Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Procesamiento del Lenguaje Natural

Trabajo Práctico Final

Rodríguez y Barros, Francisco: R-4559/4

Introducción **2**
Objetivos **2**
RAG **3**
Resumen **3**
Bases de Datos **3**
1. Base de Datos de Grafos 3
2. Base de Datos Tabular 4
3. Base de Datos Vectorial (Documental) 5
Clasificadores **6**
Clasificador con Regresión Logística 6
Clasificador basado en LLM 7
Consultas Dinámicas **8**
Retriever y ReRanker 8
Retriever 9
Reranker 9
Base de datos tabular 10
Base de datos de grafos 12
Pipeline Completo **15**
Funcionamiento Completo del Pipeline 15
Agente **17**

Introducción

En este proyecto se desarrolló un **chatbot experto en el juego de mesa "Viticulture"** utilizando dos enfoques principales: **RAG (Retrieval-Augmented Generation)** y el desarrollo de un **Agente ReAct.** El objetivo fue construir un sistema que pueda responder consultas sobre distintos aspectos del juego, aprovechando múltiples fuentes de información.

Objetivos

1. Implementar un sistema basado en **RAG** que utilice:
    - **Documentos de texto**: Reglas, estrategias, opiniones y otros textos relacionados.
    - **Datos tabulares**: Componentes estructurados del juego como cartas, recursos y trabajadores.
    - **Base de datos de grafos**: Relaciones entre elementos clave, como vides, uvas, vinos y estructuras.
2. Desarrollar un **agente basado en ReAct** utilizando **Llama-Index**, que sea capaz de:
    - Elegir automáticamente entre las fuentes de datos disponibles.
    - Responder consultas complejas combinando resultados de más de una fuente.

3. Optimizar el proceso de recuperación y generación de información para obtener respuestas precisas, eficientes y acordes a la consulta del usuario.

RAG

Resumen

En este primer punto se desarrolló un sistema RAG para responder preguntas relacionadas con el juego Viticulture, integrando datos de diferentes fuentes (tabulares, de grafos y vectoriales)

Se implementaron dos clasificadores (uno basado en regresión logística y otro en un modelo LLM) para determinar la fuente más relevante según la consulta del usuario.

El sistema utiliza consultas dinámicas para recuperar la información correspondiente de cada base: filtrado por palabras clave en la base tabular, exploración de relaciones en el grafo con NetworkX y búsquedas híbridas en documentos utilizando técnicas semánticas y por palabras claves.

Finalmente un pipeline unifica estos componentes, clasificando la consulta, recuperando información relevante y generando respuestas claras con un LLM.

Bases de Datos

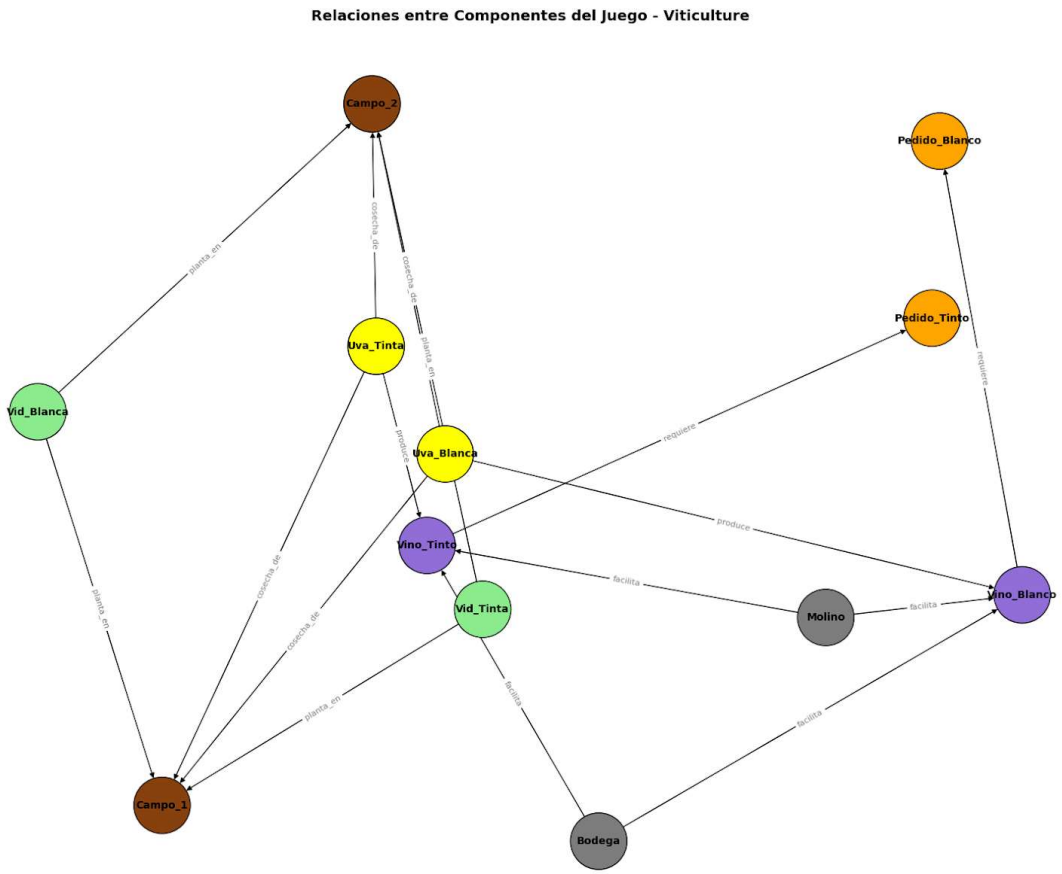
1. Base de Datos de Grafos

Esta base de datos modela las relaciones entre los diferentes componentes del juego *Viticulture*. Esta representación permite capturar de forma estructurada cómo interactúan las vides, campos, uvas, vinos, pedidos y estructuras dentro del juego.
Se utilizó la librería **NetworkX** para la creación y manipulación de la misma.

Estructura y Relaciones:
El grafo es dirigido, donde:

- **Nodos**: Representan componentes del juego, categorizados en:
  - **Vides**: Ejemplo, Vid_Tinta, Vid_Blanca.
  - **Uvas**: Ejemplo, Uva_Tinta, Uva_Blanca.
  - **Vinos**: Ejemplo, Vino_Tinto, Vino_Blanco.
  - **Pedidos**: Ejemplo, Pedido_Tinto, Pedido_Blanco.
  - **Campos**: Ejemplo, Campo_1, Campo_2.
  - **Estructuras**: Ejemplo, Bodega, Molino.
- **Aristas**: Representan relaciones específicas entre los nodos:
  - planta_en: Relaciona vides con campos donde pueden plantarse.
  - cosecha_de: Relaciona uvas con campos de origen.
  - produce: Relaciona uvas con vinos producidos.

- requiere: Relaciona vinos con pedidos que los necesitan.
- facilita: Relaciona estructuras con vinos que apoyan su producción.



El propósito de la base de datos de grafos es principalmente para responder preguntas relacionadas con las relaciones en el juego, como:

- *¿Qué vinos puedo producir si tengo Uva_Tinta?*
- *¿Qué recursos facilita la Bodega?*

2. Base de Datos Tabular

Esta base está desarrollada con **Pandas** y representa una vista estructurada y organizada de los componentes del juego, incluyendo cartas, fichas, tableros y estructuras. Cada componente cuenta con atributos específicos como su cantidad, descripción y uso principal.

La información está organizada en un dataframe con las siguientes columnas:

- **Componente**: El nombre del componente (Cartas de Vid, Fichas de Vino, etc.).
- **Cantidad**: El número total de elementos disponibles en el juego.
- **Descripción**: Una breve explicación del componente y su función.
- **Usos Principales**: Describe las acciones o beneficios del componente dentro del juego.

	Componente	Cantidad	Descripción	Usos Principales
1	Cartas de Vid	42	Representan diferentes tipos de uvas.	Plantación de viñas en los campos.
2	Cartas de Pedido	36	Pedidos de clientes que otorgan puntos al completarse.	Cumplir pedidos y ganar puntos de victoria.
3	Cartas de Visitante de Verano	40	Ofrecen ventajas durante el verano.	Habilitar acciones adicionales en verano.
4	Cartas de Visitante de Invierno	40	Ofrecen ventajas durante el invierno.	Habilitar acciones adicionales en invierno.
5	Trabajadores	30	Meeples utilizados para realizar acciones.	Realizar tareas como cosechar, plantar y construir.
6	Tablero Principal	1	Muestra las estaciones y las acciones disponibles.	Proporcionar un mapa de las acciones posibles.
7	Tableros Individuales	6	Tableros para cada jugador con su viñedo.	Personalizar estrategias según los tableros individuales.
8	Fichas de Monedas	50	Monedas para realizar transacciones.	Comprar estructuras, plantar y realizar intercambios.
9	Fichas de Vino	50	Fichas de cristal que representan vinos.	Almacenar y vender vinos producidos.
10	Fichas de Uva	50	Fichas de cristal que representan uvas.	Almacenar y procesar las uvas cosechadas.
11	Estructuras	12	Mejoras que pueden construirse en el viñedo.	Ampliar capacidades del viñedo.
12	Ficha de Primer Jugador	1	Marca quién es el primer jugador del turno.	Determinar quién toma el primer turno.
13	Fichas de Residuo	10	Indican las sobras o productos no utilizados en el viñedo.	Registrar recursos desperdiciados o sobrantes.
14	Marcadores de Puntos	6	Fichas que registran los puntos de victoria de cada jugador.	Indicar los puntos acumulados por cada jugador.
15	Fichas de Campo	12	Indican qué campos están ocupados por vides.	Gestionar los campos disponibles para plantaciones.
16	Cartas de Evento Especial	8	Acciones únicas que modifican las reglas temporalmente.	Añadir variabilidad y nuevos desafíos al juego.
17	Fichas de Estación	4	Indican las estaciones para cada acción disponible.	Definir las estaciones para organizar el turno.
18	Cartas de Construcción	20	Representan edificios y estructuras a construir.	Ayudar a planificar y ejecutar construcciones.
19	Cartas de Acción	24	Acciones especiales que pueden ejecutarse.	Ofrecer acciones estratégicas específicas.
20	Fichas de Jugador	6	Marcadores para identificar a cada jugador.	Identificar qué jugador realiza cada acción.
21	Cartas de Objetivo	10	Definen objetivos adicionales para puntuar.	Cumplir objetivos para obtener recompensas.
22	Fichas de Recursos	15	Recursos adicionales utilizados en la partida.	Gestionar y usar recursos en el viñedo.

3. Base de Datos Vectorial (Documental)

Esta base de datos responde preguntas relacionadas con las **reglas y estrategias** del juego, como:

- *¿Cuáles son las reglas del juego?*
- *¿Qué estrategias existen para jugadores principiantes?*

Contiene información textual relacionada con las **reglas del juego**, estrategias y otros detalles provenientes de documentos PDF. Esta base permite realizar búsquedas **semánticas e híbridas** sobre grandes volúmenes de texto, garantizando respuestas precisas.

Construcción:

- **Extracción del Texto**:
  Para extraer texto de un documento PDF se implementó el uso de la librería **PyPDF2.**
- **Limpieza del Texto**:
  Se aplicó preprocesamiento para eliminar caracteres especiales, normalizar espacios y garantizar que el texto sea claro y útil.
- **División del Texto en Fragmentos (Chunks)**:
  Se utilizó el método RecursiveCharacterTextSplitter de LangChain para dividir el texto en fragmentos solapados, asegurando continuidad en la información.
- **Vectorización**:
  Los fragmentos se vectorizaron utilizando SentenceTransformer con el modelo paraphrase-multilingual-MiniLM-L12-v2.

- **Almacenamiento en ChromaDB**:
  Los embeddings se almacenaron en una base de datos **ChromaDB**, permitiendo búsquedas rápidas y eficientes.

Consultas:
Se implementó una búsqueda híbrida utilizando **BM25** para búsqueda por palabras clave y **ChromaDB** para búsqueda semántica. Los resultados se combinaron y clasificaron mediante la función reranker_hibrido.

Clasificadores

El principal objetivo del clasificador es **determinar la fuente de datos correcta** (tabular, vectorial o grafos) en función de la consulta del usuario. Por ejemplo:

- Preguntas sobre componentes del juego → Base **tabular.**
- Preguntas sobre reglas del juego → Base **vectorial**.
- Preguntas sobre relaciones entre componentes → Base **de grafos.**

Clasificador con Regresión Logística

Como modelo de clasificador se implementó una **Regresión Logística** con la librería Scikit-Learn. La elección de este modelo se debe a su eficiencia y capacidad para resolver problemas de clasificación con múltiples clases.

¿Por qué se usó un modelo de Regresión Logística?

- Es simple y eficiente.
- Funciona bien en tareas de clasificación con datos linealmente separables.
- Es fácil de interpretar y ajustar.

Para entrenar el modelo, se creó un dataframe que contiene **consultas simuladas** (preguntas comunes que los usuarios harían) y la **fuente esperada** donde se encuentra la respuesta:

Cada pregunta está diseñada para representar un tipo específico de información y se incluye en la columna **consulta**, mientras que la fuente de datos esperada está en **fuente**

Para convertir el texto en un formato numérico que el modelo pueda entender, se utilizaron embeddings con el modelo pre entrenado paraphrase-multilingual-MiniLM-L12-v2 de SentenceTransformers.

Los embeddings transforman cada pregunta en una representación vectorial en un espacio de alta dimensión lo que permite capturar el significado semántico de cada consulta.

```
embed_model =
SentenceTransformer("paraphrase-multilingual-MiniLM-L12-v2")

def generar_embeddings(textos):

    embeddings = embed_model.encode(textos, batch_size=32,
show_progress_bar=True)

    return np.array(embeddings)
```

Para entrenar el modelo se dividieron los datos del dataframe en entrenamiento (80%) y prueba (20%) utilizando train_test_split para evaluar posteriormente su rendimiento. Esto asegura que el modelo se entrene y se valide en datos distintos.

Resultados

```
Accuracy: 0.9444444444444444
Reporte de Clasificación:
              precision    recall  f1-score   support

      grafos       1.00      0.75      0.86         4
     tabular       1.00      1.00      1.00         8
   vectorial       0.86      1.00      0.92         6

    accuracy                           0.94        18
   macro avg       0.95      0.92      0.93        18
weighted avg       0.95      0.94      0.94        18
```

En resumen, el modelo es confiable y presenta un buen rendimiento general.

Clasificador basado en LLM

Para este clasificador se usó un **LLM (Large Language Model)** alojado en la plataforma **HuggingFace**, el modelo **Qwen/Qwen2.5-72B-Instruct.** El objetivo principal de este clasificador es analizar el texto de entrada del usuario y asignarlo correctamente a una de las tres clases principales: VECTORIAL, TABULAR o GRAFOS.

La principal función del clasificador LLM es **interpretar las consultas en lenguaje natural** ingresadas por el usuario y determinar qué fuente de datos debe consultarse para responder a dicha consulta.

Se implementó a través de InferenceClient de la librería huggingface_hub para interactuar con el modelo Qwen/Qwen2.5-72B-Instruct. Este modelo es conocido por su capacidad de procesar tareas de clasificación de texto con alta precisión, especialmente en contextos de lenguaje natural.

En el prompt pasado al modelo se especifica claramente las reglas y ejemplos que debe seguir para clasificar las consultas.

- Se establece el rol del modelo como clasificador y se limita estrictamente su salida a las tres clases definidas: VECTORIAL, TABULAR o GRAFOS.
- Para guiarlo se proporcionan ejemplos concretos de preguntas junto con la clasificación correcta. Esto permite al LLM aprender el formato esperado y realizar la inferencia de manera precisa.

```python
from huggingface_hub import InferenceClient
def clase_LLM(input_usuario):
  cliente = InferenceClient(api_key="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")
  clases = ["VECTORIAL", "TABULAR", "GRAFOS"]
  prompt = [{
      "role": "system",
      "content": f"Sos un modelo de clasificacion de texto, tu funcion es clasificar una frase en una de las siguientes clases ['VECTORIAL', 'TABULAR', 'GRAFOS']"
                 " Unicamente tenes que centrarte en clasificar la frase en una de esas clases, NINGUNA OTRA."
                 " Como respuesta debes devolver UNICAMENTE la clase asignada a la frase."
                 " EJEMPLO: '¿Qué puedo producir si tengo una bodega y vino blanco?': GRAFOS"
                 " EJEMPLO: '¿Cuáles son los componentes del juego?': TABULAR"
                 " EJEMPLO: '¿Cuáles son las reglas para jugar en primavera?': VECTORIAL"
                 " EJEMPLO: 'Opiniones del juego en general': VECTORIAL"},
                 {"role": "user", "content": input_usuario}]
  completion = cliente.chat.completions.create(
      model="Qwen/Qwen2.5-72B-Instruct",
      messages=prompt,
      temperature=0.5
  )
  return completion.choices[0].message['content']
```

Algunos ejemplos de uso:

```
print(clase_LLM('que puedo hacer si tengo uvas?'))
print(clase_LLM('cuantas fichas de vino hay?'))
print(clase_LLM('que opina la gente?'))
print(clase_LLM('que dicen las reglas sobre la cantidad de jugadores?'))
```

```
GRAFOS
TABULAR
VECTORIAL
VECTORIAL
```

Considero que el clasificador basado en LLM es la mejor opción para este proyecto por su capacidad para manejar consultas diversas y complejas y clasificar correctamente cada una de ellas.

Consultas Dinámicas

Retriever y ReRanker

Esta sección es fundamental dentro del pipeline de RAG porque permite localizar y seleccionar de manera eficiente la información relevante de una base de datos de documentos cuando un usuario realiza una consulta.
Se combinaron dos métodos complementarios de búsqueda: la búsqueda semántica con embeddings y la búsqueda basada en palabras clave con BM25. Al integrar ambos enfoques y aplicar un mecanismo de reranking, se logra que los resultados sean más precisos y estén mejor alineados con lo que busca el usuario.

Retriever

El retriever actúa como un filtro que selecciona las piezas de información más relevantes de una base de datos extensa, permitiendo que el modelo de lenguaje solo procese lo necesario y no toda la base de conocimiento.

En este caso, se combinaron dos estrategias:

1. **Búsqueda BM25**:
   Este método usa el modelo probabilístico BM25 para puntuar documentos basados en coincidencias de palabras clave con la consulta del usuario. BM25 se basa en la frecuencia de palabras clave.
2. **Búsqueda Semántica con ChromaDB y embeddings**:
   La búsqueda semántica usa los embeddings generados con el modelo all-MiniLM-L6-v2, que convierte el texto en vectores numéricos de alta dimensión. Esto permite capturar no solo palabras clave, sino también el significado de la consulta. La búsqueda en ChromaDB utiliza estos embeddings para calcular distancias entre la consulta y los documentos almacenados.

Reranker

El reranker híbrido se implementó para combinar los resultados de BM25 y la búsqueda semántica en una única lista ordenada. Cada técnica devuelve una lista de documentos y sus respectivos puntajes:

- En BM25, los puntajes son positivos y reflejan qué tan bien coinciden las palabras clave.
- En la búsqueda semántica, las distancias se invierten y normalizan para convertirlas en una medida de similitud.

El procedimiento general del reranker incluye:

1. Para que los puntajes de BM25 sean comparables con las similitudes de la búsqueda semántica se normalizan a un rango común (0 a 2) utilizando MinMaxScaler.
2. Transformar las distancias de la búsqueda semántica en similitudes (también dentro del rango 0 a 2).
3. Combinar los puntajes normalizados de ambas búsquedas. Si un documento aparece en ambas listas, sus puntajes se suman. Si solo aparece en una, el puntaje faltante se asigna como cero.
4. Ordenar los resultados combinados de mayor a menor puntaje, priorizando aquellos documentos que tengan alta relevancia tanto para BM25 como para la búsqueda semántica.

```python
def reranker_hibrido(consulta: str, top_resultados: int = 5) -> List[Tuple[str, float]]:
    """
    Combina resultados de BM25 y búsqueda semántica, normalizando puntajes y ordenando.
    """
    # Obtener resultados individuales
    resultados_bm25 = busqueda_bm25(consulta, top_resultados)
    resultados_semanticos = busqueda_semantica(consulta, top_resultados)

    # Normalizar puntajes de BM25 al rango [0, 2]
    puntajes_bm25 = [puntaje for _, puntaje in resultados_bm25]
    scaler = MinMaxScaler(feature_range=(0, 2))
    puntajes_bm25_normalizados = scaler.fit_transform(np.array(puntajes_bm25).reshape(-1, 1)).flatten()

    # Convertir distancias semánticas en similitud (rango [0, 2])
    puntajes_semanticos = [2 - distancia for _, distancia in resultados_semanticos]

    # Combinar los resultados
    resultados_combinados = {}
    for (doc, puntaje) in zip([res[0] for res in resultados_bm25], puntajes_bm25_normalizados):
        resultados_combinados[doc] = {"bm25": puntaje, "semantic": 0}

    for (doc, puntaje) in zip([res[0] for res in resultados_semanticos], puntajes_semanticos):
        if doc in resultados_combinados:
            resultados_combinados[doc]["semantic"] = puntaje
        else:
            resultados_combinados[doc] = {"bm25": 0, "semantic": puntaje}

    # Calcular puntajes combinados y ordenar
    resultados_finales = [
        (doc, puntajes["bm25"] + puntajes["semantic"])
        for doc, puntajes in resultados_combinados.items()
    ]
    resultados_finales = sorted(resultados_finales, key=lambda x: x[1], reverse=True)

    return resultados_finales[:top_resultados]
```

La implementación del retriever y el reranker híbrido mejora en gran medida la calidad de la recuperación de información. La combinación de BM25 con embeddings y la búsqueda en ChromaDB garantizan que la información se recupere de forma eficiente y con una buena precisión.

Base de datos tabular

Su principal objetivo es permitir que el sistema RAG pueda hacer búsquedas eficientes y específicas sobre una tabla de datos, recuperando únicamente la información relevante para responder la consulta del usuario. Esta implementación filtra dinámicamente las filas basadas en las palabras clave contenidas en la consulta.

La función principal del código es **query_dinamica**, que realiza el filtrado dinámico en el dataframe.

```python
def query_dinamica(df: pd.DataFrame, consulta: str) -> pd.DataFrame:
    consulta = consulta.lower()

    # Asegurar que no haya nulos en las columnas relevantes
    df = df.fillna("")

    # Mapeo de consultas a palabras clave
    palabras_clave = {
        "cartas": ["carta", "cartas", "pedido", "visitante", "tarjeta"],
        "recursos": ["uva", "uvas", "vino", "recurso", "bodega", "fichas"],
        "trabajadores": ["trabajador", "trabajadores", "temporales", "fichas"],
        "estructuras": ["estructura", "estructuras", "molino", "campo", "viñedo"]
    }

    # Búsqueda de coincidencias en palabras clave
    filtro_clave = None
    for clave, palabras in palabras_clave.items():
        if any(palabra in consulta for palabra in palabras):
            filtro_clave = clave
            break

    # Filtrado dinámico
    if filtro_clave == "cartas":
        resultados = df[
            df["Componente"].str.contains("carta|pedido|visitante|tarjeta", case=False) |
            df["Descripción"].str.contains("carta|pedido|visitante|tarjeta", case=False)
        ]
    elif filtro_clave == "recursos":
        resultados = df[
            df["Componente"].str.contains("uva|uvas|vino|recurso|bodega|fichas", case=False) |
            df["Descripción"].str.contains("uva|uvas|vino|recurso|bodega|fichas", case=False)
        ]
    elif filtro_clave == "trabajadores":
        resultados = df[
            df["Componente"].str.contains("trabajador|trabajadores|temporales|fichas", case=False) |
            df["Descripción"].str.contains("trabajador|trabajadores|temporales|fichas", case=False)
        ]
    elif filtro_clave == "estructuras":
        resultados = df[
            df["Componente"].str.contains("estructura|molino|campo|viñedo", case=False) |
            df["Descripción"].str.contains("estructura|molino|campo|viñedo", case=False)
        ]
    else:
        resultados = pd.DataFrame()  # No se encontraron coincidencias

    # Si no hay resultados, devolver todo el DataFrame
    if resultados.empty:
        print("No se encontraron coincidencias. Se devuelve toda la base de datos.")
        return df
    else:
        return resultados
```

Para lograr esto, se utiliza un enfoque de mapa de palabras clave. Este mapeo agrupa términos similares en categorías predefinidas, como **cartas**, **recursos**, **trabajadores** y **estructuras**

Una vez categorizada la consulta, se realiza un filtrado condicional sobre las columnas del dataframe, específicamente en las columnas "Componente" y "Descripción". Esta búsqueda permite que se capturen las filas relevantes y trabajar con consultas variadas que puedan usar diferentes palabras para referirse a los mismos conceptos. Por ejemplo, si el usuario menciona "vides" o "viñedos", la función es capaz de reconocer que la consulta está relacionada con estructuras y filtrar las filas correspondientes.

En los casos donde no se identifican coincidencias claras, la función devuelve el dataframe completo, esto con el fin de garantizar que siempre haya una salida válida, aunque no sea ideal, y proporciona una respuesta que puede ser útil en consultas ambiguas.

Ejemplo de uso:

```
ejecutar_query_tabular(df_tabular, "¿Cuáles son las estructuras del juego?")
```

	Componente	Cantidad	Descripción	Usos Principales
6	Tableros Individuales	6	Tableros para cada jugador con su viñedo.	Personalizar estrategias según los tableros in...
10	Estructuras	12	Mejoras que pueden construirse en el viñedo.	Ampliar capacidades del viñedo.
12	Fichas de Residuo	10	Indican las sobras o productos no utilizados e...	Registrar recursos desperdiciados o sobrantes.
14	Fichas de Campo	12	Indican qué campos están ocupados por vides.	Gestionar los campos disponibles para plantaci...
17	Cartas de Construcción	20	Representan edificios y estructuras a construir.	Ayudar a planificar y ejecutar construcciones.

Base de datos de grafos

La consulta dinámica a la base de datos de grafos permite extraer información precisa y contextualizada de un grafo complejo mediante preguntas naturales.
Esta función se centra en interpretar preguntas del usuario relacionadas con los componentes y relaciones del juego Viticulture.
El objetivo es generar y ejecutar consultas válidas utilizando la librería NetworkX.

El proceso se divide en dos etapas principales: la **generación** de la consulta mediante un LLM y la **ejecución** de la misma en el grafo.

Generación de la consulta dinámica
La función **generar_consulta_grafo** recibe como entrada la **pregunta del usuario** y utiliza un **prompt estructurado** para el modelo LLM **Qwen/Qwen2.5-72B-Instruct.**
Este prompt describe detalladamente cómo está compuesto el grafo, los tipos de nodos y relaciones presentes, y las instrucciones precisas para generar una consulta en código Python.

```python
def generar_consulta_grafo(pregunta_usuario: str) -> str:
    """
    Genera una consulta válida en código Python para NetworkX.
    """
    prompt = [
        {"role": "system", "content": (
            "Eres un asistente experto en consultas de grafos utilizando NetworkX.\n"
            "Estás trabajando con un grafo dirigido que modela componentes del juego Viticulture.\n\n"
            "Tipos de nodos:\n"
            "1. 'Vid' (ej. Vid_Tinta, Vid_Blanca)\n"
            "2. 'Uva' (ej. Uva_Tinta, Uva_Blanca)\n"
            "3. 'Vino' (ej. Vino_Tinto, Vino_Blanco)\n"
            "4. 'Pedido' (ej. Pedido_Tinto, Pedido_Blanco)\n"
            "5. 'Campo' (ej. Campo_1, Campo_2)\n"
            "6. 'Estructura' (ej. Bodega, Molino)\n\n"
            "Relaciones del grafo:\n"
            "- 'planta_en': conecta una Vid a un Campo.\n"
            "- 'cosecha_de': conecta una Uva a un Campo.\n"
            "- 'produce': conecta una Uva a un Vino.\n"
            "- 'requiere': conecta un Vino a un Pedido.\n"
            "- 'facilita': conecta una Estructura a un Vino.\n\n"
            "Instrucciones:\n"
            "1. Interpreta preguntas genéricas del usuario.\n"
            "2. Convierte términos como 'vid', 'uva', 'vino', 'pedido' en sus nodos correspondientes.\n"
            "3. Devuelve consultas válidas en Python usando funciones de NetworkX como:\n"
            "   - list(G.successors(node))  # Para obtener nodos sucesores\n"
            "   - list(G.predecessors(node))  # Para obtener nodos predecesores\n\n"
            "Ejemplos:\n"
            "Pregunta: ¿Qué vinos puedo producir?\n"
            "Respuesta:\n"
            "list(G.successors('Uva_Tinta')) + list(G.successors('Uva_Blanca'))\n\n"
            "Pregunta: ¿En qué campos puedo plantar vides?\n"
            "Respuesta:\n"
            "list(G.successors('Vid_Tinta')) + list(G.successors('Vid_Blanca'))\n\n"
            "Pregunta: ¿Qué recursos facilita la Bodega?\n"
            "Respuesta:\n"
            "list(G.successors('Bodega'))\n\n"
            "Si no entiendes la pregunta, devuelve '[]'."
        )},
        {"role": "user", "content": pregunta_usuario}
    ]
```

- El prompt **explica al LLM** cómo mapear términos del usuario como "vid", "uva", "vino" o "bodega" a nodos específicos del grafo.
- Se muestran ejemplos concretos de preguntas y sus respuestas esperadas en formato list(G.successors(node)) o list(G.predecessors(node)). Estas funciones permiten obtener nodos sucesores y predecesores, respectivamente, y son claves en NetworkX para navegar el grafo.
- El modelo devuelve una consulta en Python, como list(G.successors('Vid_Tinta')). Si el LLM no entiende la pregunta, devuelve una lista vacía ([]).

```
try:
    # Llamada al modelo LLM
    response = client_HF.chat.completions.create(
        model="Qwen/Qwen2.5-72B-Instruct",
        messages=prompt,
        max_tokens=100
    )
    consulta = response.choices[0].message.content.strip()

    # Validación básica
    if not re.match(r"^list\(G\.(successors|predecessors)\(.*?\)\)", consulta):
        print("Consulta inválida. Se devuelve una lista vacía.")
        return "[]"

    return consulta
except Exception as e:
    print(f"Error al generar la consulta: {e}")
    return "[]"
```

Ejecución de la consulta en el grafo

Una vez generada la consulta, la función **ejecutar_consulta_grafo** se encarga de ejecutarla directamente sobre el grafo.

```
def ejecutar_consulta_grafo(consulta: str, G):
    """
    Ejecuta una consulta validada en el grafo.
    """
    try:
        resultado = eval(consulta, {"G": G, "list": list})
        return resultado
    except Exception as e:
        print(f"Error al ejecutar la consulta: {e}")
        return []
```

Ejemplo de uso:

```
# Pregunta del usuario
pregunta = "¿Qué vinos puedo producir con uva blanca?"

print("Generando consulta para el grafo...")
consulta_generada = generar_consulta_grafo(pregunta)

if consulta_generada:
    print(f"\nConsulta Generada:\n{consulta_generada}")

    # Ejecutar la consulta en el grafo
    print("\nEjecutando la consulta en el grafo...")
    resultado = ejecutar_consulta_grafo(consulta_generada, G)

    print("\nResultado de la consulta:")
    print(resultado)
else:
    print("No se pudo generar la consulta.")
```

```
Generando consulta para el grafo...

Consulta Generada:
list(G.successors('Uva_Blanca'))

Ejecutando la consulta en el grafo...

Resultado de la consulta:
['Campo_1', 'Campo_2', 'Vino_Blanco']
```

Pipeline Completo

La implementación del **pipeline completo de RAG** integra todas las etapas del sistema: clasificación de consultas, recuperación de información desde diversas fuentes (tabular, grafos y vectorial) y generación de respuestas utilizando un modelo LLM.

Cada paso del proceso está diseñado para cumplir una función específica, y su integración permite resolver consultas complejas de forma dinámica.

Funcionamiento Completo del Pipeline

1. El usuario ingresa una consulta.
2. El sistema clasifica la consulta con clase_LLM para determinar la fuente de datos.
3. Según la clasificación:
    - Se ejecuta una consulta dinámica en la base **tabular.**
    - Se genera y ejecuta una consulta en la base **de grafos.**
    - Se realiza una búsqueda híbrida en la base **vectorial**.
4. La información recuperada se envía al modelo LLM mediante generar_respuesta_llm, que genera la respuesta final.
5. El sistema retorna la respuesta al usuario.

1. **Clasificación de la consulta**
   La primera etapa del pipeline utiliza el clasificador basado en un **LLM** para determinar la fuente más adecuada para recuperar información en función de la consulta del usuario.
    - El modelo clasifica la consulta en TABULAR, GRAFOS o VECTORIAL.
2. **Recuperación de información**
   Dependiendo de la fuente clasificada, se ejecuta una estrategia específica para recuperar información:
    - **Base Tabular**:
      Se ejecuta la función ejecutar_query_tabular.
    - **Base de Grafos**:
      La consulta se transforma dinámicamente en un comando válido de **NetworkX** mediante el modelo LLM y la función generar_consulta_grafo.
        - El prompt proporcionado al modelo le permite entender la estructura del grafo y generar una consulta como list(G.successors(node)).
        - La consulta generada se ejecuta con ejecutar_consulta_grafo, devolviendo resultados precisos sobre las conexiones del grafo.
    - **Base Vectorial**:
      Se realiza una búsqueda híbrida utilizando **BM25** y **búsqueda semántica** a través del reranker reranker_hibrido.
        - BM25 extrae documentos relevantes basándose en palabras clave.
        - La búsqueda semántica utiliza embeddings para encontrar similitudes de significado.
        - Los resultados se combinan y ordenan según un puntaje final.

3. En caso de que la consulta no pueda clasificarse, se devuelve un mensaje indicando que no se pudo determinar la fuente.
4. **Generación de la respuesta**
   Una vez que se ha recuperado el contexto adecuado desde la base de datos correspondiente, la función **generar_respuesta_llm** utiliza el modelo **Qwen/Qwen2.5-72B-Instruct** para generar una respuesta en lenguaje natural.
   - El contexto recuperado se pasa al modelo LLM junto con la consulta original.
   - El prompt está diseñado para que el modelo proporcione respuestas claras y concisas, aprovechando únicamente la información del contexto.

```python
def generar_respuesta_llm(consulta: str, contexto: str) -> str:
    """
    Genera la respuesta final utilizando un modelo LLM con la información recuperada.

    Parámetros:
        - consulta (str): Pregunta original del usuario.
        - contexto (str): Información recuperada de las fuentes.

    Retorna:
        - str: Respuesta generada en lenguaje natural.
    """
    cliente = InferenceClient(api_key="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")

    prompt = [
        {"role": "system", "content": (
            "Eres un asistente experto en el juego Viticulture. Responde preguntas de los usuarios "
            "usando el contexto proporcionado, que contiene información relevante de diferentes fuentes.\n"
            "Proporciona respuestas claras y concisas en lenguaje natural.\n\n"
            "Contexto:\n"
            f"{contexto}"
        )},
        {"role": "user", "content": f"Pregunta: {consulta}\nRespuesta:"}
    ]

    try:
        respuesta = cliente.chat_completion(
            model="Qwen/Qwen2.5-72B-Instruct",
            messages=prompt,
            max_tokens=200,
            temperature=0.5
        )
        return respuesta['choices'][0]['message']['content']
    except Exception as e:
        return f"Error al generar la respuesta: {e}"
```

```python
def pipeline_rag(consulta: str, df_tabular: pd.DataFrame, grafo: nx.DiGraph, chunks: list):

    print(f"\nConsulta del Usuario: {consulta}")

    # 1. Clasificación de la fuente
    fuente = clase_LLM(consulta)
    print(f"Fuente Clasificada: {fuente}")

    # 2. Recuperar información según la fuente
    contexto = ""
    if fuente == "TABULAR":
        print("Consultando en la base TABULAR...")
        resultado_tabular = ejecutar_query_tabular(df_tabular, consulta)
        if not resultado_tabular.empty:
            contexto = resultado_tabular.to_string(index=False)
        else:
            contexto = "No se encontraron resultados relevantes en la base tabular."

    elif fuente == "GRAFOS":
        print("Consultando en la base de GRAFOS...")
        consulta_grafo = generar_consulta_grafo(consulta)
        print(f"Consulta Generada: {consulta_grafo}")
        resultado_grafo = ejecutar_consulta_grafo(consulta_grafo, grafo)
        contexto = f"Resultados del grafo: {resultado_grafo}" if resultado_grafo else "No se encontraron resultados en el grafo."

    elif fuente == "VECTORIAL":
        print("Consultando en la base VECTORIAL...")
        resultados_vectoriales = reranker_hibrido(consulta)
        contexto = "\n".join([f"{i+1}. {res[0]} (Score: {res[1]:.2f})" for i, res in enumerate(resultados_vectoriales)])
    else:
        contexto = "No se pudo clasificar la consulta."

    # 3. Generar la respuesta final con LLM
    respuesta = generar_respuesta_llm(consulta, contexto)
    return respuesta
```

El pipeline completo integra todas las etapas de **clasificación**, **recuperación de datos según el contexto** y **generación** de respuestas**.**

Agente