB. También incluye metadatos sobre el modelo y la fuente de los datos (como etiquetas).

6. Generación y almacenamiento de embeddings

```
# Procesar texts1
sentences1 = texts1
embeddings_gte_1 = get_embeddings(model_gte, sentences1)
embeddings_e5_1 = get_embeddings(model_e5, sentences1)

add_embeddings_to_chroma(gte_collection, embeddings_gte_1,
sentences1, "GTE", "texts1")
add_embeddings_to_chroma(e5_collection, embeddings_e5_1, se
ntences1, "E5", "texts1")

# Procesar texts2
sentences2 = texts2
embeddings_gte_2 = get_embeddings(model_gte, sentences2)
embeddings_e5_2 = get_embeddings(model_e5, sentences2)

add_embeddings_to_chroma(gte_collection, embeddings_gte_2,
sentences2, "GTE", "texts2")
add_embeddings_to_chroma(e5_collection, embeddings_e5_2, se
ntences2, "E5", "texts2")
```

Se procesan dos conjuntos de textos (`texts1` y `texts2`) y se generan los embeddings utilizando los modelos GTE y E5. Estos embeddings se almacenan en las colecciones correspondientes de ChromaDB.

7. Verificación del contenido almacenado

```
gte_documents = gte_collection.get(include=["embeddings",
"metadatas", "documents"])
e5_documents = e5_collection.get(include=["embeddings", "me
tadatas", "documents"])

print("Datos en GTE Collection:")
for doc, metadata in zip(gte_documents["documents"], gte_do
```

```
cuments["metadatas"]):
    print(f"Documento: {doc}, Metadata: {metadata}")

print("\nDatos en E5 Collection:")
for doc, metadata in zip(e5_documents["documents"], e5_docu
ments["metadatas"]):
    print(f"Documento: {doc}, Metadata: {metadata}")
```

Se recuperan los documentos almacenados en ChromaDB (junto con sus embeddings y metadatos) y se muestran en la consola para verificar que la información se haya almacenado correctamente.

8. Tokenización y creación de índices BM25

```
def tokenize_documents(documents):
    return [doc.lower().split() for doc in documents]

# Crear índices para BM25
tokenized_texts1 = tokenize_documents(documents_texts1)
tokenized_texts2 = tokenize_documents(documents_texts2)

bm25_texts1 = BM25Okapi(tokenized_texts1)
bm25_texts2 = BM25Okapi(tokenized_texts2)
```

Se tokenizan los textos de entrada (`texts1` y `texts2`), convirtiéndolos en una lista de palabras (en minúsculas). Luego, se crea un índice BM25 para cada conjunto de textos, que es útil para realizar búsquedas basadas en términos.

9. Función de consulta BM25

```
def query_bm25(bm25_index, query, documents, top_n=5):
    tokenized_query = query.lower().split()
    scores = bm25_index.get_scores(tokenized_query)
    ranked_indexes = np.argsort(scores)[::-1][:top_n]

    results = []
    for idx in ranked_indexes:
        results.append({
            "document": documents[idx],
```

```
            "score": scores[idx]
        })
    return results
```

Esta función toma una consulta y la busca en el índice BM25, devolviendo los documentos más relevantes (los que tienen los puntajes más altos) junto con sus respectivos puntajes.

10. Consultas y análisis de resultados

```
queries = ["como se juega?", "cuantos ganadores hay?"]

for query in queries:
    print(f"\nResultados para la consulta: '{query}' en {so
urce_label1}")
    results_texts1 = query_bm25(bm25_texts1, query, documen
ts_texts1, top_n=3)
    for result in results_texts1:
        print(f"Documento: {result['document']}, Score: {re
sult['score']:.4f}")

    print(f"\nResultados para la consulta: '{query}' en {so
urce_label2}")
    results_texts2 = query_bm25(bm25_texts2, query, documen
ts_texts2, top_n=3)
    for result in results_texts2:
        print(f"Documento: {result['document']}, Score: {re
sult['score']:.4f}")
```

Se realizan consultas en los índices BM25 de `texts1` y `texts2` y se muestran los 3 documentos más relevantes para cada consulta, junto con su puntaje.

11. Matriz de similitud y análisis

```
def create_similarity_matrix(embeddings):
    norm_embeddings = embeddings / np.linalg.norm(embedding
s, axis=1, keepdims=True)
    return np.dot(norm_embeddings, norm_embeddings.T)
```

```
def print_similarity_matrix(matrix, model_name, sentences, source_label):
    df = pd.DataFrame(matrix, index=[f"{source_label} - Frase {i+1}" for i in range(len(sentences))], columns=[f"{source_label} - Frase {i+1}" for i in range(len(sentences))])
    print(f"\nMatriz de similitud para {model_name} ({source_label}):")
    print(df.round(4))
```

- **create_similarity_matrix**: Esta función crea una matriz de similitud utilizando los embeddings. Primero normaliza los embeddings y luego calcula el producto punto entre ellos para obtener las similitudes.
- **print_similarity_matrix**: Muestra la matriz de similitud de manera legible en formato de tabla.

12. Búsqueda híbrida

```
def hybrid_search(bm25_index, embeddings, query, documents, top_n=5, alpha=0.5):
    bm25_scores = bm25_index.get_scores(tokenized_query)
    embedding_similarities = np.dot(embeddings, query_embedding)
    combined_scores = alpha * bm25_scores + (1 - alpha) * embedding_similarities
    ranked_indexes = np.argsort(combined_scores)[::-1][:top_n]
    ...
```

Esta función realiza una búsqueda híbrida combinando la similitud de los índices BM25 y los embeddings. Los puntajes se combinan mediante una ponderación (`alpha`), lo que permite controlar la influencia de cada tipo de similitud en los resultados finales.

Creación del Clasificador Basado en Ejemplos y Embeddings

Objetivo del Modelo

El propósito de este modelo es clasificar frases en diferentes categorías basadas en el juego *Viticulture*. Las clases definidas incluyen:

1. **Roles y Objetivos**
2. **Tareas Estacionales**
3. **Visitantes y su Impacto**
4. **Nada Relacionado**

Se utiliza una combinación de **embeddings** generados mediante el modelo `all-mpnet-base-v2` de *Sentence Transformers* y un modelo de **Regresión Logística Multinomial**.

Metodología Implementada

1. **Dataset Preparado**
   - Se recopilaron 16 frases divididas en las 4 categorías mencionadas.
   - Las clases fueron etiquetadas numéricamente (1-4).
2. **Embeddings con `all-mpnet-base-v2`**
   - El modelo de embeddings convierte las frases en vectores numéricos.
   - Esto permite representar los textos en un espacio multidimensional, capturando su significado semántico.
3. **Modelo de Clasificación**
   - Se utilizó **Regresión Logística Multinomial** como clasificador debido a su capacidad para manejar múltiples clases.
   - Se dividió el dataset en **entrenamiento (80%)** y **prueba (20%)** para evaluar el rendimiento.
4. **Métricas Utilizadas**
   - **Accuracy**: Precisión global del modelo.
   - **Precision, Recall, F1-Score**: Métricas detalladas por clase.

Resultados Obtenidos

1. Precisión Global del Modelo

La precisión del modelo fue del **50%**, lo que indica que el modelo acertó la clasificación en la mitad de los casos evaluados.

2. Reporte de Clasificación

Clase	Precision	Recall	F1-Score	Soporte
Roles y Objetivos (1)	1.00	0.00	0.00	2
Tareas Estacionales (2)	1.00	1.00	1.00	1
Visitantes (3)	0.00	1.00	0.00	0
Nada Relacionado (4)	1.00	1.00	1.00	1

- **Clase 1 (Roles y Objetivos):**

  Aunque la precisión es alta, el *Recall* es **0**, lo que indica que el modelo no predijo correctamente ninguna frase de esta clase.

- **Clase 2 (Tareas Estacionales):**

  El modelo clasificó correctamente todos los ejemplos de esta clase.

- **Clase 3 (Visitantes y su Impacto):**

  No hubo ejemplos en el conjunto de prueba, lo que dificulta evaluar el desempeño en esta clase (*Support = 0*).

- **Clase 4 (Nada Relacionado):**

  Se logró una clasificación perfecta.

Análisis de los Resultados

1. **Desequilibrio de Clases**
   - El dataset contiene pocas frases por clase (4 por clase en total) y una cantidad mínima de ejemplos en el conjunto de prueba.
   - La falta de representatividad de ciertas clases (como clase 3) afecta las métricas y limita la generalización del modelo.
2. **Rendimiento Desigual entre Clases**

- El modelo mostró un rendimiento excelente para las clases **2** y **4**, pero no pudo identificar correctamente ejemplos de la clase **1**.
- La clase **3** no tuvo evaluación debido a la ausencia de muestras en el conjunto de prueba.

3. **Generalización del Modelo**
   - Aunque el modelo logra capturar correctamente frases relacionadas a "Tareas Estacionales" y "Nada Relacionado", su capacidad para identificar "Roles y Objetivos" es limitada.

Clasificación de Nuevos Textos

Al evaluar frases nuevas, el modelo predice correctamente aquellas relacionadas con tareas estacionales y visitantes, mientras clasifica de forma separada los textos irrelevantes:

Ejemplos de Resultados:

Texto	Predicción
"Durante el verano, los jugadores pueden construir estructuras."	**Tareas Estacionales**
"Los visitantes aportan habilidades útiles temporalmente."	**Visitantes y su Impacto**
"¿Quién es el presidente actual de Estados Unidos?"	**Nada Relacionado**
"Los jugadores deben plantar vides para producir vino."	**Roles y Objetivos**

Conclusiones y Recomendaciones

1. **Conclusión Principal:**

   El modelo tiene un rendimiento limitado debido al tamaño reducido y desequilibrado del dataset. La precisión global del **50%** refleja que el modelo no generaliza bien para todas las clases.

2. **Recomendaciones para Mejorar el Modelo:**
   - **Ampliar el Dataset**: Incorporar más frases por clase para mejorar la representatividad.
   - **Balancear las Clases**: Asegurar que cada clase tenga una cantidad similar de ejemplos.

- **Ajustar Hiperparámetros**: Probar diferentes parámetros para la Regresión Logística (por ejemplo, cambiar el solver).
- **Explorar Otros Modelos**: Utilizar clasificadores más complejos como SVM, Random Forest o redes neuronales simples.
- **Aumentar los Embeddings**: Experimentar con modelos de embeddings más robustos o ajustar el fine-tuning del modelo *all-mpnet-base-v2*.

Este es el otro clasificador implementado basado en LLM (Unidad 6)

Estructura del Código

1. **Definición de plantillas Jinja para formato de mensajes**
   - Se utiliza la biblioteca Jinja para crear mensajes dinámicos que siguen un formato específico compatible con un modelo LLM (Zephyr en este caso).
   - La función `plantilla_instruccion_zephyr` adapta los mensajes de entrada para estructurarlos como un diálogo entre "usuario", "asistente", y "sistema".
2. **Conexión con el modelo LLM**
   - La función `conexion_llm` realiza una solicitud POST a la API de Hugging Face para generar texto basado en el prompt dado.
   - Se establece una temperatura baja (0.01) para obtener respuestas más determinísticas.
3. **Clasificación inicial de categorías (Unit 6)**
   - Una plantilla estática (`PLANTILLA_CLASIFICACION_BASE_DATOS`) organiza las categorías en texto para guiar al modelo Zephyr en la clasificación de una consulta en una de tres categorías:
     - **Categoría 1:** Información sobre trabajadores relacionados con madera.
     - **Categoría 2:** Descripciones generales del juego *Viticulture*.
     - **Categoría 3:** Información híbrida detallada del juego *Viticulture*.

- La decisión inicial es manejada por el modelo LLM, utilizando prompts diseñados específicamente para esta tarea.

4. **Derivación de consultas a fuentes de datos**
   - Basado en la categoría asignada, las consultas se procesan con funciones específicas:
     - **Consulta SPARQL:** Para información en grafos.
     - **Consulta híbrida:** Combina datos de múltiples fuentes.
   - La función `generar_respuesta_guia` asegura que la respuesta se formule según las reglas proporcionadas (respuestas cortas, en español, y basadas únicamente en contexto).
5. **Uso del clasificador zero-shot**
   - Se utiliza un modelo *zero-shot* (`mDeBERTa-v3-base-mnli-xnli`) para clasificar preguntas sin entrenamiento previo en las siguientes etapas:
     - **Clasificación por categoría:** Etiquetas '1', '2', '3', '4'.
     - **Clasificación por consulta específica:** Etiquetas descriptivas como "trabajadores_relacionados_madera".
   - Este enfoque es independiente del LLM y proporciona una comparación adicional.
6. **Evaluación del clasificador**
   - Las predicciones se comparan con las categorías reales en un dataset anotado.
   - Se calculan métricas de rendimiento, como porcentaje de aciertos generales y por categoría.

Resultados del Experimento

1. **Porcentaje de aciertos generales:**
   - El clasificador zero-shot alcanzó un **31.25% de aciertos**, lo que significa que menos de 1/3 de las predicciones fueron correctas en la clasificación de categorías.
2. **Desglose por categoría:**
   - **Categoría 1:** 25.00% de aciertos (1 acierto de 4 preguntas).

- **Categoría 2:** 100.00% de aciertos (4 aciertos de 4 preguntas).
- **Categoría 3:** 0.00% de aciertos (0 aciertos de 4 preguntas).
- **Categoría 4:** 0.00% de aciertos (0 aciertos de 4 preguntas).

Toma de decisiones:

Se eligió el clasificador basado en embeddings y ejemplos porque mostró un mejor desempeño y mayor precisión que el clasificador basado en LLM (zero-shot). Esto se debe a su capacidad para capturar similitudes semánticas de manera efectiva y aprovechar ejemplos específicos, lo que garantiza respuestas más precisas y consistentes.

Resumen: Ejercicio 2 - Agente

Este ejercicio se basa en el ejercicio anterior y se centra en el desarrollo de un agente basado en el concepto de ReAct, utilizando la librería Llama-Index. El objetivo principal es maximizar las capacidades del agente al integrar al menos tres herramientas diferentes: `doc_search()` para buscar información en documentos, `graph_search()` para buscar en bases de datos de grafos, y `table_search()` para acceder a datos tabulares.

Instalación de Ollama y la configuración del agente ReAct

Este proyecto está diseñado para configurar un sistema de búsqueda basado en el modelo de lenguaje **Ollama** con la interfaz de API de OpenAI. A través de esta configuración, se pretende construir un agente de conversación que interactúa con varios recursos (documentos, bases de datos, y grafos) usando el modelo de lenguaje para responder consultas sobre el juego **Viticulture**. Aquí te explico cada paso de forma detallada:

1. Instalación de Ollama

Primero, descargamos e instalamos el software **Ollama**, que proporciona acceso a modelos de lenguaje como Phi-3 Medium.

Descarga de Ollama

```
!curl -fsSL https://ollama.com/install.sh | sh
```

- Usamos `curl` para descargar el script de instalación desde la página oficial de Ollama y lo ejecutamos directamente con `sh`. Esto asegura que instalemos la última versión de Ollama en nuestro entorno.

Inicio de Ollama en background

```
!rm -f ollama_start.sh
!echo '#!/bin/bash' > ollama_start.sh
!echo 'ollama serve' >> ollama_start.sh
!chmod +x ollama_start.sh
!nohup ./ollama_start.sh &
```

- Eliminamos cualquier script anterior (`ollama_start.sh`), luego creamos un nuevo script que contiene el comando `ollama serve`, que inicia el servidor Ollama.
- Usamos `chmod +x` para hacer que el script sea ejecutable y `nohup` para ejecutarlo en segundo plano, de forma que siga funcionando aunque cerremos el terminal.

2. Descarga del modelo Phi-3 Medium

```
!ollama pull phi3:medium > ollama.log
```

- Usamos `ollama pull` para descargar el modelo Phi-3 Medium de Ollama. El `>` redirige la salida a un archivo de log, `ollama.log`, para verificar el proceso de descarga.

Verificación del modelo disponible

```
!ollama list
```

- Ejecutamos `ollama list` para asegurarnos de que el modelo Phi-3 Medium esté disponible y correctamente instalado en Ollama.

3. Configuración del proxy para utilizar la API de OpenAI

Instalamos las herramientas necesarias para configurar un proxy que permita que Ollama funcione con la misma interfaz que la API de OpenAI:

```
%%capture
!pip install litellm[proxy]
!nohup litellm --model ollama/phi3:medium --port 8000 > lit
ellm.log 2>&1 &
```

- Instalamos `litellm` con la opción `[proxy]` para permitir que Ollama exponga una API compatible con OpenAI en el puerto 8000.
- Usamos `nohup` nuevamente para ejecutar `litellm` en segundo plano, de manera que el servidor se mantenga corriendo sin interrumpirse.

4. Configuración del agente ReAct

Instalamos las librerías necesarias para la integración de herramientas en un agente ReAct que pueda consultar información de diversas fuentes como documentos, tablas y grafos:

```
%%capture
!pip install llama-index-llms-ollama llama-index pygooglewe
ather wikipedia
```

- `llama-index-llms-ollama` permite la integración de modelos de Ollama con LlamaIndex, un sistema que facilita el trabajo con grandes volúmenes de datos en busca de respuestas.
- Además, instalamos `pygoogleweather` y `wikipedia` para obtener información sobre el clima y la Wikipedia, que puede ser útil para ampliar las respuestas del agente.

Desinstalación e instalación de Ollama

```
!pip uninstall ollama -y
!pip install ollama
```

- Desinstalamos cualquier versión previa de `ollama` y volvemos a instalarla para asegurarnos de tener la versión más reciente.

5. Prueba de conexión al servidor Litellm

Verificamos que Litellm, el proxy que conecta Ollama con la API de OpenAI, esté funcionando correctamente:

```
url = "http://localhost:8001"
try:
    response = requests.get(url)
    if response.status_code == 200:
        print("Conexión exitosa al servidor litellm.")
    else:
        print(f"Error del servidor: {response.status_cod
e}")
except Exception as e:
    print(f"No se pudo conectar al servidor litellm: {e}")
```

- Realizamos una solicitud HTTP al servidor `litellm` que debería estar corriendo en el puerto 8001.
- Si la respuesta es satisfactoria (código 200), significa que el servidor está funcionando correctamente.

6. Configuración del Agente ReAct

Definimos las funciones de búsqueda que el agente ReAct utilizará para responder a las consultas:

```
def doc_search(query: str) -> str:
    """Busca información en los documentos simulados."""
    documentos = [texto_doc1_leido, texto_doc2_leido]
    resultados = [texto for texto in documentos if query.lo
wer() in texto.lower()]
    if resultados:
        return "\n".join(resultados)
    return "No se encontró información en los documentos."
```

- `doc_search`: Busca en una lista de documentos simulados por coincidencias con la consulta del usuario.

```
def graph_search(query: str) -> str:
    """Busca información en el grafo RDF."""
    try:
        q = f"""
        SELECT ?o WHERE {{
```

```
            ?s ?p ?o .
            FILTER(CONTAINS(LCASE(STR(?o)), "{query.lower
()}"))
        }}
        """
        resultados = g.query(q)
        respuesta = [str(row[0]) for row in resultados]
        if respuesta:
            return "\n".join(respuesta)
        return f"No se encontró información en el grafo rel
acionada con '{query}'."
    except Exception as e:
        return f"Error al buscar en el grafo: {e}"
```

- `graph_search`: Realiza una búsqueda en un grafo RDF utilizando consultas SPARQL. Devuelve los resultados si existen.

```
def table_search(query: str) -> str:
    """Busca información en una tabla de datos."""
    try:
        resultados = df_componentes.applymap(lambda x: quer
y.lower() in str(x).lower()).any(axis=1)
        filas_encontradas = df_componentes[resultados]

        if filas_encontradas.empty:
            return f"No se encontró información relacionada
con '{query}' en la tabla."
        else:
            return f"Resultados encontrados:\n{filas_encont
radas.to_string(index=False)}"
    except Exception as e:
        return f"Error en table_search: {str(e)}"
```

- `table_search`: Busca en una tabla de datos (un DataFrame de pandas) para encontrar registros que coincidan con la consulta.

7. Configuración del agente ReAct

Creamos un **agente ReAct** que utiliza las herramientas de búsqueda definidas anteriormente:

```
llm = Ollama(
    model="phi3:medium",
    request_timeout=120.0,
    temperature=0.1,
    context_window=4096
)
Settings.llm = llm

agent = ReActAgent.from_tools(
    tools,
    llm=llm,
    verbose=True,
    chat_formatter=ReActChatFormatter(),
    system_prompt="""Eres un asistente útil que responde en
español sobre el juego Viticulture. Debes seguir ESTRICTAME
NTE el formato:

Thought: Aquí explico qué necesito hacer
Action: nombre_de_la_herramienta
Action Input: "parametro"

Observation: [Resultado de la herramienta]
... [Repetir el proceso si es necesario]
Final Answer: La respuesta final combinando toda la informa
ción
"""
)
```

- Se configura el modelo LLM (Phi-3 Medium) con parámetros como el tiempo de espera (`request_timeout`), la temperatura (`temperature`), y la ventana de contexto (`context_window`).
- Se crea el agente ReAct, que está configurado para responder preguntas sobre el juego **Viticulture**. El `system_prompt` guía al agente para que actúe de manera coherente.

8. Interacción con el agente

Finalmente, definimos una función para interactuar con el agente y probamos algunas consultas de ejemplo:

```
def chat_con_agente(query: str):
    """Función para interactuar con el agente ReAct sobre V
iticulture."""
    try:
        if not query.strip():
            return "La consulta está vacía"
        response = agent.chat(query)
        return response
    except Exception as e:
        return f"Error al procesar la consulta: {str(e)}"
```

- Esta función recibe una consulta y la pasa al agente ReAct para obtener la respuesta.

9. Ejecución de ejemplos

Finalmente, ejecutamos ejemplos de interacción con el agente sobre preguntas relacionadas con el juego **Viticulture**:

```
def ejecutar_ejemplo():
    print("=== Ejemplo de interacción con el agente ReAct s
obre Viticulture ===")
    queries = [
        "¿Qué componentes tiene Viticulture?",
        "Cuéntame sobre el juego Viticulture.",
        "¿Como se juega?",
        "¿Cuantos jugadores pueden jugar?",
        "¿Contame sobre las cartas?"
    ]

    for i, query in enumerate(queries):
        print(f"\nConsulta {i+1}: {query}")
        response = chat_con_agente(query)
        print(f"Respuesta {i+1}: {response}")
```

```
        print("-----------------------------------------
----------")
```

Resultados del agente ReAct sobre Viticulture

El agente ReAct tuvo resultados mixtos: logró responder preguntas generales de manera adecuada, como cómo se juega o cuántos jugadores pueden participar, utilizando razonamiento implícito sin herramientas adicionales. Sin embargo, falló al usar herramientas específicas como `doc_search`, `table_search` o `graph_search`, debido a errores en los argumentos esperados o tiempos de espera. Esto muestra que el agente tiene potencial para responder preguntas simples directamente, pero necesita ajustes en la integración de herramientas para manejar consultas más complejas o específicas.

Ejemplos de prompts donde se necesita más de una herramienta y evaluación de los resultados:

1. **Prompt:** *"Cuéntame sobre el juego Viticulture."*
   - **Lo que intentó hacer:** Usar `doc_search` y otras herramientas como `table_search` y `graph_search`.
   - **Resultado:**
     - Falló con `doc_search` por un parámetro incorrecto.
     - Falló con `table_search` y `graph_search` porque faltaban argumentos obligatorios como `query`.
   - **Problema:** El agente no logró usar las herramientas correctamente y terminó sin respuesta útil.
2. **Prompt:** *"¿Qué componentes tiene Viticulture?"*
   - **Lo que intentó hacer:** Buscar información usando `doc_search` o herramientas similares.
   - **Resultado:** La consulta quedó colgada por *timeout*.
   - **Problema:** El agente no pudo completar la búsqueda a tiempo y no ofreció ninguna alternativa.
3. **Prompt:** *"Contame sobre las cartas del juego."*
   - **Lo que intentó hacer:** Buscar detalles en documentos o tablas.

- **Resultado:** Otro *timeout*.
- **Problema:** El tiempo de espera no fue suficiente y el agente no supo qué hacer después.

4. **Prompt:** *"Busca los recursos disponibles sobre Viticulture."*
   - **Lo que intentó hacer:** Usar varias herramientas.
   - **Resultado:** No pudo porque faltaban argumentos como `query`.
   - **Problema:** No hay una validación previa para asegurar que los parámetros necesarios estén presentes.
5. **Prompt:** *"Dame ejemplos de eventos especiales en Viticulture."*
   - **Lo que intentó hacer:** Buscar en documentos o tablas.
   - **Resultado:** El agente no encontró cómo usar ninguna herramienta y no intentó dar una respuesta parcial.
   - **Problema:** No hay un plan B cuando fallan las herramientas.

Ejemplos donde el agente falla o no responde bien:

1. **Ejemplo 1:** *"Cuéntame sobre el juego Viticulture."*
   - **Fallo:** Mala configuración de herramientas y parámetros faltantes.
   - **Por qué pasa:** El agente no valida los parámetros antes de llamar a las herramientas.
2. **Ejemplo 2:** *"Contame sobre las cartas del juego."*
   - **Fallo:** Timeout sin solución.
   - **Por qué pasa:** El tiempo de espera es corto y el agente no intenta otras opciones.
3. **Ejemplo 3:** *"Busca los recursos sobre Viticulture."*
   - **Fallo:** Parámetros faltantes.
   - **Por qué pasa:** No hay una verificación previa para asegurar que la herramienta se use bien.

Mejoras que se podrían hacer:

1. **Revisar parámetros antes de ejecutar herramientas:**

Asegurarse de que todas las herramientas tengan lo que necesitan para funcionar (por ejemplo, que `query` no esté vacío).

2. **Manejar *timeouts* mejor:**

   Ajustar los tiempos de espera y agregar un sistema de reintento automático si algo falla.

3. **Tener un plan B (fallback):**

   Si una herramienta falla, el agente debería intentar dar una respuesta parcial o usar otra fuente de información.

4. **Combinar herramientas de forma más inteligente:**

   Si una herramienta no puede resolver la consulta por completo, el agente debería usar otra para completar la información.

5. **Detectar errores en tiempo real:**

   Implementar un sistema para que el agente entienda por qué falló una herramienta y lo corrija sobre la marcha.