Sistemas de Recuperación de Información y de Recomendación - 2024-2025

Universidad de Granada

Agente potenciado por sistema RAG

Miguel García López

Índice

Índice de figuras

Índice de cuadros

1. Objetivos

Este documento tiene como objetivo principal explicar de forma clara y suficientemente detallada la implementación de un agente conversacional potenciado con un sistema *RAG* (*Retrieval-Augmented Generation*).

Se procederá a una explicación detallada del proceso de diseño, así como tecnologías usadas, patrones de diseño de clases y tipos de métodos usados.

También se explicarán ciertos conceptos teóricos, como qué es un agente y qué es un *RAG*.

2. Introducción

En este trabajo se detallan y explican los procesos llevados a cabo en la creación de un agente conversacional potenciado con un sistema *RAG*, que es fundamentalmente, un sistema muy potente de recuperación de información.

Este tipo de técnicas han sido usadas desde hace muchos años atrás en el tiempo. Antes de la invención formal de los sistemas *RAG* (*Retrieval-Augmented Generation*), existieron numerosos sistemas y enfoques que sentaron las bases conceptuales para esta tecnología. Por ejemplo, en [1] se establecieron las bases de los sistemas modernos de recuperación de información, introduciendo conceptos como el modelo de espacio vectorial que más tarde serían fundamentales para *RAG*.

3. Tecnologías Utilizadas

El sistema conversacional con capacidades *RAG* desarrollado en este trabajo se apoya en un conjunto de tecnologías avanzadas en inteligencia artificial y procesamiento de lenguaje natural. Para la recuperación eficiente de información se utiliza *FAISS* [2], mientras que *Sentence-Transformers* permite generar representaciones semánticas de consultas y documentos. La generación de respuestas se basa en los modelos de lenguaje de *OpenAI*, integrados mediante técnicas de *prompt engineering*. *LangGraph* facilita la orquestación de los componentes mediante estructuras de flujo conversacional. *NumPy* se encarga del manejo eficiente de datos vectoriales, y *Streamlit* proporciona una interfaz web interactiva que permite al usuario visualizar y entender el funcionamiento del sistema.

4. Teoría

En esta sección se presentan los fundamentos teóricos de las dos tecnologías principales empleadas en este trabajo: los sistemas de Generación Aumentada por Recuperación (*RAG*) y los agentes basados en modelos de lenguaje. Ambos representan avances significativos en el campo de la inteligencia artificial aplicada al procesamiento del lenguaje natural y la creación de sistemas conversacionales inteligentes.

4.1. Generación Aumentada por Recuperación (RAG)

La Generación Aumentada por Recuperación (*Retrieval-Augmented Generation* o *RAG*) [3] es un paradigma que combina las capacidades generativas de los modelos de lenguaje de gran escala (*Large Language Models* o *LLMs*) con sistemas de recuperación de información para producir respuestas más precisas, actualizadas y verificables.

4.1.1. Fundamentos del RAG

Un sistema RAG opera fundamentalmente en dos fases principales:

1. **Fase de Recuperación (*Retrieval*)**: Dada una consulta del usuario, el sistema busca y recupera documentos o fragmentos de información relevantes de una base de conocimiento.
2. **Fase de Generación (*Generation*)**: Los documentos recuperados se incorporan como contexto adicional al *prompt* del modelo de lenguaje, que luego genera una respuesta basada tanto en su conocimiento parametrizado como en la información recuperada.

Esta arquitectura híbrida resuelve varias limitaciones inherentes a los LLMs puros:

- **Conocimiento estático**: Los modelos de lenguaje tienen conocimientos limitados al período hasta su entrenamiento, mientras que RAG permite acceso a información actualizada.
- **Alucinaciones**: Al anclar la generación en fuentes externas verificables, RAG reduce significativamente la tendencia de los LLMs a generar información incorrecta.
- **Transparencia**: RAG puede proporcionar las fuentes de su información, permitiendo verificar la procedencia de las respuestas.
- **Especialización**: Permite adaptar modelos generales a dominios específicos sin necesidad de reentrenamiento.

4.1.2. Estrategias de Fragmentación (*Chunking*)

Un componente crítico en el diseño de sistemas RAG es la estrategia de fragmentación (*chunking*) de los documentos de la base de conocimiento. Entre las principales estrategias se encuentran:

- **Fragmentación por tamaño fijo**: Divide los documentos en fragmentos de un número determinado de tokens o caracteres. Simple pero puede romper la coherencia del texto.
- **Fragmentación por límites semánticos**: Algoritmos más sofisticados que dividen el texto basándose en cambios temáticos o estructuras semánticas.
- **Fragmentación recursiva (*recursive chunking*)**: Crea una jerarquía de fragmentos de diferentes granularidades, permitiendo representaciones multi-nivel del conocimiento.

La elección de la estrategia de fragmentación afecta directamente a la calidad de la recuperación, ya que fragmentos demasiado grandes pueden contener información irrelevante, mientras que fragmentos demasiado pequeños pueden perder contexto importante.

4.1.3. Estrategias de Recuperación (*Retrievers*)

El componente de recuperación (*retriever*) de un sistema *RAG* es el encargado de identificar y seleccionar los fragmentos más relevantes para una consulta dada. Las principales tecnologías incluyen:

- **Recuperación basada en *embeddings* densos**: Utiliza representaciones vectoriales densas del texto para capturar la semántica profunda. Tecnologías como modelos basados en *Sentence-Transformers* generan estas representaciones.
- **Recuperación léxica**: Basada en la coincidencia de términos, usando algoritmos como *BM25* [4] o *TF-IDF*. Más simple pero eficaz para coincidencias exactas.
- **Recuperación híbrida**: Combina métodos semánticos y léxicos para aprovechar las fortalezas de ambos enfoques, como en arquitecturas de tipo *ColBERT* o *REALM*.

4.2. Agentes

Un agente basado en modelos de lenguaje puede definirse como un sistema que combina un *LLM* con capacidades de acción mediante herramientas (*tools*), permitiéndole realizar tareas concretas en entornos específicos.

5. Implementación

5.1. Patrones de Diseño

En la implementación del sistema se han utilizado varios patrones de diseño para garantizar modularidad, escalabilidad y facilidad de mantenimiento:

- **Patrón Estrategia (*Strategy*)**: Implementado en el módulo de recuperación para permitir la selección dinámica entre diferentes estrategias de recuperación (*embeddings* densos, léxica o híbrida). Se ha utilizado para modelar distintos tipos de técnicas de *chunking* y de recuperación por similitud.
- **Patrón Singleton**: Utilizado para garantizar una única instancia del componente encargado de gestionar el acceso compartido a los recursos del sistema, asegurando una coordinación coherente entre los módulos. Se ha utilizado en la clase `VectorDB`, que representa la base de datos vectorial.

5.2. Clases y Componentes

El sistema se compone de dos bloques principales: los **Chunkers** (figura 1), encargados de dividir el texto en fragmentos, y los **Retrievers** (figura 2), responsables de recuperar la información más relevante dado un *input*. Estos dos bloques principales encajan dentro de una clase que modele el comportamiento y funcionalidades principales de una base de datos vectorial, de lo cual se encarga la clase `VectorDB` (figura 3).

5.2.1. Chunkers

FixedSizeChunker Esta clase implementa una estrategia de fragmentación de texto en bloques de longitud fija, permitiendo opcionalmente solapamiento entre ellos.

- **Constructor**: Recibe el tamaño máximo del bloque (`max_length`) y un solapamiento opcional (`overlap`).
- **chunk()**: Divide el texto en fragmentos consecutivos, asegurando que cada fragmento no exceda el tamaño definido y respeta el solapamiento.

RecursiveChunker Fragmenta el texto de forma jerárquica: por párrafos, luego oraciones, y finalmente palabras, dependiendo del nivel de recursión. Se asegura de que cada fragmento cumpla con una longitud máxima.

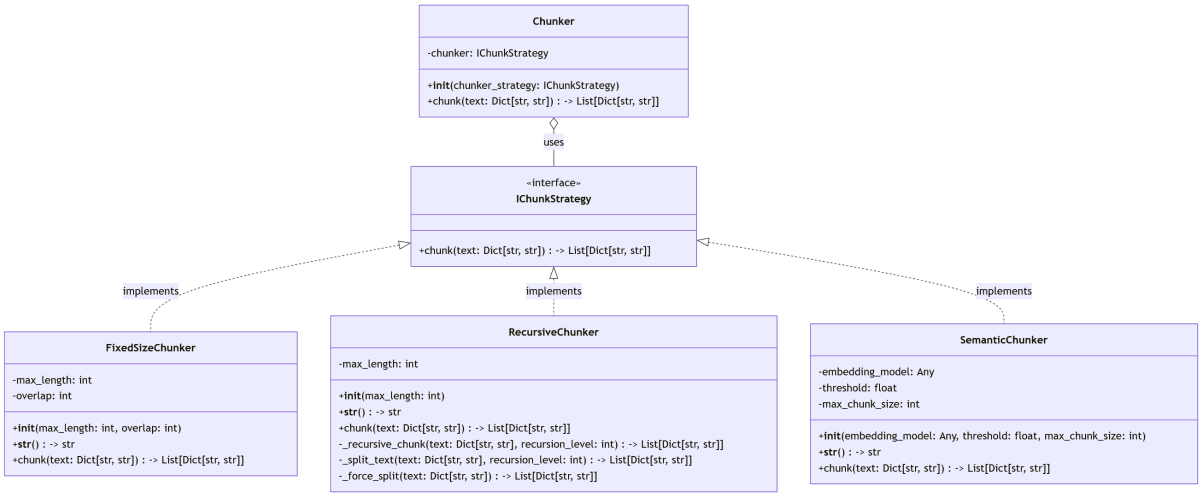

Figura 1: Diagrama de clase *Chunker*

- **chunk()**: Llama al método recursivo `_recursive_chunk()`, que adapta el criterio de fragmentación según el nivel de profundidad.
- **_split_text()**: Fragmenta el texto por saltos dobles de línea, por oraciones o por palabras, dependiendo del nivel de recursión.
- **_force_split()**: En caso de que no se pueda dividir el texto de forma semántica, lo parte en trozos estrictamente según el límite de longitud.

SemanticChunker Esta clase utiliza *embeddings* para crear fragmentos semánticamente coherentes. Agrupa frases consecutivas en un bloque mientras su similitud coseno permanezca por encima de un umbral definido.

- **Constructor**: Requiere un modelo de *embeddings*, un umbral de similitud y una longitud máxima del fragmento.
- **chunk()**: Calcula los *embeddings* de cada oración y agrupa aquellas cuya similitud sea alta y cuya longitud combinada no exceda el máximo.

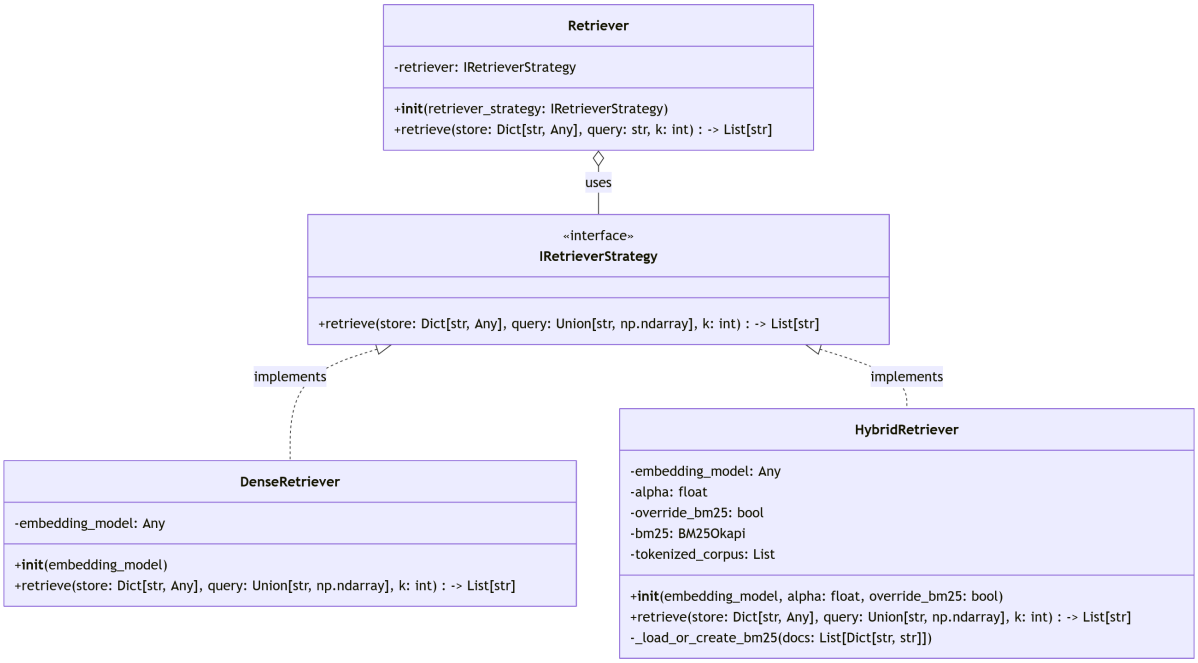

Figura 2: Diagrama de clase *Retriever*

5.2.2. Retrievers

DenseRetriever Implementa una estrategia de recuperación basada en similitud de vectores densos generados por modelos de lenguaje.

- **retrieve()**: Recupera los `k` documentos más similares a una consulta dada, utilizando *embeddings* normalizados y una estructura de índice vectorial.

HybridRetriever Combina la recuperación densa con un modelo *BM25* de recuperación basada en palabras clave, ponderando ambas con un factor α.

- **Constructor**: Inicializa con un modelo de *embeddings*, un valor α para ponderar las dos fuentes de puntuación, y una bandera para sobreescribir el modelo *BM25* guardado.
- **_load_or_create_bm25()**: Carga un modelo *BM25* desde disco si está disponible o lo crea desde el corpus.
- **retrieve()**: Calcula puntuaciones densas y dispersas, normaliza ambas y las combina según α para recuperar los documentos más relevantes.

5.2.3. VectorDB

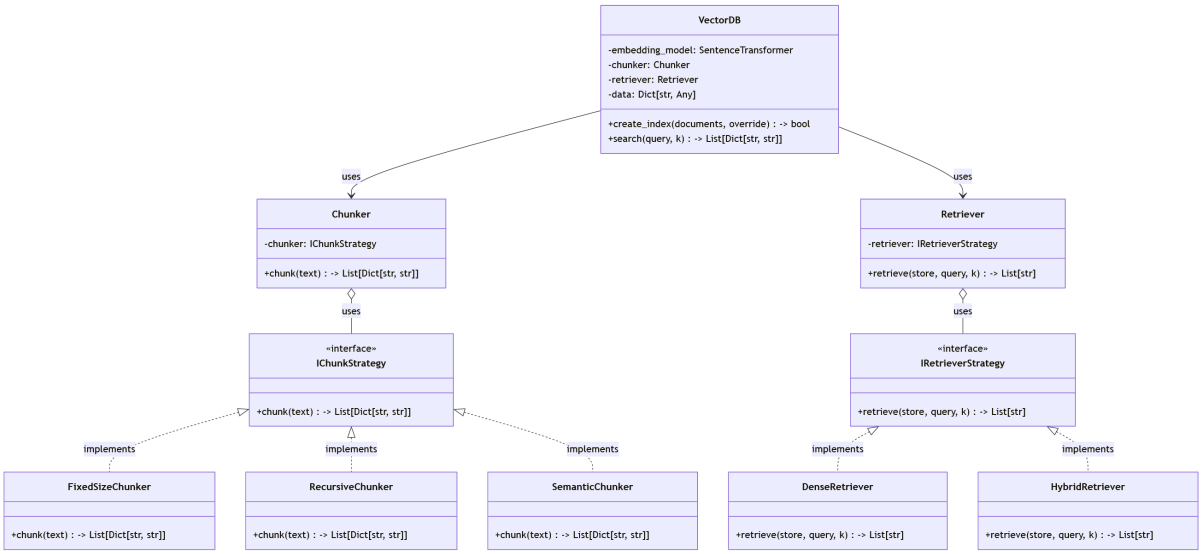

Figura 3: Diagrama de clase *VectorDB*

La clase `VectorDB` implementa una base de datos vectorial que permite gestionar documentos mediante *embeddings* generados con modelos de `SentenceTransformer`. Esta clase es responsable de la creación de un índice de búsqueda eficiente utilizando `FAISS` y proporciona métodos para almacenar, cargar y realizar búsquedas sobre los documentos indexados.

Constructor El constructor de `VectorDB` recibe varios parámetros. El primero de ellos es `embedding_model`, que puede ser un nombre de modelo o un objeto `Sentence-Transformer`. Este modelo se utiliza para generar los *embeddings* de los documentos. También recibe un `chunker`, que es el responsable de dividir los documentos en fragmentos manejables, y un `retriever`, que se utiliza para la búsqueda de documentos relevantes. Además, se puede especificar la ruta de almacenamiento de los datos con el parámetro `path`.

Métodos `VectorDB` incluye varios métodos importantes:

- **_create_dir(path)**: Este método intenta crear el directorio en la ruta especificada si no existe.

- **_generate_embeddings(documents)**: Genera los embeddings para una lista de documentos. Para ello, primero divide los documentos en fragmentos utilizando el `chunker` y luego calcula los embeddings de cada fragmento.
- **_save(index, documents)**: Guarda el índice de `FAISS` y los documentos en la ruta de almacenamiento especificada.
- **_load()**: Carga los datos previamente almacenados desde un archivo.
- **create_index(documents, override)**: Crea un índice de `FAISS` a partir de los embeddings generados. Si el índice ya existe y `override` es `False`, simplemente carga el índice existente.
- **search(query, k)**: Realiza una búsqueda de los `k` documentos más relevantes con respecto a la consulta proporcionada utilizando el `retriever`.

6. Chatbot

El agente creado con *LangGraph* utiliza la herramienta del *RAG*, de forma que esta será usada solo cuando convenga al agente. El *workflow* seguido es el que se observa en la figura 4

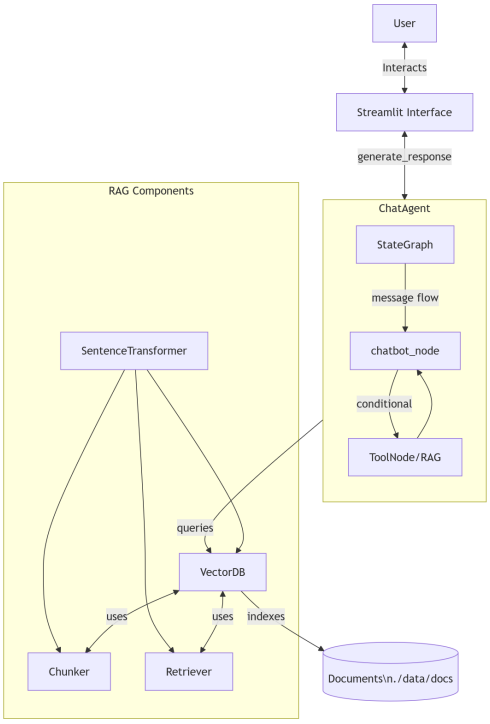

Figura 4: Diagrama de flujo del sistema del agente conversacional con *RAG*

6.1. Conversaciones

Se han utilizado datos de los apuntes generados por el estudiante en formato *Markdown* durante el transcurso de sus estudios en Ingeniería Informática y en el Máster de Ciencia de Datos. El agente tiene instrucciones de usar el *RAG* cuando se le pregunten cosas del ámbito técnico informático.

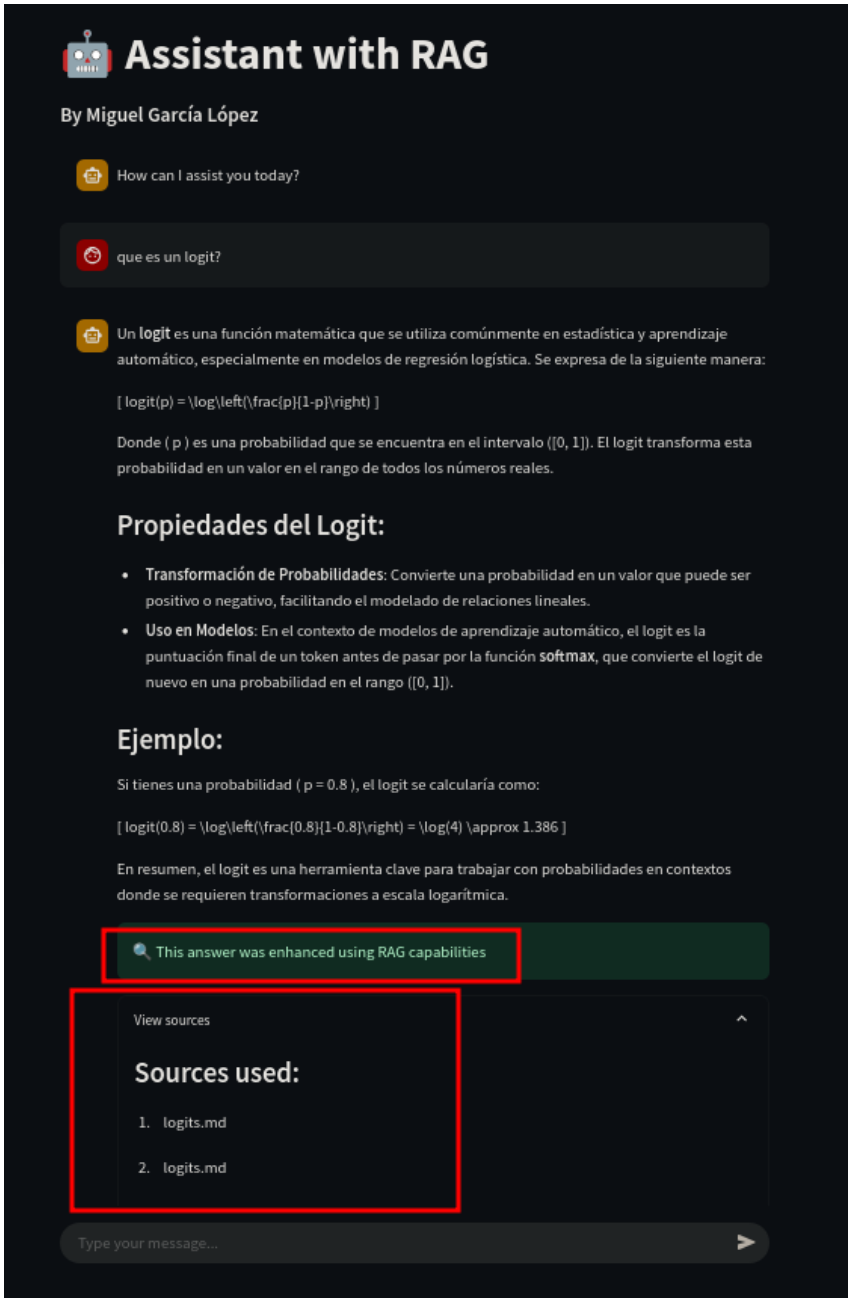

Figura 5: Ejemplo de pregunta técnica en la que se usa el *RAG*

Referencias

[1] G. Salton y M. J. McGill, *Introduction to Modern Information Retrieval*. McGraw-Hill Book Company, 1983.

[2] J. Johnson, M. Douze y H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, n.º 3, págs. 535-547, 2019.

[3] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, págs. 9459-9474, 2020.

[4] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu y M. Gatford, "Okapi at TREC-3," *NIST SPECIAL PUBLICATION SP*, vol. 109, pág. 109, 1995.