Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Tecnicatura
Universitaria en Inteligencia Artificial
Procesamiento del Lenguaje Natural

TRABAJO PRÁCTICO N°2 - Año 2024 - 2° Semestre

Antuña, Franco A-4637/1

Viticultura

El juego

En Viticultura, los jugadores se encuentran en el papel de habitantes de la Toscana rústica y premoderna que han heredado unos exiguos viñedos. Disponen de unas pocas parcelas de tierra, una vieja majada, una diminuta bodega y tres trabajadores. Cada uno de ellos sueña con ser el primero en considerar su bodega un verdadero éxito.

Los jugadores se encuentran en la tesitura de determinar cómo quieren distribuir a sus trabajadores a lo largo del año. Cada estación es diferente en un viñedo, por lo que los trabajadores tienen distintas tareas de las que ocuparse en verano y en invierno. Hay competencia por esas tareas, y a menudo el primer trabajador que llega al trabajo tiene ventaja sobre los siguientes.

Afortunadamente para los jugadores, a la gente le encanta visitar las bodegas, y da la casualidad de que muchos de esos visitantes están dispuestos a ayudar en el viñedo cuando lo visitan, siempre y cuando les asignes un trabajador que se ocupe de ellos. Sus visitas (en forma de cartas) son breves pero pueden ser muy útiles.

Con esos trabajadores y visitantes, los jugadores pueden ampliar sus viñedos construyendo estructuras, plantando viñas (cartas de viña) y cumplimentando pedidos de vino (cartas de pedido de vino). Los jugadores trabajan para conseguir el objetivo de dirigir la bodega con más éxito de la Toscana.

Objetivo del proyecto

El proyecto tiene como objetivo desarrollar e implementar dos sistemas de diálogo avanzados: un chatbot basado en modelos de lenguaje de última generación (LLM) y un agente inteligente especializado. Ambos sistemas están diseñados para responder preguntas relacionadas con un juego específico, utilizando exclusivamente información recopilada y estructurada automáticamente por un algoritmo previamente desarrollado. Esto asegura que las respuestas sean precisas, consistentes y basadas únicamente en las bases de datos generadas, sin depender de fuentes externas o no controladas.

Resumen

Se desarrolló un algoritmo capaz de extraer datos de documentos PDF y realizar web scraping mediante Selenium, estructurando la información en bases de datos organizadas y funcionales. Este proceso permitió la automatización de la recopilación y estructuración de datos clave para alimentar los sistemas de diálogo.

Debido a que en el tiempo disponible para el proyecto no se logró implementar el agente inteligente de forma completamente funcional, este presenta problemas intermitentes en su ejecución que afectan la interacción con el usuario. El chatbot basado en modelos de lenguaje demostró poder responder a múltiples preguntas tanto en inglés como en español, aunque tiene algunas dificultades que serán enunciadas más adelante.

Metodología

Herramientas

La selección de herramientas fue un proceso iterativo que implicó evaluar diversas alternativas antes de identificar aquellas que mejor se adaptan a los objetivos del proyecto. Este análisis consideró criterios como la capacidad de las herramientas para manejar grandes volúmenes de datos, su compatibilidad con las tecnologías existentes, y la facilidad de integración en los sistemas desarrollados.

Las herramientas finalmente seleccionadas incluyen:

Selenium

Selenium es una herramienta de automatización de navegadores web que permite realizar web scraping ejecutando el código completo de una página web. Esto incluye la ejecución de scripts *JavaScript*, hojas de estilo *CSS*, y otros componentes dinámicos que no se cargan al analizar solo el *HTML* crudo. A diferencia de *BeautifulSoup*, que procesa el contenido estático de una página, Selenium abre una ventana del navegador mediante *Chromedriver* (u otros drivers compatibles) y permite interactuar con la página como lo haría un usuario, lo que resulta ideal para trabajar con páginas dinámicas o interactivas.

Como desventaja, Selenium requiere de un servidor de *Chromedriver* ejecutándose de fondo, es más lento para extraer información que otras herramientas y requiere más conocimiento de ciertas estructuras, como por ejemplo el uso de *XPath* y selectores *CSS* para la detección de elementos.

PyPDF 2

PyPDF2 es una biblioteca de Python utilizada para leer y extraer contenido de documentos PDF. En este proyecto, se empleó específicamente para obtener texto y metadatos de los reglamentos de los juegos de mesa. Su funcionalidad permite trabajar de manera eficiente con documentos PDF que contenían texto plano, facilitando la estructuración y análisis de los datos extraídos.

Durante el desarrollo, se evaluaron otras bibliotecas, como *pdfreader* y *textract*, pero ninguna logró adaptarse adecuadamente al formato desestructurado de los documentos disponibles. *PyPDF2* fue seleccionada como la solución más simple y efectiva, permitiendo un procesamiento más directo de los archivos sin requerir configuraciones complejas o herramientas externas. Esto la convirtió en una elección óptima para este tipo de tarea, donde la prioridad era la extracción fiable de contenido textual.

Embedding

Para generar representaciones vectoriales del texto extraído, se utilizó el modelo *USE-M* (Universal Sentence Encoder Multilingual), un modelo preentrenado desarrollado por Google y disponible en el hub de *TensorFlow*. Este modelo se eligió por su capacidad de manejar múltiples idiomas, permitiendo que las frases en diferentes lenguajes se conviertan en vectores comparables en un espacio de alta dimensionalidad.

Durante el desarrollo del proyecto, se evaluaron otras alternativas como *SBERT* (*Sentence-BERT*), *MPNet* y *MiniLM*. Sin embargo, cada una presenta desafíos específicos:

- *SBERT* demostró un consumo elevado de recursos, lo que lo hacía poco viable para las limitaciones técnicas del proyecto.
- *MPNet* ofrecía un buen desempeño en calidad de embeddings, pero su implementación resultó complicada y generaba fallas recurrentes.
- *MiniLM* mostró un buen rendimiento en velocidad y eficiencia, pero surgieron problemas técnicos durante su integración, lo que llevó a su descarte.

Finalmente, *USE-M* destacó por su equilibrio entre facilidad de implementación y rendimiento, convirtiéndose en la opción más práctica y efectiva para cumplir con los objetivos del proyecto.

ChromaDB

ChromaDB es una base de datos vectorial diseñada específicamente para gestionar datos en formato de vectores embebidos, como las representaciones numéricas de textos generadas por modelos de *embeddings*. Este tipo de base de datos resulta particularmente útil en tareas de búsqueda semántica y sistemas de *RAG* (*Retrieval-Augmented Generation*).

Una de las características principales de *ChromaDB* es su capacidad para realizar búsquedas por similitud entre vectores, utilizando métricas como la distancia coseno o la distancia euclidiana. Esto permite encontrar textos o conceptos relacionados en un espacio vectorial, optimizando la recuperación de información relevante.

Otro aspecto destacado es el soporte para metadatos asociados a los vectores almacenados. Estos metadatos permiten filtrar y segmentar búsquedas antes de realizar una consulta a un modelo LLM, mejorando la eficiencia del sistema al reducir el número de tokens enviados al modelo. Este enfoque optimiza el flujo de procesamiento, evitando consultas innecesarias y costos adicionales.

RedisGraph

RedisGraph es una extensión de *Redis* diseñada específicamente para trabajar con grafos, permitiendo almacenar y consultar datos estructurados como grafos. Un grafo está compuesto por nodos (entidades) y aristas (relaciones entre entidades), lo que facilita representar relaciones complejas entre diferentes elementos.

Se utiliza una sintaxis basada en *Cypher*, similar a la de bases de datos como *Neo4j*, lo que permite realizar consultas precisas y eficientes sobre los datos estructurados en grafos.

En el contexto del proyecto, se consideró *RedisGraph* como una alternativa a *Neo4j* para manejar datos relacionados, especialmente debido a su integración con *Redis* y su capacidad para gestionar grandes volúmenes de relaciones. Aunque *Neo4j* ofrece una versión en la nube, al ser gratuita, presentaba limitaciones significativas que complicaba el almacenamiento de datos a largo plazo.

Un ejemplo básico de consulta en *RedisGraph* utilizando *Cypher* sería:

$$MATCH\ (p{:}\ Publisher\ \{name{:}\ "Stonemaier\ Games"\})\ <-\ [:Publishes]\ -\ (g{:}\ Game)\ RETURN\ g.title$$

Esto encuentra todos los juegos

$$(g{:}\ Game)\ de\ un\ editor\ (name{:}\ "Stonemaier\ Games").$$

Pandas

Pandas es una librería en Python diseñada para la manipulación y análisis de datos tabulares y estructurados. Es ampliamente utilizada debido a su flexibilidad, rendimiento eficiente y la facilidad para trabajar con grandes volúmenes de datos.

Pandas se integra perfectamente con el ecosistema de Python, lo que permite un flujo de trabajo sencillo y eficiente. Permite la manipulación de datos mediante estructuras como DataFrames, Series y funciones integradas que facilitan la limpieza, transformación y análisis de datos. Ofrece diversas funciones para realizar consultas y filtrados específicos sobre conjuntos de datos tabulares.

LLM: Qwen/QwQ-32B-Preview

El modelo *Qwen/QwQ-32B-Preview* fue seleccionado como solución en el proyecto debido a su capacidad para generar respuestas a partir de interacciones con el usuario. Sin embargo, se presentaron desafíos en su implementación, especialmente en entornos como Colab, donde hubo problemas de compatibilidad y rendimiento.

Las respuestas generadas por este modelo a menudo no cumplían con los requerimientos específicos de los prompts, cómo agregar contenido incorrecto o insuficiente. Además, la velocidad y la consistencia de las respuestas fueron variables, lo que afectó la experiencia del usuario en tareas complejas como la interacción con bases de datos especializadas.

A pesar de estos inconvenientes, Qwen/QwQ-32B-Preview mostró una mejor estabilidad en comparación con otros modelos evaluados, ofreciendo soporte en múltiples idiomas y mejorando la accesibilidad del sistema en contextos multilingües.

BM25

BM25 fue utilizado para realizar búsquedas basadas en palabras clave dentro de los embeddings generados en *ChromaDB*. Este método permite identificar documentos relevantes al calcular una puntuación de similitud entre las consultas del usuario y los vectores almacenados en la base de datos, considerando tanto la frecuencia de las palabras como la dispersión de los términos en los documentos.

Implementación

Armado de BBDD Vectorial.

El armado de la base de datos vectorial tiene como propósito central almacenar y estructurar la información relevante del reglamento y la guía rápida del juego. Este conjunto de datos incluye todas las acciones posibles según la estación, las estructuras disponibles, así como los elementos como figuras y piezas que forman parte del juego, junto con las recomendaciones para su uso.

Originalmente, no se pudo utilizar el reglamento del juego debido a su estructura desordenada, lo que dificulta la lectura y procesamiento adecuado del texto en Python. Este reglamento contenía oraciones mezcladas, desplazamientos en su ubicación y la inclusión de símbolos y palabras incorrectas, lo que afectaba negativamente el análisis. Por lo tanto, se optó por un reglamento en texto plano que permitió una lectura más precisa y accesible.

Asimismo, se intentó la extracción de información desde foros relacionados al juego para crear una amplia base de datos con preguntas, respuestas e interacciones de los usuarios. Sin embargo, este proceso enfrentó serias dificultades debido a la necesidad de una extensa limpieza y preparación de los datos. Además, el tiempo requerido para leer todos los foros resultó excesivo, alcanzando aproximadamente 45 minutos, lo que ocasiona frecuentes desconexiones en entornos como Colab.

Se dividió el texto en secciones a partir de su metadata, de esta manera segmentar la búsqueda.

'OBJECT OF THE GAME',
'GOAL',
'COMPONENTS',
'GAME BOARD SETUP',
'SETUP',
'FIRST ROUND',
'SPRING',
'FALL',
'SUMMER',
'WINTER',
'YEAR END',
'WORKER PLACEMENT',
'VINE DECK VARIETIES AND VALUES',
'GRANDE WORKER PLACEMENT',
'YOKE',
'SCALING THE GAME BASED ON NUMBER OF PLAYERS',
'STRUCTURES',
'HARVESTING FIELDS AND MAKING WINE',

'MAKE WINE',
'RESIDUAL PAYMENTS',
'YEAR',
'VISITOR CARDS',
'FRIENDLY VARIANT'

Como siguiente paso a la extracción, se aplicaron algoritmos de limpieza del texto, por ejemplo reemplazo de símbolos innecesarios (como tildes o caracteres especiales que puedan complicar al LLM), eliminación de espacios y estructuración en chunks.

fill_db

Se implementó la función *fill_db*, esta tiene como propósito procesar datos textuales organizados en un diccionario, generar los embeddings y almacenarlos dentro de *ChromaDB* .

Recibe un diccionario (*secciones_dict*) donde las claves representan las secciones y los valores son el texto en sí. También recibe el modelo de embedding, la colección de *ChromaDB* y un identificador base para los IDs únicos.

```
# Aplicar a las distintas secciones
fill_db(secciones_dict, embed, collection, 'RULEBOOK')
```

Procesa el texto, siendo una combinación en una sola cadena si el contenido es una lista, y se divide el texto en chunks de tamaño fijo para que cada fragmento pueda ser procesado individualmente. Esto se percibe en la estructura cuando los textos son muy largos

```
ID: RULEBOOK_0, Embedding Length: 512, Metadata: {'seccion': 'OBJECT OF THE GAME'}, Chunk: OBJECT OF THE GAME Players begin the game with an ...
ID: RULEBOOK_1, Embedding Length: 512, Metadata: {'seccion': 'OBJECT OF THE GAME'}, Chunk: summer, players place workers on action spaces to ...
ID: RULEBOOK_2, Embedding Length: 512, Metadata: {'seccion': 'GOAL'}, Chunk: GOAL The goal is to reach 20 victory points, with ...
ID: RULEBOOK_3, Embedding Length: 512, Metadata: {'seccion': 'COMPONENTS'}, Chunk: COMPONENTS Cards (118 total) 42 vine cards (green ...
ID: RULEBOOK_4, Embedding Length: 512, Metadata: {'seccion': 'COMPONENTS'}, Chunk: 48 wooden structure tokens (8 unique tokens for ea...
```

En la imagen se percibe como si un chunk es demasiado grande, se separa en varios embeddings con un ID distinto, compartiendo la metadata.

Finalmente, es posible hacer querys directamente a la base de datos, por ejemplo

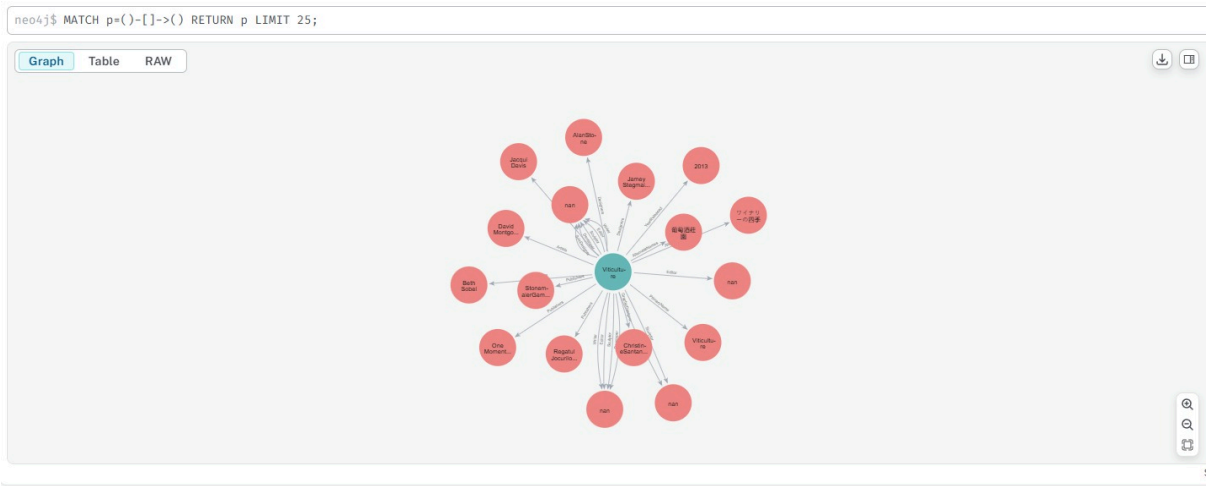
```python
# Ejemplo de uso
query = "STRUCTURE"
results = query_db(query=query, collection=collection, embedding_model=embed, n_results=1)

# Mostrar resultados
for result in results:
    print(f"ID: {result['ID']}")
    print(f"Documento: {result['Documento']}")
    print(f"Distancia: {result['Distancia']}")
    print(f"Fuente: {result['Metadata'].get('source', 'Fuente no disponible')}")
    print(f"Índice: {result['Metadata'].get('index', 'Índice no disponible')}")
    print('-' * 40)
```

```
ID: ['RULEBOOK_35']
Documento: ['STRUCTURESStructures allow you to plant higher quality vines,
Distancia: [1.7046794891357422]
Fuente: Fuente no disponible
Índice: Índice no disponible
----------------------------------------
```

Donde podemos ver que identifica el ID y encuentra dónde está STRUCTURE, sin embargo a veces no devuelve correctamente la Fuente y el Índice.

Armado de BBDD de Grafos

Como se dijo anteriormente, en primera instancia se probó el uso de *Neo4j*, la cual es una solución fácil de implementar y más gráfica, sin embargo traía consigo problemas como que la inactividad del grafo elimina la instancia, y para volver a crearla debe hacerlo el administrador del mismo. Como ejemplo se deja una imagen de una prueba dentro de *Neo4j*



Determinado que era inviable utilizar esta herramienta, se decantó por usar *RedisGraph* que, en definitiva, funciona de manera similar utilizando código en lenguaje *Cypher*.

La base de datos de grafos se utiliza para almacenar datos que poseen una relación entre ellos, pero que no tienen que ver con la jugabilidad per se, sino que representan elementos más informativos como diseñadores, editores, tópicos relacionados, etc. De cada diseñador y artista se almacenaron también otros proyectos y juegos lanzados.

La obtención de la información se hizo exclusivamente a través de *Web Scrapping*, donde dentro de cada dirección se busca la información correspondiente a una “entidad” (por ejemplo, un diseñador) y se almacena en diccionarios.

Armando de BBDD Tabular.

La base tabular es la más simple, se obtienen los datos a partir de *Web Scrapping* sobre las estadísticas del juego.

Como alternativa se intentó obtener los marketplace disponibles, sin embargo estos tenían una complejidad añadida a la hora de hacer la detección de elementos lo cual se dejó anotado como posible mejora a futuro.

Clasificador

Se armaron varias alternativas de clasificadores, variando no solo modelos LLM sino enfoques para el prompt inicial para el rol de "*system*". El enfoque final utiliza como información dos listas, una que contiene la temática o "*class*" a la que se clasifican los prompts del usuario, y otra que, en caso de que la "*class*" sea "*Reglas*" le asigna un índice para luego ser utilizado en la base de datos Vectorial.

Combinándolo con una función "*obtain_database*" se puede encontrar de forma eficiente a qué base de datos debe apuntar el algoritmo

```
[ ]  1 obtain_class("Cuando se publico el juego?")
'Informacion'

[ ]  1 obtain_class("Cuales son las reglas basicas?")
'Reglas SETUP'

[ ]  1 print(obtain_database(obtain_class("Cuando se publico el juego?")))
grafo_bd

[ ]  1 print(obtain_database(obtain_class("Cuantos reviews tiene el juego?")))
tabular_bd
```

Querys dinámicas

Query dinámica de Grafos

Para los grafos, el modelo LLM recibe como system un prompt que le explica como debe armar las query para *RedisGraph*, y se le da como información las entidades y relaciones.

```
chat_prompt = [{
    "role": "system",
    "content": """
    Es importante que solo respondas con la query, no agregues explicaciones ni acotaciones, solo devuelve la query.
    Eres un modelo que genera consultas RedisGraph basadas en frases del usuario.
    La base de datos tiene la siguiente estructura: las entidades son: Game, Publisher, Designer, RelatedTo,
    y las relaciones son: Publishes, Designs, DesignedGame, RelatedTo.
    La estructura de las querys tiene esta forma:
    MATCH (n1:Entity1 {"property": "value"})-[:Relation]->(n2:Entity2)
    RETURN n2.property
    Debes interpretar correctamente la frase del usuario para generar una consulta válida en lenguaje RedisGraph.
    Asegúrate de generar consultas utilizando solo MATCH sin agregar condicionales ni nada similar.
    Es importante que solo respondas con la query, no agregues explicaciones ni acotaciones, solo devuelve la query.

    Ejemplos:
    - Si el usuario escribe: "¿Cuáles son los diseñadores del juego?" deberías devolver:
    "MATCH (g:Game {title: 'Viticulture'})-[:Designs]->(d:Designer) RETURN d.name"
    - Si el usuario escribe: "¿Que otros juegos diseño el diseñador 'Jacqui Davis'?" deberías devolver:
    MATCH (d:Designer {name: 'Jacqui Davis'})-[:DesignedGame]->(g:Game) RETURN g.title
    - Si el usuario escribe: ¿Qué países están relacionados con el juego Viticulture?
    MATCH (g:Game {title: "Viticulture"})-[:RelatedTo]->(r:RelatedTo {name: "'Country'", subject: "'Country'"}) RETURN r.name

    Es importante que generes solo UNA consulta y que sea válida en lenguaje RedisGraph.
    """
}, {
    "role": "user",
    "content": input_text
}]
```

Combinado con una función “*execute_query*” se obtienen estos resultados

Query dinámica vectorial

Lleva a cabo una búsqueda combinada que utiliza tanto métodos tradicionales como semánticos para mejorar la precisión y relevancia de los resultados.

lista_chunks_vec_db accede a los documentos almacenados en la colección *ChromaDB*, permitiendo su posterior procesamiento.

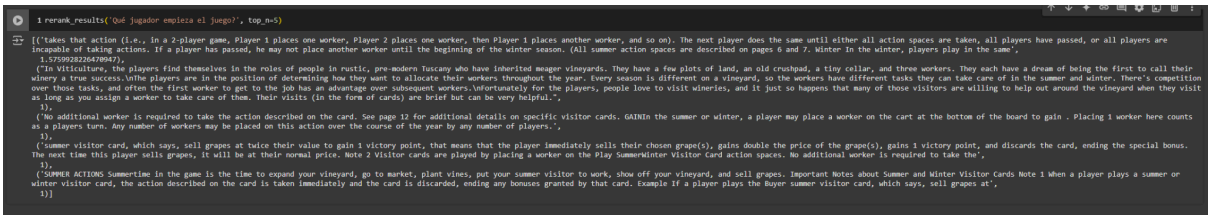
La función *tokenize_spanish* se utiliza para convertir los textos en representaciones *tokenizadas* en español. Esto separa y limpia los textos para facilitar su análisis.

extract_bm_25 implementa la búsqueda utilizando el modelo *BM25* para clasificar documentos basados en palabras clave. Los documentos *tokenizados* se comparan con la consulta *tokenizada*, produciendo puntuaciones para cada documento.

semantic_search utiliza embeddings generados mediante el modelo *USE-M* para representar los textos en vectores. Estos vectores se comparan directamente utilizando *ChromaDB* para obtener similitudes entre documentos.

rerank_results combina los resultados de *BM25* y la búsqueda semántica para mejorar la calidad de los resultados. Los puntajes de *BM25* se normalizan y escalan al rango 0-2 para integrarlos con los puntajes de similitud semántica generados por *ChromaDB*.

Finalmente, los resultados combinados se ordenan en base a su puntuación total, generando una lista *re-rankeada* de los documentos más relevantes.



Query dinámica tabular

En este proceso, el modelo LLM recibe un prompt que incluye los índices específicos del DataFrame, lo que le indica los datos que se deben extraer o consultar. Basado en esta información, el modelo interpreta y genera una consulta adecuada para acceder a los datos necesarios.

El objetivo principal es que el modelo proporciona únicamente una referencia o llamada directa al DataFrame, evitando cualquier mensaje adicional o comentario que no sea relevante para la ejecución de la consulta. Esto garantiza que la respuesta sea directa y específica, enfocándose únicamente en los datos solicitados.

Una vez generada la consulta, se interpreta y extrae los datos deseados del DataFrame.

ChatBot

Se implementó un chatbot interactivo que permite a los usuarios realizar consultas específicas sobre las diferentes bases de datos: vectorial, de grafos o tabular. La función chatbot(prompt) maneja cada tipo de consulta dependiendo del tipo de base de datos identificado.

El flujo principal comienza al recibir una consulta del usuario. Con esta consulta, se determina el tipo de base de datos relevante mediante *obtain_class(prompt)* y *obtain_database(obtain_class(prompt))*. A partir de esta información, el chatbot actúa en consecuencia:

- Para bases de datos vectoriales (*vectorial_bd*), se utiliza la función *rerank_results(prompt)* para procesar los datos y generar resultados relevantes basados en las similitudes de los vectores embebidos.
- Para bases de datos de grafos (*grafo_bd*), se ejecuta una consulta específica a través de *execute_query(obtain_query_graph(prompt))*, que procesa una consulta en formato gráfico y retorna resultados relacionados.
- Para bases de datos tabulares (*tabular_bd*), se construye una consulta directamente basada en el prompt del usuario, utilizando *obtain_query_tabular(prompt)* y evaluando el resultado con *eval(query)*.

Si la base de datos no está relacionada, el chatbot informa al usuario que la consulta no pudo ser procesada.

Después de obtener los resultados de la base de datos correspondiente, estos se procesan mediante la *API* del modelo LLM *Qwen/Qwen2.5-72B-Instruct*, el cual fue el que mejor respondía a los prompts para generar respuestas contextualizadas. La respuesta se limita únicamente a los datos presentados en los mensajes, sin agregar información externa ni contexto adicional.

Además, la función *detect_language(text)* permite detectar automáticamente si el idioma del prompt es español o inglés, adaptando la respuesta según el idioma detectado.

```
if __name__ == "__main__":
    main_interface()
```

```
[CHAT] Bienvenido al Chatbot de Viticultura
[CHAT] Escribe 'Salir' para terminar el programa.
Pregunta: Que tan popular es el juego?

[INFO] Idioma detectado: Español
[CHAT]

El juego tiene 1,255,613 visitas de página, lo que sugiere que es bastante popular.
Pregunta: How popular is the game?

[INFO] Detected language: English
[CHAT]

The game has received 1,255,613 page views, indicating a significant level of popularity.
Pregunta: Salir
[CHAT] Gracias por usar el chatbot. ¡Hasta la próxima!
```

Agente

Lamentablemente, el agente presentaba múltiples falencias en su ejecución, y dado que no se logró obtener un resultado, se plantea como mejora la implementación de este.

Entre las múltiples complicaciones que se tuvieron, lo que más retrasó el proyecto fue que el modelo Ollama tiene algún funcionamiento intermitente dentro del proyecto, lo cual genera que responda de forma aleatoria, o simplemente devuelva un error *errno 99.*

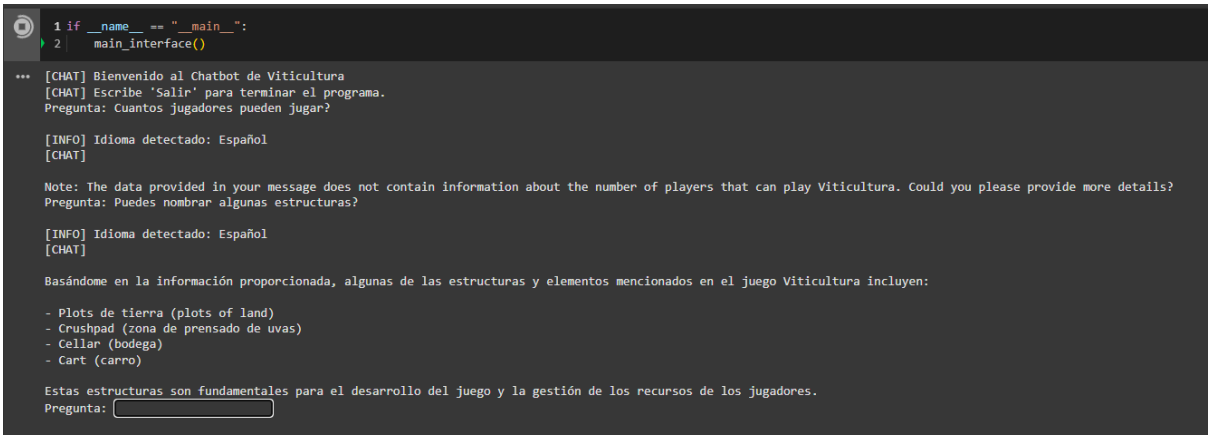
```
WARNING:llama_index.core.agent.react.formatter:ReActChatFormatter.from_context is deprecated, please use `from_defaults` instead.
How can I help you today?

Query: What's the maximum playtime?
> Running step 0b13c1ac-b1ef-4338-afa4-ca0029ebaf7d. Step input: What's the maximum playtime?

Answer: Error processing the query: [Errno 99] Cannot assign requested address
------------------------------------------------------------
```

Resultados y puntos de mejora

En general, se avanzó en el desarrollo e implementación de diversas bases de datos y sus respectivas consultas dinámicas. El chatbot, a pesar de los problemas mencionados, es capaz de responder a muchas preguntas y manejar adecuadamente el contexto de las bases de datos.

Sin embargo, en el caso de la base vectorial, el chatbot presenta algunos problemas relacionados con la velocidad de respuesta, así como dificultades para encontrar ciertos resultados. Por ejemplo,

en ocasiones, los tiempos de respuesta pueden ser más prolongados o simplemente no arrojar resultados deseados debido a restricciones de acceso o inconsistencias en los datos almacenados.



En la imagen se identifican dos principales problemas con el chatbot:

1. El chatbot responde en inglés incluso cuando el prompt del usuario está en español o en otro idioma. Esto ocurre porque la base vectorial está configurada en inglés. Se propone mejorar este aspecto buscando alternativas para que el LLM siempre responda en el idioma del prompt del usuario, independientemente de la base de datos utilizada.
2. En cuanto a la base vectorial, no se obtuvo respuesta para la consulta "Cuántos jugadores pueden jugar?". Tras analizar el rendimiento de la query dinámica vectorial, se concluye que el problema radica en que el proceso de rerank no está funcionando de manera óptima. Se considera necesario implementar mejoras en el rerank para ofrecer respuestas más sólidas.

En el caso de la base tabular, no se presentan mayores inconvenientes y suele brindar respuestas confiables, mientras que para las consultas que acceden a la base de datos de grafos es donde se presenta el mayor desafío.

Se pueden identificar dos principales situaciones problemáticas en el chatbot:

1. El LLM no está comprendiendo correctamente el contexto proporcionado en la base de datos, lo que provoca respuestas vacías en ciertas consultas. Un ejemplo claro es la respuesta a la pregunta “¿Qué juegos diseñó David Montgomery?”, donde se obtuvo un

resultado vacío. Tras un análisis exhaustivo, no se logró identificar una mejora inmediata para abordar esta situación.
2. En ocasiones, la query dinámica para la base de grafos no se arma correctamente, lo que genera errores al ejecutarse y termina interrumpiendo la ejecución. Como posible mejora, se sugiere evitar que el LLM cree directamente la consulta de grafos, y en su lugar, identificar las entidades relevantes para luego construir la query correctamente en función de esas entidades.

El proyecto presenta varios desafíos, pero es suficientemente escalable para incorporar mejoras a medida que se identifican los problemas.

Conclusiones

El desarrollo de un chatbot implica una serie de pasos que se integran de manera progresiva para lograr un funcionamiento eficiente. Es fundamental no solo seleccionar las fuentes de información, sino también estructurarlas de manera que permitan un acceso dinámico, mejorando tanto la calidad como la velocidad de las respuestas, y optimizando el uso de tokens al extraer de manera más efectiva la información relevante.

Dentro de estas estructuras, se pueden derivar las siguientes conclusiones:

Bases de Datos Vectoriales: Son extremadamente útiles para almacenar información que requiere una interpretación más avanzada, especialmente en búsquedas semánticas complejas. Estas bases permiten tanto búsquedas por similitud semántica como por palabras clave, influenciadas por factores como el tamaño de los chunks, los embeddings utilizados y la metadata asignada.

Bases de Datos Tabulares: Son esenciales para manejar datos numéricos, estadísticos o aquellos que no presentan relaciones directas entre sí.

Bases de Datos de Grafos: Resultan imprescindibles para modelar relaciones clave entre diferentes entidades. Facilitan una comprensión directa y clara de estas relaciones al eliminar ambigüedades que pueden surgir de la redacción en formato de oraciones o párrafos, permitiendo respuestas precisas en contextos complejos donde las relaciones entre entidades son esenciales.

Un aspecto fundamental es la necesidad de integrar búsquedas combinadas entre palabras clave y similitud semántica. La búsqueda basada exclusivamente en palabras clave puede ser engañosa, ya que, aunque incluyan términos exactos, no siempre garantizan respuestas adecuadas desde una perspectiva semántica. Asimismo, una búsqueda puramente semántica podría excluir frases relevantes que enriquecerán la respuesta.

El reranking emerge como una solución clave para equilibrar ambas estrategias, asegurando resultados más precisos y contextualizados, aprovechando tanto las fortalezas de las búsquedas basadas en palabras clave como las semánticas, y minimizando sus respectivas limitaciones.

En cuanto a la experiencia con modelos, se ha observado que el chatbot puede ofrecer respuestas rápidas y relevantes siempre y cuando la información proporcionada sea precisa y bien estructurada. No obstante, existen puntos débiles en su funcionamiento que tienden a presentar fallos. Estos problemas surgen principalmente cuando la calidad de los datos almacenados o el procesamiento de las consultas no están optimizados, afectando la consistencia y la precisión de las respuestas generadas.