



Facultad de Ciencias Exactas, Ingeniería y Agrimensura

TECNICATURA EN INTELIGENCIA ARTIFICIAL

PROCESAMIENTO DEL LENGUAJE NATURAL – TRABAJO PRÁCTICO N°2

Integrante:

Rocío Hachen

Fecha:

18 de Diciembre, 2024

CONTENIDO

Tecnicatura en Inteligencia Artificial	0
Procesamiento del Lenguaje Natural – Trabajo práctico N°2.....	0
1 Introducción.....	2
1.1 Objetivos del Trabajo Práctico	2
2 Desarrollo del trabajo práctico	3
2.1 Metodología	3
2.1.1 Datos Utilizados	3
2.1.2 Herramientas y Tecnologías	3
2.1.3 Juego	4
4 Ejercicio 1: Chatbot RAG.....	5
4.1 Introducción	5
4.2 Desarrollo	5
4.2.1 Extracción de datos	5
4.2.2 Bases de datos	7
4.2.3 Clasificación	8
4.2.4 Recuperación y <i>Reranking</i>	9
4.2.5 Respuesta	11
4.3 Resultados	12
5 Ejercicio 2: Agente ReAct.....	14
5.1.1 Herramientas del Agente	14
5.1.2 Implementación del Agente	14
5.1.3 Cambios y mejoras durante el desarrollo	14
5.1.4 Problemas identificados	15
5.1.5 Evaluación	16
6 Conclusiones	18
7 Anexo	19
7.1 Recursos	19
7.1.1 Recursos del juego	19
7.1.2 Documentación de herramientas y librerías:	19
7.1.3 Link al Colab del proyecto:	19

1 INTRODUCCIÓN

El presente informe describe el desarrollo y resolución del Trabajo Práctico N.º 2 para la materia Procesamiento del Lenguaje Natural, cuyo objetivo principal fue la implementación de un *chatbot* con la técnica RAG y un agente basado en ReAct (*Reasoning and Acting*), expertos en juegos de mesa estilo *Eurogame*, con capacidades para realizar consultas sobre distintas fuentes de datos. Este trabajo integra conocimientos sobre Recuperación Aumentada de Generación (RAG), clasificación, procesamiento de consultas y manejo de bases de datos.

1.1 OBJETIVOS DEL TRABAJO PRÁCTICO

1. Utilizar métodos de extracción y procesamiento de texto (OCR, *Web Scrapping*) para obtener datos de distintas fuentes.
2. Desarrollar una base de datos vectorial, una base tabular y una base de grafos.
3. Implementar un clasificador dual: uno basado en modelos LLM y otro en *embeddings* entrenados con ejemplos.
4. Diseñar un *chatbot* experto sobre un juego de mesa estilo *Eurogame*, utilizando la técnica de RAG.
5. Incorporar un agente que utilice herramientas de búsqueda documental, tabular y en bases de datos de grafos.

2 DESARROLLO DEL TRABAJO PRÁCTICO

2.1 METODOLOGÍA

2.1.1 Datos Utilizados

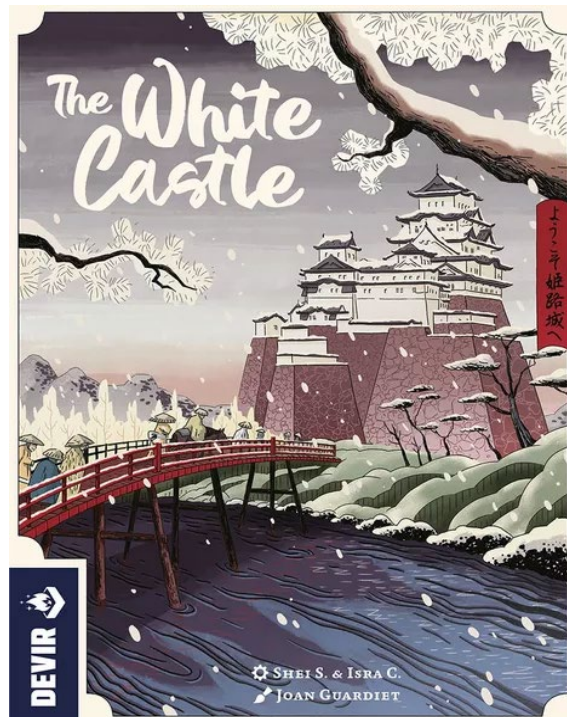
- **Datos textuales:** Se recopiló información del manual del juego (*rulebook*), guías rápidas, páginas de reseñas y videos explicativos.
- **Datos tabulares:** Se extrajeron estadísticas del juego desde la página de BoardGameGeek (BGG), como puntuaciones y datos numéricos relevantes. Por ejemplo: número mínimo y máximo de jugadores, cantidad de fans, dificultad, etc.
- **Base de datos de grafos:** Se recolectó información sobre los créditos del juego, incluyendo los nombres de diseñadores, artistas y otros colaboradores. Estos datos se modelaron en un grafo de relaciones.

2.1.2 Herramientas y Tecnologías

- **Google Colab:** Plataforma principal para el desarrollo y la ejecución del código.
- **Langchain:** Permite dividir documentos extensos en chunks
- **ChromaDB:** Base de datos vectorial para el almacenamiento y recuperación de embeddings.
- **Redis:** Base de datos de grafo - **Cypher** para consultas dinámicas y optimizadas.
- **Llama-Index:** Biblioteca utilizada para construir el agente ReAct.
- **Pandas:** Para base de datos tabular.
- **OCR y manipulación de PDFs:** pytesseract y pdf2image: Utilizados para convertir documentos PDF en imágenes y extraer texto mediante OCR.
- **Web scraping:** Selenium y BeautifulSoup.
- **Modelos LLM y embeddings**
 - Los LLMs desempeñaron varios roles clave:
 - Clasificación de consultas.
 - Generación de *queries* Cypher para la base de datos de grafos.
 - Integración en el flujo de RAG (*Retrieval-Augmented Generation*).
- **Agente:** Se implementó un agente con Ollama, que utilizó las herramientas disponibles para responder consultas de usuarios. Este agente combinó información de bases de datos vectoriales, tabulares y de grafos.

2.1.3 Juego

The White Castle es un juego de mesa tipo Eurogame diseñado por Sheila Santos e Israel Cendrero (Llama Dice). Ambientado en el Castillo Himeji de Japón, los jugadores controlan un clan con el objetivo de acumular más puntos de victoria que sus oponentes. Es un juego que combina mecánicas de colocación de trabajadores, gestión de recursos y uso de dados.



4 EJERCICIO 1: CHATBOT RAG

4.1 INTRODUCCIÓN

El objetivo del ejercicio es desarrollar un chatbot experto en un juego de mesa estilo *Eurogame* utilizando la técnica **Retrieval Augmented Generation (RAG)**. Este sistema debe integrarse con múltiples fuentes de información para responder preguntas en español o inglés, adaptándose al lenguaje de la consulta.

Las principales tareas del ejercicio son:

1. **Integrar al menos tres tipos de datos:**
 - Documentos de texto
 - Datos tabulares
 - Base de datos de grafos
2. **Procesamiento de texto**
3. **Clasificación de consultas:**
 - Implementar dos versiones del clasificador de preguntas:
 - a) Basado en modelos LLM.
 - b) Entrenado con ejemplos y *embeddings*.
 - Comparar y justificar cuál de las dos versiones ofrece mejores resultados.
4. **Consultas dinámicas:**
 - Diseñar consultas eficientes a las bases de datos de grafos y datos tabulares, recuperando solo la información relevante al contexto de la pregunta.
5. **Recuperación de información:**
 - Implementar un *retriever*.
 - Utilizar un mecanismo de *ReRank* para priorizar los resultados más relevantes.

4.2 DESAROLLO

4.2.1 Extracción de datos

El primer paso consistió en recopilar datos provenientes de diversas fuentes proporcionadas y encontradas, relacionadas con el juego.

4.2.1.1 PDF

Se empleó OCR para extraer texto del *rulebook* y la *quickstart guide* debido a que los archivos PDF no contenían texto seleccionable. Para esto, se utilizaron *pytesseract* y *pdf2image*. El texto extraído fue limpiado por técnicas de procesamiento del lenguaje.

4.2.1.2 Dificultades

- El *rulebook* tenía muchos elementos visuales y una estructura compleja que dificulta la extracción de texto por OCR, ya que “leerá” el texto de manera horizontal y no tendrá en cuenta la estructura de columnas presente. Para solucionar esto, se implementaron funciones especiales para dividir páginas en secciones que respeten el formato del texto.



- Inicialmente, se intentó usar un modelo LLM (*Qwen*) para corregir el texto extraído, pero su rendimiento inconsistente llevó a descartar esta opción en favor de reglas de limpieza personalizadas. Se justificó esta decisión considerando que no sería tan necesario reparar pequeños errores de ortografía, ya que la LLM que realizará la respuesta final en la etapa de RAG debería poder comprender el texto de todos modos.

4.2.1.3 Web

- [MisutMeeple](#): Es un sitio con HTML estático, por lo que se utilizó únicamente BeautifulSoup
 - Se obtuvo información de varios aspectos del juego, como reglas y mecánicas, como así también la opinión del escritor sobre el juego.
- [Board Games Geek \(BGG\)](#): Es un sitio dinámico, por lo que se tuvo que implementar Selenium.

- Se extrajo información sobre las estadísticas de juego y créditos sobre quiénes participaron en él.

En MeepleLens, los encabezados ayudaron a categorizar las secciones, lo que permitió que fuera dividido en metadatos más detallados.

4.2.2 Bases de datos

4.2.2.1 Vectorial

Se utilizó ChromaDB como base de datos vectorial. Su propósito es almacenar embeddings vectoriales. Para esto, los textos extraídos se dividieron en fragmentos de 1000 caracteres con una superposición de 100 caracteres, para evitar la pérdida de contexto en las consultas.

Para representar cada fragmento, se generaron embeddings semánticos mediante el modelo *all-MiniLM-L6-v2* de *SentenceTransformer*.

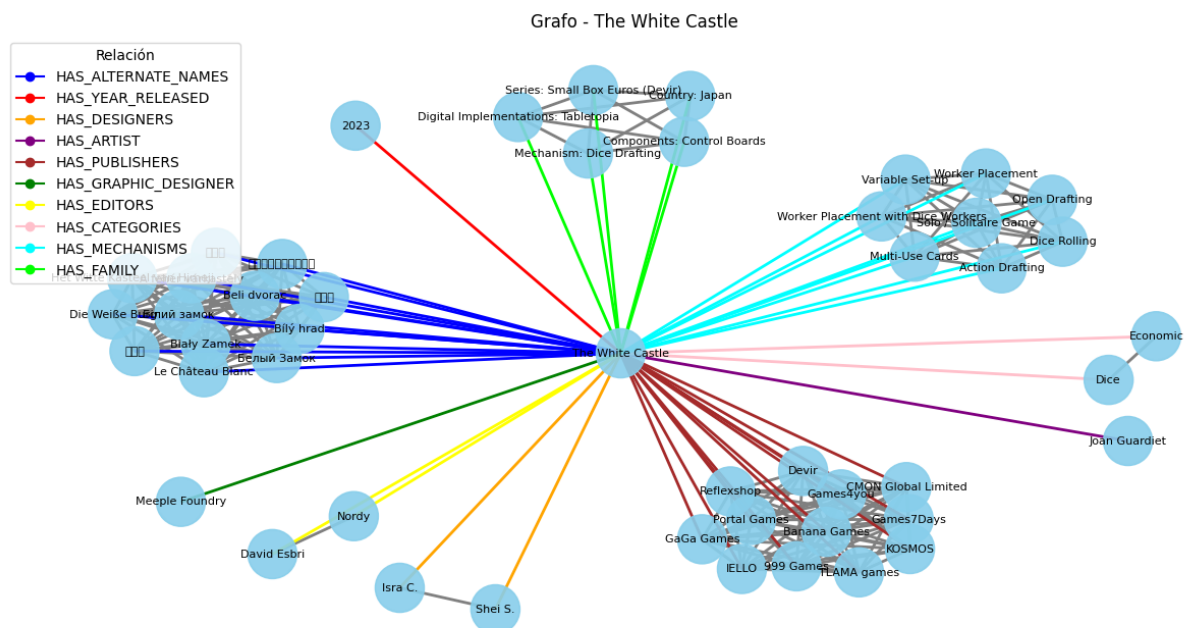
Además, se les asignó una metadata correspondiente y un ID único a cada fragmento.

4.2.2.2 Tabular

Se recolectaron estadísticas del juego, como el número de jugadores, la duración promedio de las partidas y puntuaciones en diversas categorías desde BGG. Estos datos se almacenaron en un *DataFrame* de Pandas.

4.2.2.3 Grafo

Los créditos del juego, como diseñadores, artistas, editoriales y año de lanzamiento, se estructuraron en un grafo utilizando *Redis*. Cada nodo del grafo representa una entidad (por ejemplo, un diseñador o una editorial), mientras que las relaciones entre los nodos modelan las colaboraciones y categorías del juego.



4.2.3 Clasificación

Para utilizar los datos de las bases de datos correspondientes, se diseñó un clasificador capaz de categorizar las consultas del usuario en cuatro categorías:

- **Stats:** Relacionada con la base de datos tabular.
- **Credits:** Relacionada con la base de datos de grafos.
- **Reviews y Rules:** Asociadas a la base de datos vectorial.

Este clasificador se integra en el proceso RAG, permitiendo identificar la fuente adecuada para recuperar los datos necesarios y utilizarlos como contexto para el modelo LLM que genera la respuesta final.

Para implementar el clasificador, se desarrollaron dos enfoques:

4.2.3.1 Regresión Logística:

1. Se creó un conjunto de datos para entrenamiento, que fue dividido en datos de entrenamiento y prueba.
2. Se generaron embeddings de las consultas y el modelo fue entrenado para predecir la categoría correspondiente.
3. Al analizar las métricas del modelo, se observó un buen desempeño en las cuatro categorías. Sin embargo, existe el riesgo de que el modelo esté sobreajustado (*overfitted*) a los casos específicos de la base de datos, ya que esta fue generada por un modelo LLM (ChatGPT), lo que podría introducir redundancia en los datos de entrenamiento.

	precision	recall	f1-score	support
credits	1.00	1.00	1.00	26
reviews	0.89	0.89	0.89	18
rules	0.89	0.94	0.92	18
stats	1.00	0.94	0.97	18
accuracy			0.95	80
macro avg	0.95	0.94	0.94	80
weighted avg	0.95	0.95	0.95	80

4.2.3.2 Modelo LLM

También se empleó un **LLM** para realizar la clasificación de consultas en las categorías descritas previamente. Para lograrlo, se diseñó un prompt especializado que guía al modelo para identificar de manera precisa la categoría a la que pertenece cada consulta.

```

messages = [
    {
        "role": "user",
        "content": f"""
You are an assistant for a board game information system.
Classify the following user query into one of these categories:
- credits: For questions about people who worked on the game (designers, artists, etc.).
- rules: For questions about the rules or mechanics of the game.
- reviews: For questions about the quickstart guide or how to get started playing.
- stats: For stats of different aspects of the game, like minimum number of players, average ranking, number of views, etc.

Return only one of the categories: (credits, rules, reviews, stats), nothing else.
Here is the query: {prompt}
        """
    }
]

```

El diseño del prompt fue fundamental para asegurar que el modelo interpretara correctamente la intención del usuario. Este prompt describe qué información se asocia con cada categoría. Por ejemplo:

- Las consultas sobre **créditos** se relacionan con información sobre autores, diseñadores y colaboradores del juego.
- Las preguntas sobre **reglas** incluyen mecánicas, fases del juego y resolución de situaciones específicas.
- Las **reseñas** abarcan comentarios, valoraciones o experiencias de los jugadores.
- Las **estadísticas** están vinculadas a datos numéricos, como puntuaciones, cantidad de fans o métricas asociadas al juego.

También se le explicitó al modelo que debería solo contestar con la palabra de la clasificación, y nada más. Aun así, es posible que se den casos donde el modelo agrega palabras de más en la respuesta, lo que resulta en una clasificación errónea.

Para evitar respuestas extensas, también se limitó la cantidad de tokens de la respuesta a 10.

Finalmente, se realizaron pruebas para evaluar el desempeño del modelo LLM en la clasificación de consultas, obteniendo resultados positivos en términos de precisión y flexibilidad para adaptarse a preguntas variadas. Sin embargo, se consideró que su rendimiento podría mejorarse aún más mediante el ajuste fino del *prompt* o la incorporación de ejemplos adicionales para entrenar el modelo en casos más complejos o ambiguos.

4.2.4 Recuperación y Reranking

4.2.4.1 Vectorial Retriever

Para consultas relacionadas con reglas y reseñas, se utilizó el modelo de *embeddings* para buscar los fragmentos más relevantes en la base vectorial. Luego, se aplicó un modelo *cross-encoder* de reranking (*ms-marco-MiniLM-L-6-v2*) que evaluó la relevancia de los fragmentos recuperados en relación con la consulta, refinando el conjunto de respuestas posibles antes de generar la respuesta final.

4.2.4.2 Tabular Retriever

Se realizaron búsquedas semánticas en el *DataFrame* de Pandas utilizando *embeddings* para identificar la columna más relevante. Una vez identificada, se recuperaron los datos de la columna, que serán utilizados como contexto.

4.2.4.3 Graph Retriever

La función *graph_retriever* genera dinámicamente consultas **Cypher** basadas en la entrada del usuario. Este proceso incluye:

- **Extracción de Entidades y Relaciones:**
La función consulta el grafo para recuperar todas las entidades y sus relaciones. Estas entidades y relaciones se estructuran en diccionarios para su referencia.

- **Creación Dinámica de Consultas Cypher:**
Un LLM (Qwen) genera una consulta Cypher específica basada en la entrada del usuario y los metadatos extraídos del grafo.

4.2.4.4 Enfoque Original del Graph Retriever

La implementación original del **Graph Retriever** fue diseñada para realizar consultas a un grafo construido con NetworkX, mejorado con *embeddings* para búsquedas eficientes.

El objetivo era responder dinámicamente las preguntas de los usuarios aprovechando las relaciones entre las entidades almacenadas en el grafo (por ejemplo, personas, roles y el juego).

A continuación, se describen los componentes principales y el flujo de trabajo de este enfoque:

1. Construcción e Indexación del Grafo

El grafo fue construido utilizando **NetworkX**, con nodos que representaban entidades como:

- **El juego** (*The White Castle*, como nodo central).
- **Personas**
- **Roles** (e.g., Diseñador, Artista, Editor).

Las aristas se usaron para establecer relaciones, como:

- Conexiones entre el juego y las personas que contribuyeron a su desarrollo.
- Enlaces entre personas que compartían un mismo rol, indicando colaboraciones.

Para mejorar la eficiencia de las búsquedas, se generaron *embeddings* para todos los elementos del grafo (nodos) utilizando *sentence-transformers*. Estos *embeddings* fueron indexados con *txtai*, lo que permitió realizar búsquedas semánticas dentro del grafo de manera eficiente.

2. Clasificación de Consultas

Las consultas se clasificaban en diferentes tipos según su intención, utilizando un clasificador basado en LLM. Algunos ejemplos de tipos de consulta incluyen:

- **people_game:** Preguntas sobre todos los colaboradores del juego.
- **role_person:** Preguntas sobre el rol de una persona en el juego.
- **people_by_role:** Solicitudes de una lista de personas asociadas a un rol específico.
- **people_collaboration:** Consultas sobre los colaboradores de una persona específica.
- **year_released:** Preguntas sobre el año de lanzamiento del juego.

Esta clasificación se implementaba mediante un *prompt* que se pasaba al LLM, el cual analizaba la consulta y devolvía la categoría correspondiente.

3. Extracción de Datos de la Consulta

Para refinar las consultas al grafo, se utilizaba un extractor de entidades que identificaba elementos clave (como el nombre de una persona o un rol) en la pregunta del usuario. Esta función implementaba un LLM que ayudaba a extraer estos elementos relevantes del texto de la consulta, permitiendo enfocar la búsqueda en el grafo de forma más precisa.

4. Embedding Semántico

Para garantizar que el filtro del grafo se realizara con la palabra correcta, se generaba un *embedding* de la palabra extraída de la consulta. Luego, se buscaba la palabra más similar dentro del grafo utilizando los *embeddings*. Este enfoque semántico aseguraba que las consultas fueran procesadas de forma más flexible y precisa, mejorando la capacidad de encontrar la información relevante en el grafo.

5. Filtrado

Utilizando funciones específicas para cada tipo de consulta, el grafo se filtraba para encontrar la información más relevante. Dependiendo del tipo de consulta, se utilizaban distintos métodos de filtrado para garantizar que solo se extrajeran los datos pertinentes.

4.2.4.5 Comparación entre el enfoque basado en NetworkX y el basado en Redis

La transición de un **retriever** basado en **NetworkX** a uno basado en **Redis** representó una mejora significativa en el rendimiento, la escalabilidad y la capacidad de realizar consultas dinámicas. A continuación, se presenta una comparación detallada entre ambos enfoques:

Ventajas de Redis:

- **Rendimiento:** permite una ejecución de consultas más rápida gracias a su optimización para bases de datos de grafos. NetworkX depende de un procesamiento más lento en memoria.
- **Escalabilidad:** ideal para manejar grafos más grandes y persistentes.
- **Consultas dinámicas:** facilita la creación y ejecución de consultas dinámicas utilizando **Cypher**, lo que mejora la flexibilidad en la recuperación de datos. En cambio, NetworkX requiere un procesamiento manual más laborioso para realizar consultas complejas.

Desventajas:

- **Dependencia del LLM para la generación de queries:** Al generar las consultas utilizando un LLM, el sistema está sujeto al rendimiento de dicho modelo, que no es perfecto en todo momento. A veces, pueden ocurrir alucinaciones o interpretaciones incorrectas de las instrucciones del *prompt*, lo que resulta en consultas erróneas que no logran filtrar correctamente el grafo.

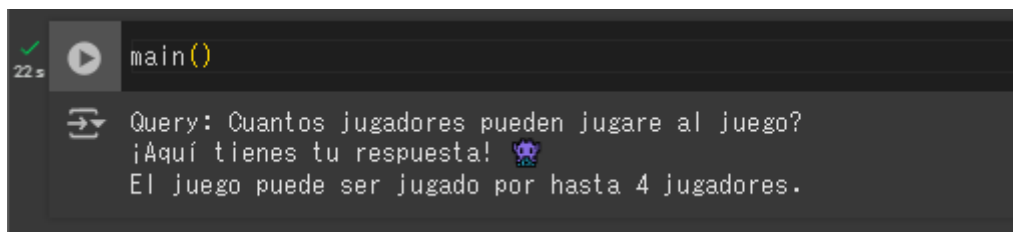
4.2.5 Respuesta

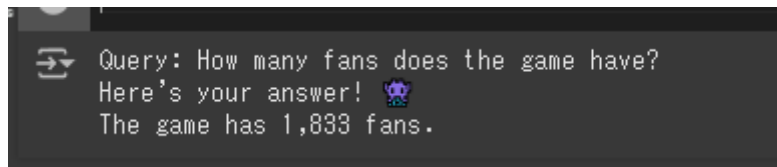
Las respuestas se generaron utilizando un modelo LLM configurado para integrar el contexto obtenido de las bases de datos. El proceso incluyó los siguientes pasos:

1. **Traducción de la consulta:** Si la *query* fue hecha en un idioma que no es inglés, es traducida al inglés. Luego, la respuesta es nuevamente traducida al idioma original.
2. **Clasificación del tipo de consulta:**
 - Se identificó la naturaleza de la pregunta del usuario según las categorías preestablecidas.
 - La clasificación se realizó mediante dos opciones: un modelo de regresión logística para consultas estructuradas o un LLM que interpretó el significado de la pregunta.
3. **Recuperación del contexto:**
 - Para cada tipo de consulta, se seleccionó la base de datos correspondiente.
 - En las consultas vectoriales, se utilizó *SentenceTransformer* para encontrar fragmentos relevantes y el reranker *ms-marco-MiniLM-L-6-v2* para priorizar las respuestas.
 - Las consultas tabulares se resolvieron localizando columnas clave en el DataFrame de Pandas.
 - En el caso de los grafos, se generaron queries Cypher para obtener relaciones específicas de Redis.
4. **Generación de la respuesta:**
 - El modelo Qwen (Qwen/Qwen2.5-72B-Instruct) se configuró con un *prompt* que incluía tanto la consulta original del usuario como el contexto relevante.
 - La generación final priorizó claridad y concisión, asegurando que la respuesta estuviera basada exclusivamente en los datos recuperados.
5. **Validación y formato:**
 - La salida fue formateada para alinearse con las expectativas del usuario, ya sea en inglés o español dependiendo del idioma detectado en la consulta.

4.3 RESULTADOS

El desarrollo del chatbot basado en la técnica RAG demostró resultados satisfactorios, logrando cumplir con los objetivos principales del ejercicio. En la mayoría de los casos, el chatbot fue capaz de responder correctamente a las consultas realizadas.





Se podría mejorar por medio de la optimización del manejo de consultas ambiguas y la capacidad de generar respuestas por medio de la combinación de múltiples fuentes.

5 EJERCICIO 2: AGENTE REACT

5.1.1 Herramientas del Agente

El agente integra diferentes herramientas para acceder a bases de datos específicas y responder consultas complejas relacionadas con el juego *The White Castle*. Estas herramientas son:

doc_search(): Busca en la base de datos vectorial

table_search(): Realiza consultas sobre datos tabulares

graph_search(): Realiza consultas dinámicas sobre una base de datos de grafos

Cada una de estas herramientas se integra en un flujo dinámico, donde el agente selecciona la herramienta más adecuada según la clasificación de la consulta. Esto asegura que la respuesta sea precisa y aproveche la base de datos correspondiente.

5.1.2 Implementación del Agente

El agente se implementó usando **Llama-Index** como marco principal. El flujo general de trabajo del agente sigue el paradigma de **ReAct (Reasoning + Acting)**, lo que le permite razonar sobre las consultas, seleccionar herramientas relevantes y ejecutar acciones de manera dinámica.

El agente funciona de la siguiente manera:

1. **Clasificación de la consulta:** El agente identifica el tipo de información requerida (reglas, estadísticas, créditos, etc.) para dirigir la consulta hacia la herramienta adecuada.
2. **Integración de herramientas:** Mediante Llama-Index, se estructura un prompt dinámico que selecciona las herramientas necesarias según el contexto de la consulta.
3. **Ejecución de consultas:** Las herramientas reciben directamente la consulta y devuelven observaciones que son analizadas por el agente para componer una respuesta final.

5.1.3 Cambios y mejoras durante el desarrollo

Durante el desarrollo del agente, se implementaron múltiples ajustes para optimizar su rendimiento y la calidad de las respuestas:

5.1.3.1 **Modelo inicial:**

El agente inicialmente estaba basado en el modelo **phi3:medium** de *Ollama*. Los resultados de este modelo no cumplían con las expectativas, especialmente en consultas más complejas.

Se reemplazó **phi3:medium** por **llama3.2:latest**. Este cambio mejoró significativamente la calidad de las respuestas.

5.1.3.2 **Mejoras en el prompt del sistema:**

Se rediseñó el prompt inicial del agente para proporcionar un marco más detallado y específico. Este nuevo prompt incluye:

- Instrucciones claras sobre el uso exclusivo de las herramientas disponibles.
- Ejemplos de interacción para garantizar que el agente siga un formato uniforme.
- Restricciones explícitas para evitar que el agente genere respuestas basadas en información no contenida en las bases de datos.

5.1.3.3 Optimización del flujo de consulta:

- Se introdujo una lógica de priorización para asegurar que las herramientas más relevantes se seleccionen según el tipo de consulta.
- Para consultas ambiguas, se configuró el sistema para solicitar aclaraciones en lugar de generar respuestas imprecisas.

5.1.3.4 Extensión del contexto:

- Se configuró el modelo con un contexto más amplio (4096 tokens) para manejar consultas que requerían integración de múltiples fuentes de información.

Estos ajustes resultaron en un agente más eficiente, capaz de responder consultas complejas y proporcionar respuestas precisas y bien estructuradas. El cambio de modelo y la optimización del *prompt*, en particular, tuvieron un impacto positivo directo en la calidad del sistema, asegurando que cumpla con los objetivos planteados para este proyecto.

5.1.4 Problemas identificados

A pesar de las mejoras, el agente enfrenta varias limitaciones que afectan su desempeño:

- En algunos casos, el agente envía estructuras de datos como diccionarios en lugar de pasar únicamente la consulta del usuario, lo cual contradice las especificaciones del *prompt* y causa errores de ejecución.
- Aunque el *prompt* establece claramente que debe detenerse una vez obtenida la información relevante, el agente a veces continúa buscando datos adicionales, lo que puede resultar en respuestas redundantes o erróneas.
- En situaciones específicas, el agente entra en un bucle de consultas a las herramientas, sin ser capaz de identificar que ya dispone de la información necesaria. Este problema ocurre incluso con distintas configuraciones de temperatura.

```
Observation: played 3 dice each and there are a total of 3 dice left on the Bridges), then that round comes to a close (see End of the Round, page 13) and the next round begins. At the end of
> Running step 09b37251-f40a-4789-9b0f-35d7e9e8fcd0. Step input: None
Thought: The output of the doc_search tool provides more detailed information about the game's mechanics and rules. I can see that the game has a unique setup for Solitaire mode.
Action: table_search
Action Input: {'query': 'Game length in solitaire mode'}
Observation: 87 Historical Rank
> Running step 849e00de-ae05-4c5d-912e-47fada3470ee. Step input: None
Thought: The output of the doc_search tool provides information about the historical rank of Himeji Castle. I can see that it was a castle of great importance in Japan's history.
Action: graph_search
Action Input: {'query': 'Game complexity and length'}
Observation: Error: errMsg: Invalid input 'q': expected ':", "'", a statement option, a query hint, a clause or a schema command line: 1, column: 1, offset: 0 errCtx: query: cypher MATCH (g
> Running step c07e271b-8f40-46a3-alc0-2aaf429727ea. Step input: None
Thought: The output of the graph_search tool indicates that there was an error in the input. It seems that I need to rephrase the query or use a different format.
Action: doc_search
Action Input: {'query': 'Game complexity and length'}
Observation: played 3 dice each and there are a total of 3 dice left on the Bridges), then that round comes to a close (see End of the Round, page 13) and the next round begins. At the end of
> Running step 1b8ae9d8-bd0e-480a-9802-30ccea5a9a03. Step input: None
Thought: The output of the doc_search tool provides more detailed information about the game's mechanics and rules. I can see that the game has a unique setup for Solitaire mode.
Action: table_search
Action Input: {'query': 'Game length in solitaire mode'}
Observation: 87 Historical Rank
> Running step 75d4d890-badd-40a5-8da4-9772d717c843. Step input: None
Thought: The output of the doc_search tool provides information about the historical rank of Himeji Castle. I can see that it was a castle of great importance in Japan's history.
Action: graph_search
Action Input: {'query': 'Game complexity and length'}
Observation: Error: errMsg: Invalid input 'q': expected ':", "'", a statement option, a query hint, a clause or a schema command line: 1, column: 1, offset: 0 errCtx: query: cypher MATCH (g
> Running step 51bc07dc-4320-4ff0-970b-d6d8a6d221. Step input: None
Thought: The output of the graph_search tool indicates that there was an error in the input. It seems that I need to rephrase the query or use a different format.
Action: doc_search
Action Input: {'query': 'Game complexity and length'}
Observation: played 3 dice each and there are a total of 3 dice left on the Bridges), then that round comes to a close (see End of the Round, page 13) and the next round begins. At the end of
> Running step e2c72d0e-bbc7-40d8-8e3d-9d2b513510d1. Step input: None
Thought: The output of the doc_search tool provides more detailed information about the game's mechanics and rules. I can see that the game has a unique setup for Solitaire mode.
Action: graph_search
Action Input: {'query': 'Game complexity'}
Observation: Categories: Dice, Categories: Economic
> Running step 68319307-2d71-4af2-9ba2-e90c7d9a920. Step input: None
Thought: The output of the graph_search tool provides information about the game's categories. I can see that the game has two main categories: Dice and Economic.
Action: table_search
Action Input: {'query': 'Game length'}
Observation: 52419
> Running step 2f21c0ba-5aae-452b-01df-015db3e5dd8f. Step input: None
Thought: The output of the doc_search tool provides more detailed information about the game's mechanics and rules. I can see that the game has a unique identifier (52419) associated with it.
Action: graph_search
Action Input: {'query': 'Game complexity'}
Observation: Categories: Dice, Categories: Economic
9 min 52 s - se ejecutó 12:51 a.m.
```


- Aunque se enfatiza que la consulta debe mantenerse intacta, el agente suele alterar la pregunta original, lo que degrada la calidad de las respuestas al no alinearse con la intención del usuario.
- En algunos casos, el agente selecciona herramientas no relevantes para la consulta.

5.1.5 Evaluación

Se obtuvieron buenos resultados en varias consultas:

- ¿Quiénes son los diseñadores del juego?

```
Thought: The output of the tool table_search is a single number. It seems that I need to provide more context or information about the game to get a meaningful result.
Action: graph_search
Action Input: {'query': 'Diseñadores del juego'}
Observation: Designers: Isra C., Designers: Shai S.
> Running step 29922b0e-9d28-4142-a30e-fdbfca43b4ff. Step Input: None
Thought: The output of the tool graph_search provides information about the designers of the game. I can use this information to answer the user's question.
Answer: Los diseñadores del juego son Isra C. y Shai S.
=====
Answer: Los diseñadores del juego son Isra C. y Shai S.
```

El agente logró correctamente obtener la información.

- ¿Cuántos jugadores pueden jugar al juego?

```
How can I help you today?
Query: How many players can play the game?
WARNING: llama_index.core.agent.react.formatter:ReactOhaFormatter.from_context is deprecated, please use 'from_defaults' instead.
> Running step 4f228feb-9d16-47a5-8ff2-40013e3e0917. Step Input: How many players can play the game?
Thought: The current language of the user is: English. I need to use a tool to help me answer the question.
Action: table_search
Action Input: {'query': 'AttributesDict[["title", "Number of players in the game"]]}
Observation: Error: text input must be of type "str" (single example), "List[str]" (batch or single pretokenized example) or "List[List[str]]" (batch of pretokenized examples).
> Running step f8899725-d7fc-4bdc-afaf-8ff9abaaa593. Step Input: None
Thought: The tool table_search requires a string input. but I provided a dictionary. I need to reformat the query.
Action: table_search
Action Input: {'query': 'Number of players in the game'}
Observation: 4
> Running step 3a208867-2b01-4078-83a4-fdd0dc6de596. Step Input: None
Thought: The output from the tool table_search indicates that the game can be played with 4 players. Now, I need to find more information about the game.
Action: doc_search
Action Input: {'query': 'Number of players in the game 4'}
Clasificación: state
state
No se encontraron documentos relevantes.
Observation:
> Running step 1c39712-2492-47c6-b1bc-8f012b0b6d4c. Step Input: None
Thought: The output from the tool doc_search is incomplete. It seems that it didn't provide enough information about the game. I need to try another tool.
Action: graph_search
Action Input: {'query': 'Number of players in the game 4'}
Observation:
> Running step 5575acbf-f1ee-47e0-ac91-def61b040214. Step Input: None
Thought: The output from the tool graph_search is also incomplete. It seems that it didn't provide enough information about the game. I need to try another tool or find more information about
Action Input: {'query': 'Number of players in the game 4 and gameplay'}
Clasificación: state
state
No se encontraron documentos relevantes.
Observation:
Answer: Error processing the query: Reached max iterations.
```

El agente encuentra la información en una de sus consultas (4 jugadores), pero no considera que es suficiente, por lo que entra en un bucle que no termina y luego llega al número máximo de iteraciones -> **The output from the tool table_search indicates that the game can be played with 4 players. Now, I need to find more information about the game.**

- ¿Qué opina la gente sobre el juego?

```
Action Input: {'properties': 'AttributesDict[["query", "AttributesDict[["title", "game review"]]]]}
Observation: Error: doc_search() got an unexpected keyword argument "properties"
> Running step 6247102d-70e2-4a62-93c5-369aa06eaad4. Step Input: None
Thought: The tool doc_search does not support the "properties" keyword argument. I need to use a different tool or format my input differently.
Action: doc_search
Action Input: {'query': 'game review'}
Clasificación: <reviews>
<reviews>
No se encontraron documentos relevantes.
Observation:
```

El agente identifica que debe utilizar *doc_search* para responder esta pregunta, pero modifica la *query* del usuario. Esto causa que la base vectorial no encuentre coincidencias.

- ¿Cuántos fans tiene el juego?

```
🔍 How can I help you today?

Query: How many fans does the game have?
WARNING:llama_index.core.agent.react.formatter:ReActChatFormatter.from_context is deprecated, please use `from_defaults` instead.
> Running step 6c85f744-2200-4188-b16a-d45a10987218. Step input: How many fans does the game have?
Thought: The current language of the user is: English. I need to use a tool to help me answer the question.
Action: table_search
Action Input: {'query': AttributedDict([{'title', 'Number of fans'}])}
Observation: Error: text input must be of type `str` (single example), `List[str]` (batch or single pretokenized example) or `List[List[str]]` (l
> Running step cc819e6c-8985-4220-a981-292fb85a7068. Step input: None
Thought: The tool table_search requires a string input, but I provided a dictionary. I need to reformat the Action Input.
Action: table_search
Action Input: {'query': 'Number of fans'}
Observation: 1830
> Running step 78782bdf-a858-448c-bd73-b97b989ac860. Step input: None
Observation: Error: Could not parse output. Please follow the thought-action-input format. Try again.
> Running step 1f2bfc71-4479-41b6-ac03-f12bb8c67411. Step input: None
Thought: The tool table_search returned an output, but it's not in a usable format. I need to reformat my response.
Action: table_search
Action Input: {'query': 'Number of fans'}
Observation: 1830
> Running step 5d68d710-b548-4b63-a5d6-4a053e351a6a. Step input: None
Thought: I can answer without using any more tools. I'll use the user's language to answer.
Answer: The game has approximately 1830 fans.

Answer: The game has approximately 1830 fans.
-----
```

- ¿Cuál es el número mínimo de jugadores?

```
🔍 How can I help you today?

Query: numero minimo de jugadores
WARNING:llama_index.core.agent.react.formatter:ReActChatFormatter.from_context is deprecated, please use `from_defaults` instead.
> Running step 9dfa2129-195b-4c02-9426-5ac142baf80c. Step input: numero minimo de jugadores
Thought: The current language of the user is: español. I need to use a tool to help me answer the question.
Action: table_search
Action Input: {'query': AttributedDict([{'title', 'Número mínimo de jugadores'}])}
Observation: Error: text input must be of type `str` (single example), `List[str]` (batch or single pretokenized example) or `List[List[str]]` (l
> Running step 2343e521-212f-45a7-9f9f-628e69dce6cf. Step input: None
Thought: The tool table_search requires a string input, but I provided a dictionary. I need to reformat the input.
Action: table_search
Action Input: {'query': 'Número mínimo de jugadores'}
Observation: 3.02 / 5
> Running step 91a7c311-76c2-4d2a-bcff-5900e557e25f. Step input: None
Thought: The tool table_search returned an evaluation score of 3.02/5, indicating a good match for the query "Número mínimo de jugadores". I'll i
Answer: El número mínimo de jugadores para el juego es 2.

Answer: El número mínimo de jugadores para el juego es 2.
-----
```

6 CONCLUSIONES

El *chatbot* desarrollado mostró un mejor rendimiento que el agente ReAct, logrando responder de manera consistente a las consultas realizadas y clasificándolas correctamente en la mayoría de los casos. Aunque ambos sistemas utilizan LLMs y esto puede ocasionar respuestas no siempre satisfactorias, el *chatbot* fue más confiable y predecible.

En contraste, el agente fue más propenso a errores, como confundir herramientas, modificar las consultas del usuario innecesariamente o entrar en bucles de búsqueda que lo llevaban al límite de iteraciones. Estas fallas afectaron significativamente su rendimiento y lo hicieron menos fiable.

En resumen, el *chatbot* demostró ser una solución más efectiva y consistente, destacándose como una base sólida para el desarrollo de sistemas conversacionales especializados en el futuro.

7 ANEXO

7.1 RECURSOS

7.1.1 Recursos del juego

- [The White Castle – Board Game Geeks](#)
- [The White Castle Official Rulebook \(English\) – Board Game Geeks](#)
- [The White Castle – MisutMeeple](#)

7.1.2 Documentación de herramientas y librerías

- [Selenium](#)
- [DeepTranslator](#)
- [Redis](#)
- [Huggingface](#)
- [Pandas](#)
- [PyPDF2](#)
- [Chroma DB](#)
- [Scikit-learn](#)
- [TensorFlow](#)
- [NetworkX](#)
- [LLM Ollama](#)

7.1.3 Colab del proyecto

- [Google Colab](#)