Escuela Técnica Superior de Ingeniería
Universidad de Huelva

Grado en Ingeniería Informática

Trabajo Fin de Grado

Estudio y construcción de una interfaz conversacional mediante Generación Aumentada por Recuperación (RAG) para mejora de extracción de información sobre una enfermedad rara (síndrome de Rett)

Alejandro Pérez Domínguez
Huelva, Julio 2025

1 Resumen

La popularización de los Modelos de Lenguaje Grandes (LLM) y, últimamente, de Modelos de Lenguaje Pequeños (SLM, por sus siglas en inglés), están facilitando soluciones a un viejo desafío en el campo de la inteligencia artificial: Enseñar a los ordenadores a entender cómo hablan y escriben los humanos.

Los modelos grandes del lenguaje o *Large Language Models* son modelos de inteligencia artificial que se entrenan utilizando algoritmos de *Deep Learning* sobre conjuntos enormes de información generada por humanos que son capaces de ofrecernos respuestas generales y con un patrón muy humano a las preguntas que se les hacen.

Sin embargo, si buscamos respuestas precisas en un contexto determinado o en un periodo muy reciente, posterior al entrenamiento del modelo, los LLM por sí solos no proporcionarán respuestas específicas o habrá una alta probabilidad de que alucinen y se inventen completamente la respuesta. Las técnicas de Generación Aumentada por Recuperación o (RAG, por sus siglas en inglés, *Retrieval Augmented Generation*) se han desarrollado para mejorar la calidad de las respuestas en contextos específicos, ofreciendo respuestas más precisas y actualizadas a, por ejemplo, una disciplina concreta o en base a repositorios de conocimiento privados.

Este trabajo pretende estudiar qué es un RAG, cuáles son sus componentes principales y construir un sistema conversacional en un contexto muy especializado con base a información muy reciente. Se ha optado por aplicar las técnicas de RAG al contexto de una enfermedad rara, el síndrome de Rett.

Por otro lado, un objetivo secundario del trabajo será evaluar cuantitativamente los resultados del sistema RAG y compararlos con un LLM para corroborar la hipótesis de que un RAG es más eficaz que un LLM.

Palabras clave

Modelos de lenguaje, inteligencia artificial, RAG, síndrome de Rett, *embeddings*, *prompt*.

2 Abstract

The popularization of Large Language Models (LLMs), and more recently, Small Language Models (SLMs), is helping to address a long-standing challenge in the field of artificial intelligence: teaching computers to understand how humans speak and write.

LLMs, or Large Language Models, are artificial intelligence systems trained using deep learning algorithms on massive datasets of human-generated content. These models are capable of providing general responses that follow very human-like patterns when answering questions.

However, when precise answers are needed within a specific context or concerning very recent information that falls outside the model's training data, LLMs alone tend to be insufficient. They may fail to provide accurate responses or are likely to "hallucinate" and generate completely fabricated answers. Retrieval-Augmented Generation (RAG) techniques have been developed to improve response quality in domain-specific contexts, delivering more accurate and up-to-date answers—such as in a particular discipline or based on private knowledge repositories.

This work aims to study what a RAG system is, identify its main components, and build a conversational system within a highly specialized context based on very recent information. The chosen application domain for the RAG system is a rare disease: Rett syndrome.

Additionally, a secondary objective of this study is to quantitatively evaluate the system's performance and compare it to a standalone LLM, in order to support the hypothesis that a RAG system is more effective than an LLM in such contexts.

Keywords

Language Models, artificial intelligence, RAG, Rett syndrome, embeddings, prompt.

3 Tabla de contenidos

4 Índice de figuras

5 Introducción

La inteligencia artificial vive una era de transformación sin precedentes, impulsada en gran medida por la irrupción de los Modelos de Lenguaje Grandes (LLM). Estos modelos, y sus derivados más ligeros, los Modelos de Lenguaje Pequeños (SLM), están aportando soluciones efectivas a uno de los desafíos de la computación: la capacidad de las máquinas para comprender y generar lenguaje humano de forma natural. Entrenados con algoritmos de *Deep Learning* sobre grandes conjuntos de datos, los LLM son capaces de ofrecer respuestas generales con una coherencia y un patrón notablemente humanos.

Sin embargo, a pesar de su versatilidad, estos modelos adolecen de limitaciones importantes. Cuando se enfrentan a dominios de conocimiento muy especializados o a información generada tras su fecha de entrenamiento, su fiabilidad disminuye drásticamente. En estos escenarios, los LLM tienden a proporcionar respuestas imprecisas o, lo que es más problemático, a "alucinar", inventando información que, aunque pueda parecer verosímil, es incorrecta. Esta deficiencia representa una barrera fundamental para su adopción en contextos donde la exactitud y la veracidad son innegociables.

Para solventar estas limitaciones, surge la arquitectura de Generación Aumentada por Recuperación (RAG, por sus siglas en inglés). Esta técnica enriquece a los modelos de lenguaje al aportarles contexto obtenidos de bases de conocimiento externas, actualizadas y específicas. Al complementar las respuestas del modelo en datos recuperados de una fuente fiable, RAG mitiga el riesgo de alucinaciones y aumenta la precisión, relevancia y fiabilidad de la información generada, especialmente en nichos de conocimiento o al usar repositorios de datos privados.

El presente trabajo se adentra en el estudio y la aplicación práctica de esta tecnología. Como objetivo principal, se aborda la construcción de un sistema conversacional especializado en una enfermedad rara, el síndrome de Rett, utilizando un corpus de conocimiento biomédico muy reciente. Además, como objetivo secundario, se busca corroborar empíricamente la hipótesis de que un sistema RAG es cuantitativamente más preciso que un LLM estándar en un dominio tan acotado.

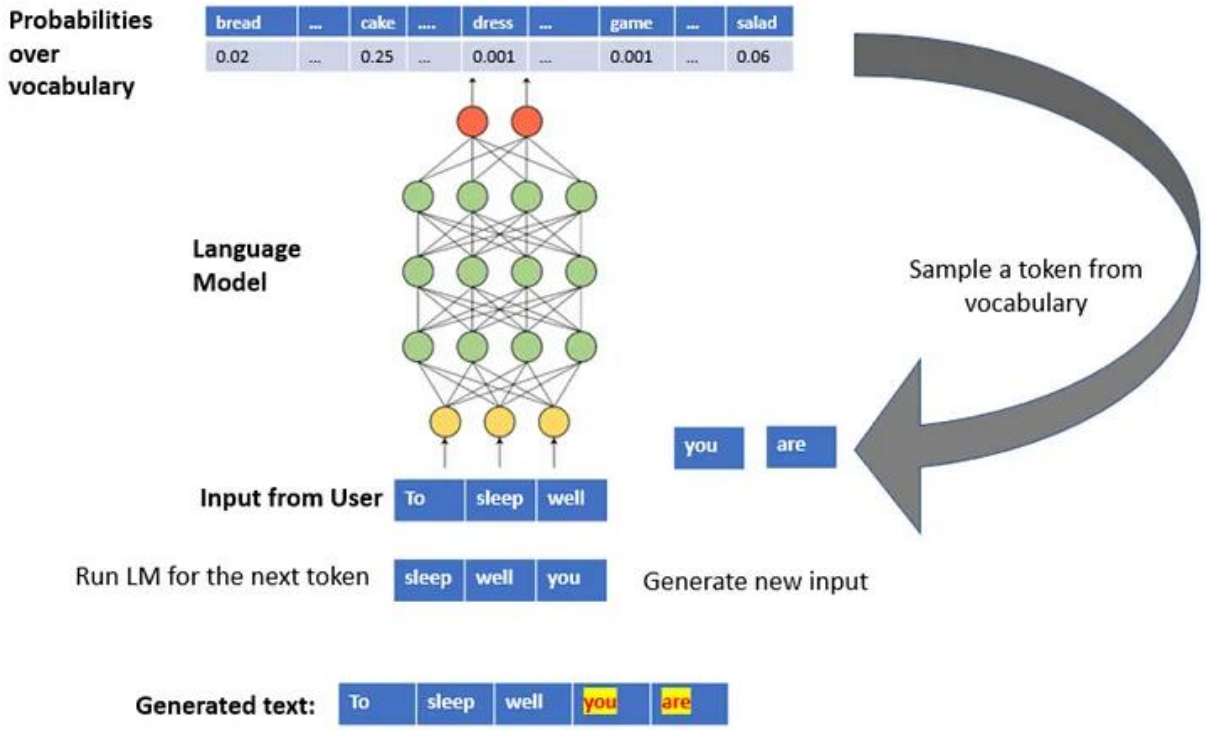
El documento se ha estructurado de la siguiente manera: Se comenzará estableciendo los conceptos teóricos fundamentales de RAG, describiendo sus arquitecturas (Ingenuo, Avanzado y Modular) y su posicionamiento frente a otras técnicas como el *fine-tuning*. A continuación, se desglosarán en detalle los componentes técnicos que conforman el pipeline de un sistema RAG, desde la ingesta y el procesamiento de datos hasta la recuperación y síntesis de la respuesta. La sección central del trabajo se dedicará a la construcción práctica del RAG biomédico, documentando la metodología y las herramientas seleccionadas.

Finalmente, se presentarán los resultados de la evaluación cuantitativa del sistema y se expondrán las conclusiones del proyecto, junto con posibles líneas de trabajo futuro.

6 Qué es un RAG.

Para entender la necesidad de desarrollar la tecnología RAG, debemos profundizar en el concepto de modelos de lenguaje de gran tamaño (o sus siglas en inglés, LLM). Estos son modelos de inteligencia artificial diseñados para procesar, generar y comprender texto con una notable capacidad. Estos modelos están basados en arquitecturas de redes neuronales profundas, como los Transformers (Vaswani, y otros, 2017), que permiten analizar grandes volúmenes de texto y aprender patrones complejos del lenguaje.

En el ejemplo de la siguiente imagen, se describe el funcionamiento básico de una red neuronal de predicción de texto: Recibe un texto como entrada (en este caso, *To sleep well…*, es decir, para dormir bien…), busca en su base de datos de palabras, y extrae la que mayor probabilidad tiene de ser la siguiente según su entrenamiento, que en este caso es *you*, es decir, tú. Luego se vuelve a repetir el proceso con la frase generada como entrada hasta que el modelo considere que ha finalizado.



[1]

Ilustración 1. Red neuronal de predicción de texto.

El proceso de entrenamiento de los LLM se realiza en dos fases principales: el preentrenamiento y el ajuste fino, o en inglés, *fine-tuning*. Durante el preentrenamiento, el modelo se entrena con enormes cantidades de texto no etiquetado para predecir la siguiente palabra en una secuencia, lo que le permite adquirir conocimientos generales sobre el lenguaje. Posteriormente, el modelo puede ser ajustado mediante *fine-tuning* con conjuntos

[1] https://ajay-arunachalam08.medium.com/an-illustration-of-next-word-prediction-with-state-of-the-art-network-architectures-like-bert-gpt-c0af02921f17

de datos específicos y tareas concretas, mejorando su rendimiento en aplicaciones como traducción automática, generación de texto o análisis de conceptos.

El uso de los LLM se ha extendido a numerosos ámbitos, desde la atención al cliente y la creación de contenido, hasta la investigación científica y el desarrollo de software. Su capacidad para generar texto coherente y realizar tareas complejas ha revolucionado la interacción humano-máquina y ha abierto nuevas posibilidades para la automatización y la asistencia en la toma de decisiones.

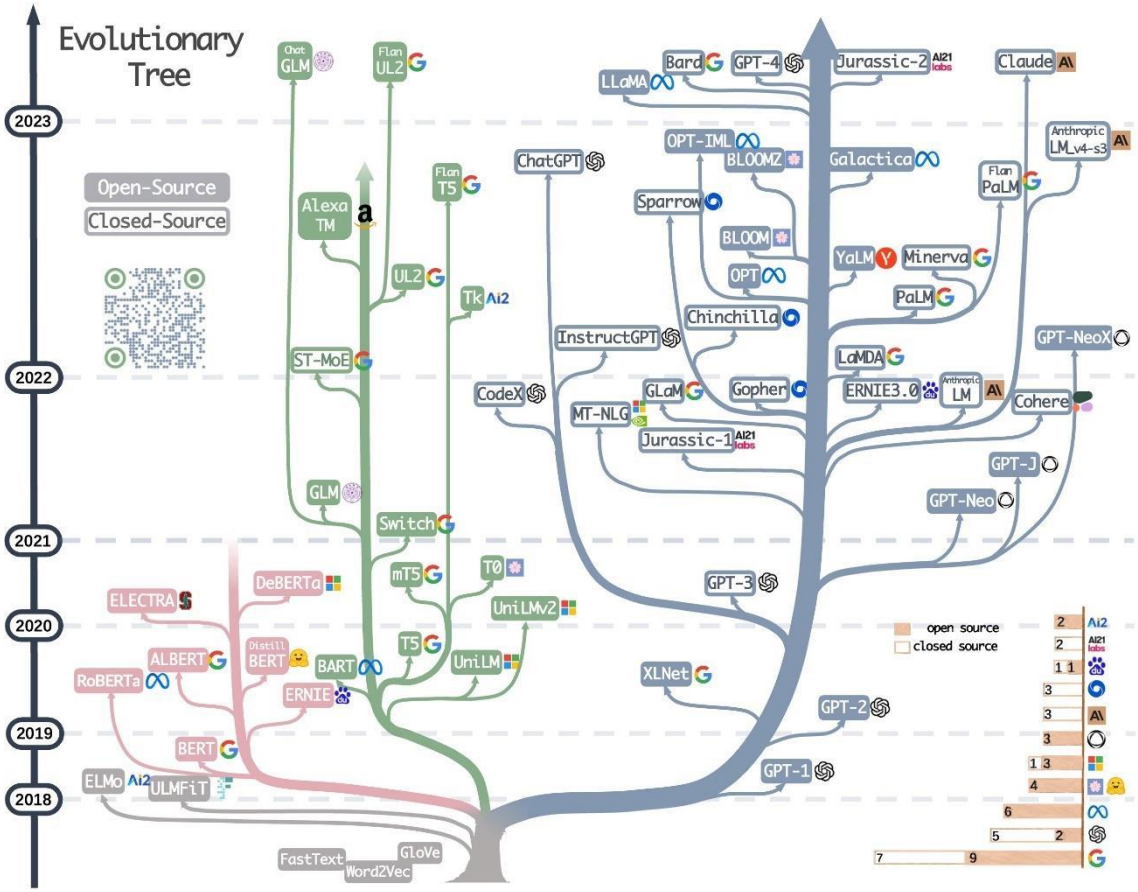


Ilustración 2. Evolución y desarrollo de los modelos de lenguaje a lo largo del tiempo (Yang, Hongye Jin, Ruixiang Tang, & Xiaotian Han , 2023)

A pesar de sus ventajas, los modelos preentrenados también presentan limitaciones y desafíos que deben tenerse en cuenta al utilizarlos. Estos pueden estar influenciados por los sesgos inherentes a los conjuntos de datos utilizados en su entrenamiento. Además, pueden tener dificultades para adaptarse a dominios o contextos específicos que difieren significativamente de los datos de entrenamiento. Esto puede requerir adaptaciones adicionales o incluso entrenamiento desde cero en algunos casos.

La tecnología RAG, de sus siglas *Retrieval Augmented Generation*, consiste en la optimización de Modelos de Lenguaje para que puedan dar respuestas más actualizadas y precisas en contextos en los que los modelos típicos no están entrenados. Esta mejora en las respuestas es posible añadiendo contexto a la consulta, y este contexto se crea a partir de una serie de documentos preseleccionados.

Para dar un ejemplo práctico de su funcionamiento, en la siguiente figura se ilustra una aplicación típica de RAG.

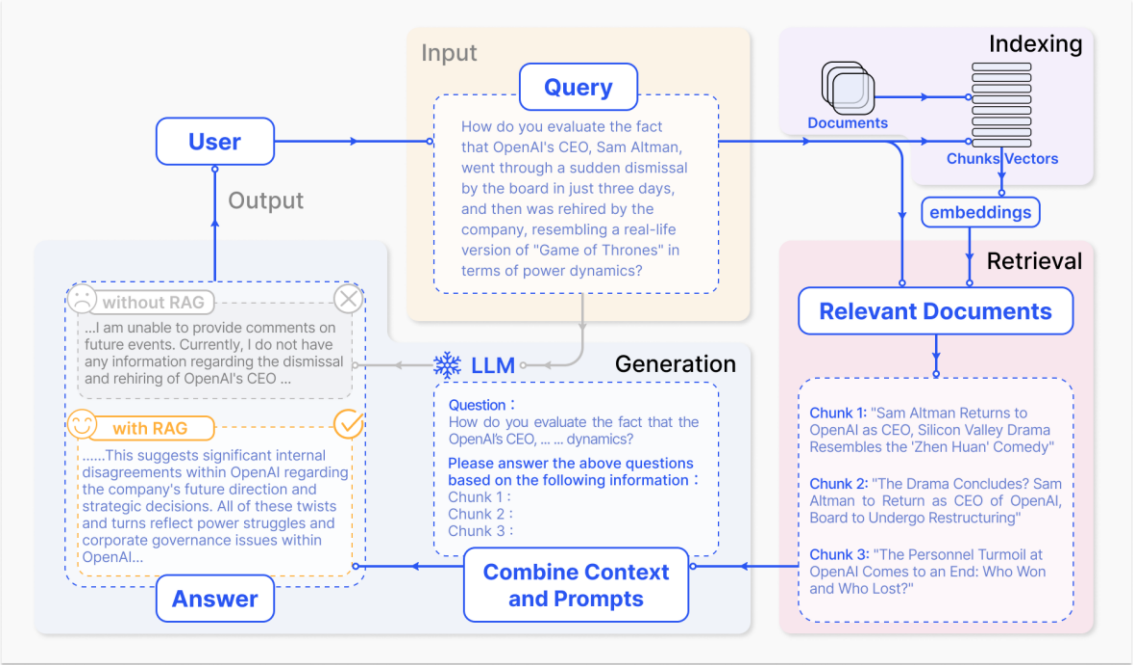


Ilustración 3. Ejemplo típico de RAG (Gao, 2023)

Esta tecnología es especialmente útil para ámbitos demasiado específicos sobre los cuales el modelo tradicional no tiene suficiente información, o para contextos que necesitan información actualizada.

En Ilustración 3, el usuario pregunta a *ChatGPT* algo relacionado con una noticia reciente. Ya que *ChatGPT* depende de los datos con los que se entrenó al modelo, no puede dar información actualizada.

La tecnología RAG hace de nodo de interconexión a partir de la información de bases de datos externas. En el caso mostrado en Ilustración 3. Ejemplo típico de RAG , el usuario realiza una consulta relativa a una noticia, por lo que requiere información actualizada (*¿Cómo valoras el hecho de que el CEO de OpenAI, Sam Altman, fue despedido por la directiva en sólo 3 días, y después fue readmitido por la compañía, asemejándose a una versión de "Juego de Tronos" en la vida real?)*. Esta misma consulta es después procesada por el modelo de lenguaje, se compara con el corpus de datos suministrados al RAG, y éste recupera documentos relacionados con la consulta, como por ejemplo: *"Sam Altman vuelve como CEO de OpenAI, el drama de Silicon Valley se asemeja a la comedia de 'Zhen Huan'"*, que luego se indexarán de forma automática al *prompt*, funcionando como apoyo para el modelo de lenguaje.
De esta forma, el modelo puede dar una respuesta más elaborada, como *"[…] Esto sugiere la existencia de desacuerdos internos dentro de OpenAI en relación con el futuro de la compañía y sus decisiones estratégicas. Todos estos giros argumentales reflejan las dificultades del poder y los problemas de la dirección corporativa dentro de OpenAI […]"*, mientras que si se usa el modelo de lenguaje base, sin ningún tipo de apoyo, su respuesta

es la siguiente: *"[…] Me es imposible proporcionar comentarios relativos a eventos futuros. En este momento, no tengo ninguna información en cuanto al despido y readmisión de Sam Altman […]"*.

7 Conceptos básicos de un sistema RAG

Para que los documentos en los que queremos que se apoye la aplicación puedan ser usados por el LLM, se pasan a un formato conocido ***tokens***, y luego se dividen en ***chunks***. Después, estos *chunks* se pasan a una representación vectorial con un proceso conocido ***embedding***, y se guardan en una **base de datos vectorial**.

El *chunking* es una técnica genial utilizada en la IA generativa para manejar grandes cantidades de datos dividiéndolos en piezas más pequeñas y manejables llamadas *chunks*. Es una técnica crucial que consiste en fragmentar grandes bloques de texto en segmentos más pequeños y fáciles de gestionar. Este proceso es especialmente importante al trabajar con modelos de lenguaje de gran tamaño (LLM) y sistemas RAG, ya que afecta directamente a la relevancia y precisión de los resultados obtenidos por estos modelos.

Sin embargo, una vez decidido el proceso de dividir nuestro corpus, o base de conocimiento del RAG, también es importante decidir cuál será el tamaño de cada *chunk*. Es por esto por lo que debemos definir una unidad para poder dividir el texto en fragmentos de tamaño similar.

Los modelos de lenguaje usan como unidad básica de texto un concepto conocido como **token**. Esta división se hace por medio de un modelo de tokenización, integrado en los modelos de lenguaje. Cuando un texto se divide en diferentes tokens, la unidad se establece por medio de un algoritmo tokenizador propio del modelo de lenguaje. Cada token se asocia con una palabra, fragmentos de una palabra o caracteres dentro del vocabulario del modelo, y tiene un identificador numérico que le corresponde. Por ejemplo, el carácter " ", es decir, el espacio en blanco, tiene como equivalente el identificador 220.

Seguidamente, en la Ilustración 4, se muestra un ejemplo de un texto dividido en tokens, usando el algoritmo tokenizador de GPT-4.
El proceso por el cual se decide en cuántos tokens se va a dividir un texto y qué representa cada token en el modelo GPT-4 viene dado por un algoritmo llamado BPE (*Byte Pair Encoding*). Para dividir el texto, el modelo reconoce las palabras más comunes como vocabulario con el que se entrenó al modelo. Es por esto por lo que palabras simples como "En", "no", "nombre" y similares están representadas por un único token, mientras que otras palabras como "astillero" están divididas en tokens que representan partes de palabras más comunes, como el sufijo "ero".

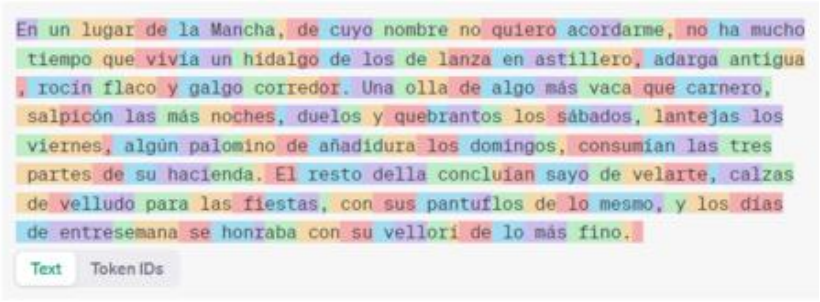


Ilustración 4. División de un texto básico en tokens

En el siguiente fragmento de código, se muestra cómo un simple texto se divide en tokens, mostrándose en orden, el identificador de cada token en el vocabulario del modelo, el número de tokens en el que se ha deconstruido el texto, y el equivalente de cada identificador, (es decir, cada token en formato de lenguaje natural).

```
import tiktoken
import textwrap

texto = "No camines delante de mí, puede que no te siga. No camines detrás de mí, puede que no te guíe. Camina junto a mí y sé mi amigo.  ~Albert Camus"

tokenizer = tiktoken.get_encoding("cl100k_base")  #Usado en GPT-4 y en el modelo de embedding ada-002 de OpenAI
tokens = tokenizer.encode(texto)
print("Tokens:")
print(tokens)
print("Cantidad de tokens:", len(tokens))
print("Tokens decodificados:")
print(str([tokenizer.decode([t]) for t in tokens]))
```

```
Tokens:
[2822, 6730, 1572, 1624, 5048, 409, 77426, 11, 21329, 1744, 912, 1028, 8531, 64, 13, 2360, 6730, 1572, 35453, 7206, 409, 77426, 11, 21329, 1744, 912, 1028, 1709, 2483, 68, 13, 8215, 2259, 63088, 264, 77426, 379, 19266, 9686, 71311, 13, 220, 4056, 67722, 8215, 355]

Cantidad de tokens: 46

Tokens decodificados:
['No', ' cam', 'ines', ' del', 'ante', ' de', ' mí', ',', ' puede', ' que', ' no', ' te', ' sig', 'a', '.', ' No', ' cam', 'ines', ' detr', 'ás', ' de', ' mí', ',', ' puede', ' que', ' no', ' te', ' gu', 'í', 'e', '.', ' Cam', 'ina', ' junto', ' a', ' mí', ' y', ' sé', ' mi', ' amigo', '.', ' ', ' ~', 'Albert', ' Cam', 'us']
```

Luego, se muestra un ejemplo en el que un texto se divide en *chunks*, con tamaño de 20 tokens y una solapación (*overlap*) de 5 *tokens*. Este último concepto consiste en que cada *chunk*, con excepción del primero, empieza un poco antes de que acabe el anterior, para no perder contexto. En el apartado **Descomposición en** chunks se explica en mayor detalle este concepto de solapación.

```
from langchain.text_splitter import TokenTextSplitter

texto = "Caminante, son tus huellas el camino y nada más; Caminante, no hay camino, se hace camino al andar. Al andar se hace el camino, y al volver la vista atrás se ve la senda que nunca se ha de volver a pisar. Caminante no hay camino sino estelas en la mar."
```

```python
#Se establece el tamaño de cada chunk
splitter = TokenTextSplitter(chunk_size=20, chunk_overlap=5)

chunks = splitter.split_text(texto)

for i, chunk in enumerate(chunks):
    print(f"Chunk {i + 1}: {chunk}")
```

```
Chunk 1: Caminante, son tus huellas el camino y nada más; C
Chunk 2: ada más; Caminante, no hay camino, se hace camino al and
Chunk 3: ace camino al andar. Al andar se hace el camino, y al vol
Chunk 4: ino, y al volver la vista atrás se ve la senda que nunca
Chunk 5:  senda que nunca se ha de volver a pisar. Caminante no hay
Chunk 6:  Caminante no hay camino sino estelas en la mar.
```

Una vez establecida la división de nuestro texto en un formato más manejable para el modelo de lenguaje, es necesario convertirlo a un formato en el que sea más fácil recuperar la información. Para nosotros, resulta fácil entender conceptos abstractos como la justicia o la belleza, pero los computadores necesitan un formato matemático en el que poder procesar esta información, por esto se utilizan un tipo específico de vectores, los ***embeddings*.**

El concepto de *embedding* consiste en una técnica de representación de información en el que, para cada concepto, se evalúan todas sus características de una forma numérica, mediante el uso de un algoritmo y se guarda como un vector de distintos números. Luego cada posición de cada vector se trata como dimensiones en un espacio, en el que cada vector o concepto, significa un punto dentro de este espacio multidimensional.

Una vez se han transformado los tokens en *embeddings*, se genera una asociación entre todas las listas de vectores, para poder darle una representación vectorial al *chunk* por completo. En el modelo de *embedding "ada-002"* de *OpenAI*, se calcula el promedio de todas las representaciones vectoriales de cada token para generar la representación vectorial de su *chunk*.

Lo siguiente sería analizar dónde se guarda esta información, y la respuesta está en las bases de datos vectoriales. En lugar de la clásica base de datos relacional, con tuplas y columnas, cada elemento se representa con un vector, que en definitiva significa un punto dentro de un complejo espacio euclídeo de amplias dimensiones (el modelo de *embeddings* de OpenAI "ada-002" utiliza 1536 dimensiones distintas para calcular la posición de cada vector). Algunos ejemplos comerciales de instancias de esta tecnología son FAISS (*Facebook AI Similarity Search*), Milvus y Pinecone.

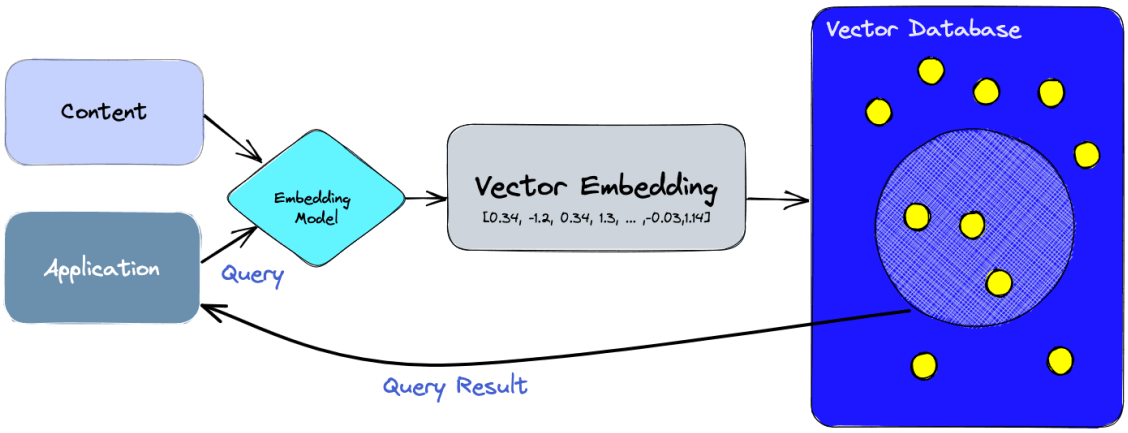


Ilustración 5. Demostración visual de una base de datos vectorial

En la Ilustración 5 se aprecia cómo sería el proceso de descomposición en vectores para su posterior uso en la aplicación:
Cuando el usuario escribe una consulta, se transforma en un formato de *embeddings* y su información numérica se usa para buscar textos similares en el corpus de nuestra aplicación. Una vez se hayan recuperado los *n* documentos más relevantes y relacionados con nuestra consulta (esta constante *n* se define en el código), se transforman de nuevo a un formato de texto, y se añaden como contexto adicional en la consulta, para que el modelo se apoye en estos nuevos datos.

Justamente, la magia de la tecnología RAG está en las búsquedas por similitud. Al tratar con vectores que significan puntos en el espacio, se puede establecer una similitud entre ellos. Cuanto más cerca estén en el espacio, más parecidos serán los conceptos. Para medir la distancia entre ellos, se puede calcular la distancia euclídea o el coseno del ángulo que forman.

A continuación, en la siguiente figura, se puede interpretar de forma gráfica cómo funcionaría una base de datos de vectores, si los vectores tuviesen únicamente 3 dimensiones, (como se ha indicado anteriormente, pueden llegar a tener miles). En la imagen, se observa cómo las palabras, en su representación de punto en un espacio tridimensional, se agrupan en función de su significado, ya que palabras como *student, school, college* (estudiante, escuela, universidad), se encuentran en posiciones cercanas entre ellas, pero están lejos de palabras como *food, meat, supply* (comida, carne, provisiones).

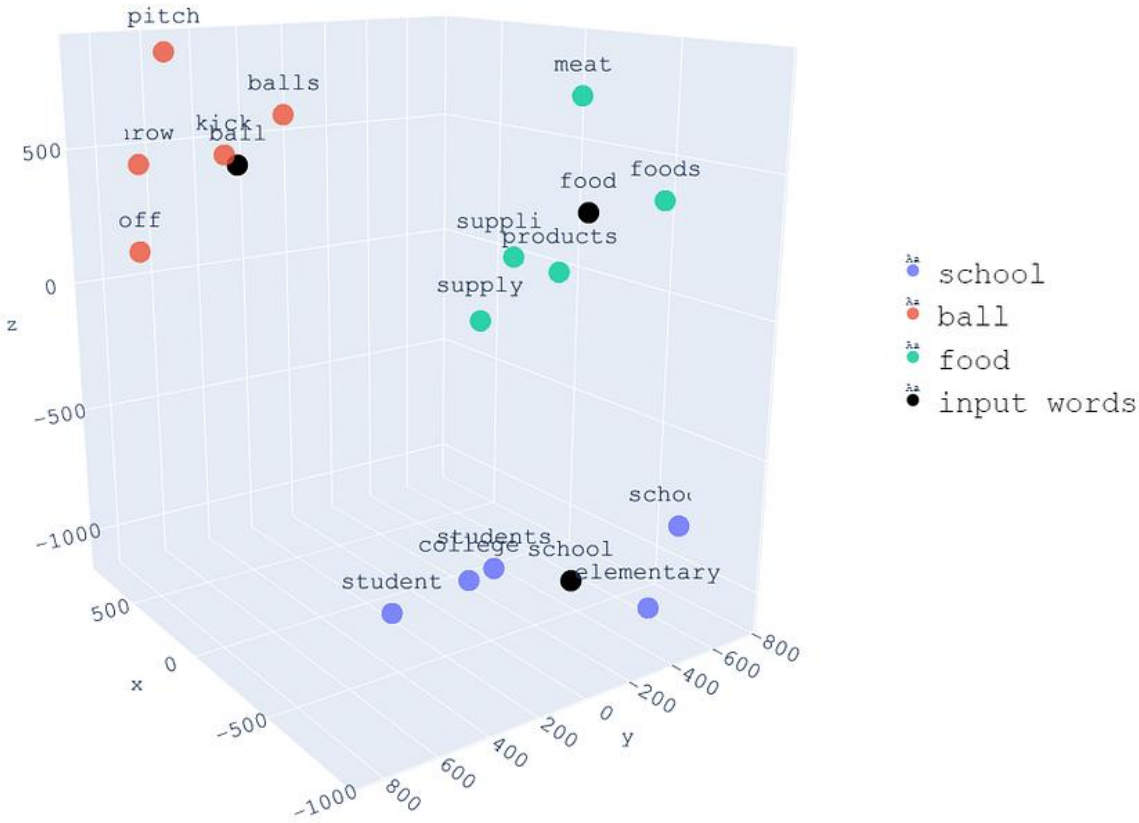


Ilustración 6. Búsqueda por similitud en base de datos vectorial

8 Estructura del desarrollo de un sistema RAG

La investigación de modelos RAG está en constante desarrollo, y según el estudio realizado por (Gao, 2023), se divide de forma genérica en 3 tipos, atendiendo a su complejidad. RAG ingenuo, RAG avanzado y RAG modular, como se puede ver en la siguiente figura.

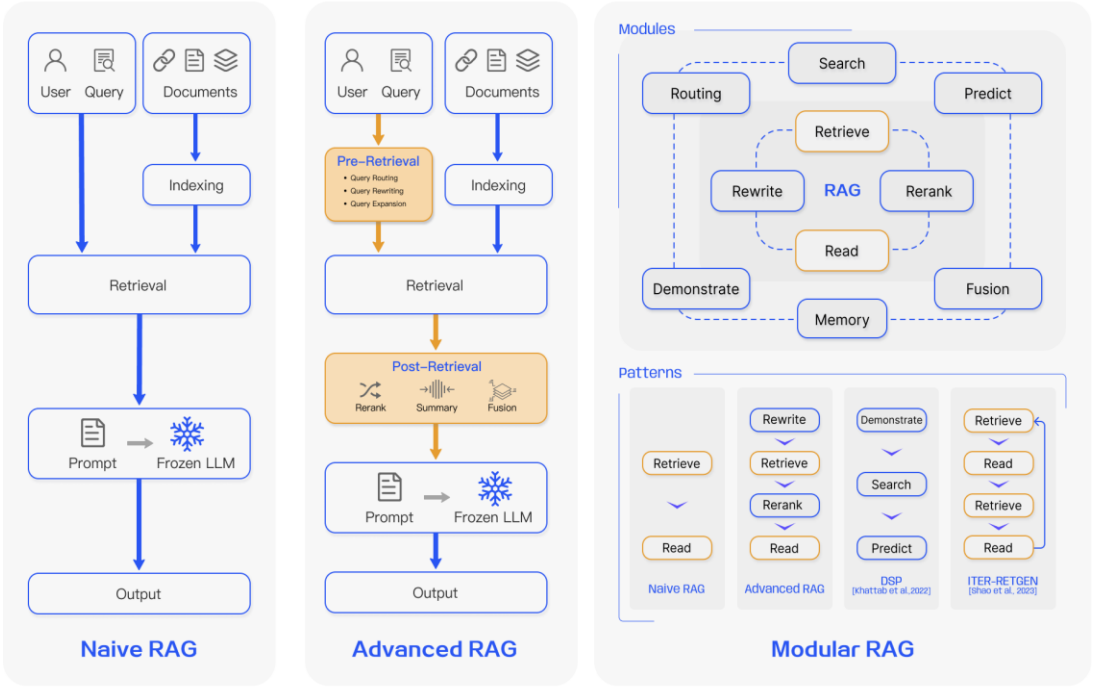


Ilustración 7. Representación en diagramas de las fases de un RAG (Gao, 2023)

Aunque los modelos RAG son muy útiles para enriquecer los LLM, hay que tomar conciencia de las limitaciones que conllevan.

8.1 RAG Ingenuo (*Naive RAG*):

Representa la metodología más primitiva dentro del paradigma de investigación del RAG. El RAG ingenuo sigue un proceso que incluye 3 fases: indexación, recuperación y generación.

La indexación comienza con la extracción y refinación de datos en distintos formatos como PDF, HTML, Markdown, Word, etc..., que se convierten a texto plano. Luego, como hemos visto en Conceptos básicos de un sistema RAG, se convierte a un sistema de representación numérica y se guarda en una base de datos vectorial para poder efectuar búsquedas por similitud.

En la fase de recuperación, se recoge la consulta del usuario, y el sistema RAG ejecuta el mismo proceso que en la fase de indexación para pasar la consulta a una representación vectorial. Tras ello, calcula la similitud entre el vector de la consulta y el vector de *chunks* almacenado en la base de conocimiento del LLM. El sistema prioriza y recupera los *chunks* que poseen mayor similitud a la consulta.
Estos *chunks* se usan como contexto expandido en el *prompt*.

En la generación, la consulta y los documentos seleccionados se sintetizan en un *prompt* coherente, y el LLM deberá producir una respuesta acorde. La estrategia del modelo a la hora de responder podrá variar dependiendo según el criterio específico de la tarea, pudiendo ceñirse estrictamente a los documentos proporcionados o sacar sus propias

conclusiones. Si llega a haber un diálogo, cualquier conversación en el historial se puede integrar en el *prompt*, habilitando al modelo a seguir una conversación de forma efectiva.

Sin embargo, el RAG ingenuo posee algunas desventajas.

- Dificultad para recuperar: en la fase de recuperación, a menudo surgen problemas relacionados con la precisión, seleccionando *chunks* irrelevantes, y falta de información crucial.
- Problemas para generar: a la hora de generar respuestas, el modelo puede sufrir alucinaciones, y producir contenido que no está basado en el contexto aportado. Esta fase también puede proporcionar respuestas irrelevantes o sesgadas, lo cual daña la calidad de las respuestas.
- Desafíos en la Aumentación: Puede resultar complicado evaluar la relevancia y la importancia de los diferentes fragmentos de información, así como mantener la coherencia en el estilo y el tono del contenido generado.

Cuando se abordan temas complejos, una única recuperación basada en la consulta original suele ser insuficiente para obtener el contexto necesario. De igual forma, si la consulta requiere una respuesta simple, y el modelo recupera un contexto extenso, es posible que se presenten alucinaciones a la hora de generar la respuesta. A esto se suma el riesgo de que los modelos de generación dependan en exceso de la información recuperada, limitándose a repetir el contenido obtenido sin aportar análisis, ideas nuevas o información sintetizada.

8.2 RAG Avanzado (*Advanced RAG*)

El RAG avanzado introduce una serie de mejoras para superar las limitaciones del RAG ingenuo.
Se centra en mejorar la calidad de la recuperación, mediante estrategias pre-recuperación y post-recuperación. Para sobrellevar los problemas de indexación, esta versión refina sus técnicas de indexación mediante el uso de una ventana deslizante, segmentación refinada y la incorporación de metadatos. Además, incorpora varios métodos de optimización para el proceso de recuperación.

- Pre-recuperación: en este proceso, el objetivo principal es optimizar la consulta original, para que el modelo pueda comprender mejor lo que el usuario requiere y para que la indexación localice más fácilmente los recursos a buscar.

Para optimizar la consulta del usuario, se suelen usar métodos como división en subconsultas o reescritura de consulta.

- Post-recuperación: Una vez que tenemos el contexto, es crucial integrarlo de forma efectiva con la consulta. Los principales métodos incluyen la reordenación de *chunks* y contracción de texto.
Reordenar la información recuperada para encontrar el contexto más relevante de cara al *prompt* es un paso clave. Este proceso ya ha sido integrado en diversos frameworks como LlamaIndex, Langchain y HayStack.

Otorgar todos los documentos relevantes directamente al LLM, pueden desencadenar en una sobrecarga de información, menospreciando los detalles más importantes. Para combatir esto, los métodos post-recuperación se centran en seleccionar la información esencial, reduciendo el tamaño del contexto que se envía.

8.3 RAG Modular (*Modular RAG*)

La arquitectura del RAG incorpora varias estrategias para mejorar sus componentes, como añadir un módulo de búsqueda de similitudes y redefinir la recuperación mediante *fine-tuning*.

Estas innovaciones tratan de incluir distintos módulos para mejorar cualidades específicas del sistema RAG. La transición a un enfoque modular se está volviendo predominante, puesto que, debido a su estructura secuencial, permite un entrenamiento individualizado en sus componentes. A continuación, se enumeran algunos de los diferentes módulos que se pueden integrar en un sistema RAG.

1) **Nuevos módulos:**

- Módulo de búsqueda: se adapta para escenarios específicos, habilitando búsquedas directas a través de varias fuentes de datos como motores de búsqueda, bases de datos y gráficos de conocimiento.

- Módulo de fusión: resuelve los problemas de búsquedas tradicionales mediante una estrategia multi-consulta que divida la consulta del usuario en distintas perspectivas, usando búsquedas vectoriales de forma paralela y reordenación inteligente para poder interpretar nuevos conocimientos.

- Módulo de memoria: aprovecha la memoria del LLM para guiar la recuperación, creando un conjunto de memoria ilimitado que alinea el texto *de una forma más cercana a* la distribución de datos mediante un proceso iterativo de auto-mejora.

- Módulo de enrutamiento: navega por diferentes fuentes de datos, mezclando diferentes flujos de información.

- Módulo de predicción: intenta reducir la redundancia y ruido generando contexto directamente desde el LLM, asegurando la precisión a la hora de construir una respuesta.

- Módulo adaptador de tareas: conduce al RAG a varias tareas secuenciales, automatizando la recuperación de *prompts*. Este enfoque no solo simplifica el proceso de recuperación, sino que también mejora la calidad y relevancia de la información recibida, alcanzando un mayor abanico de tareas y solicitudes que poder completar con eficiencia y exactitud.

2) **Nuevos Patrones:**

Los RAG modulares ofrecen una flexibilidad destacable permitiendo la sustitución de módulos o reconfiguración para objetivos específicos. Esto va más allá de las estructuras fijas de las RAG ingenuo y avanzado, que se caracterizan por un mecanismo simple de "recuperación y lectura".

Además, el RAG modular expande dicha flexibilidad integrando nuevos módulos y ajustando la interacción entre los ya existentes, mejorando sus aplicaciones en distintas tareas. Algunas innovaciones como *la reescritura o clasificación de consulta* se aprovechan de las capacidades del LLM para refinar consultas de recuperación mediante un módulo de reescritura y un mecanismo de realimentación para el LLM, mejorando el rendimiento.

Otro beneficio de la arquitectura flexible es que el sistema RAG puede fácilmente integrarse con otras tecnologías, como *fine-tuning* o aprendizaje reforzado.

9 RAG vs *fine-tuning*

El crecimiento de los LLM ha atraído una atención considerable. Entre los métodos de optimización de modelos de lenguaje, el RAG a menudo se compara con *fine-tuning*, e ingeniería de *prompt*, o como se conoce en inglés, *prompt engineering*.

La ingeniería de *prompt* consiste en optimizar la redacción de las consultas al modelo para guiarle a dar una mejor respuesta.
El *fine-tuning* consiste en entrenar al modelo original con datos y preguntas relacionadas al contexto pertinente, mientras que en el RAG, estos nuevos datos simplemente se añaden como contexto en la consulta, no forman parte de su base de conocimiento.

Cada método tiene características distintas como se ilustra en la siguiente ilustración, la cual es un resultado a partir del estudio proporcionado por (Gao, 2023). En esta figura, se describen las principales técnicas para asistir a los modelos de lenguaje, en función de su adaptación requerida al modelo en el eje X y el conocimiento externo requerido en el eje Y.

De esta forma, el *prompt engineering* no necesita adaptar el modelo ni recoger información externa, porque pretende optimizar el conocimiento que el modelo ya posee. La técnica de *fine-tuning* necesita adaptar el modelo y las técnicas de RAG necesitan extraer conocimiento de artículos y noticias actuales para funcionar de forma correcta. Si se combinan ambas, también aumenta la complejidad del sistema.

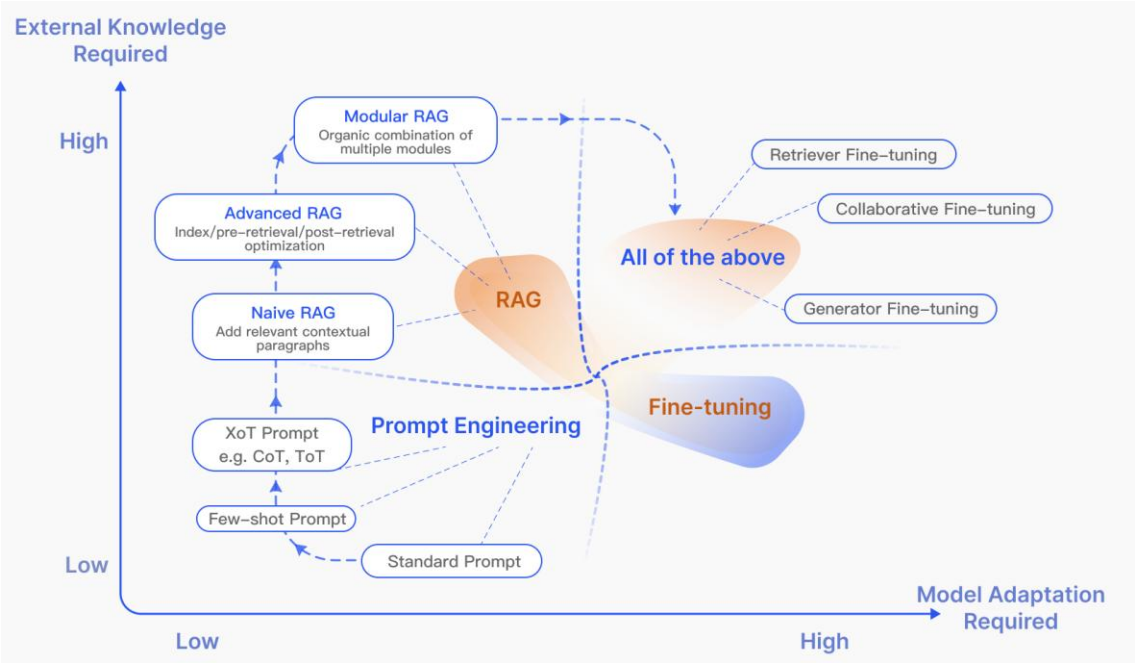


Ilustración 8. RAG comparado con fine-tuning e ingeniería de prompt (Gao, 2023)

El RAG destaca en entornos dinámicos ofreciendo conocimiento actualizado y un uso efectivo de fuentes de conocimiento externas con un amplio grado de interpretabilidad. Sin embargo, conlleva una mayor latencia y consideraciones éticas a la hora de tomar los datos. Por otra parte, el *fine-tuning* es más estático, necesitando reentrenamiento para actualizaciones, pero habilitando un mayor grado de configuración del comportamiento y estilo del modelo. Requiere más recursos computacionales para la preparación del conjunto de datos y entrenamiento y, aunque puede reducir el número de alucinaciones, puede tener problemas a la hora de lidiar con datos que no conoce.

En múltiples evaluaciones de rendimiento en varias tareas complicadas de temas diferentes realizadas en (Gao, 2023), se concluye que, aunque el *fine-tuning* sin supervisión muestra alguna mejora, el RAG siempre le supera, tanto para conocimiento existente que ya se encontraba en el entrenamiento tanto como para nuevo conocimiento. Además, también se concluye que a los LLM les cuesta encontrar nueva información contrastada mediante FT sin supervisión.

La elección entre RAG y FT depende de las necesidades específicas de la dinámica de los datos, y las capacidades computacionales *del entorno* de la aplicación. RAG y FT no son mutuamente excluyentes y se pueden complementar entre sí, mejorando las capacidades del modelo en distintos niveles. En determinados casos, su uso combinado puede dar como resultado un rendimiento óptimo.

10 Estructura de un RAG

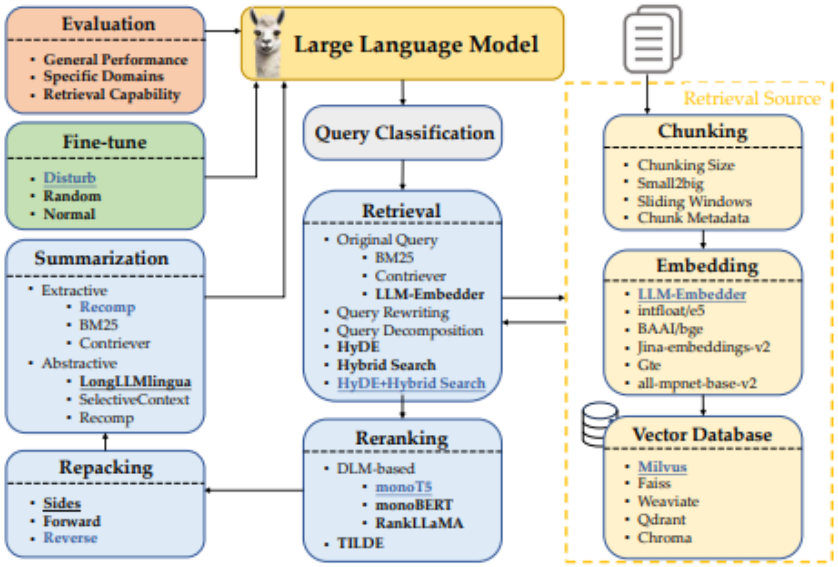


Ilustración 9. Diagrama de componentes de un sistema RAG (Wang, y otros, 2024)

Las técnicas de generación aumentada por recuperación (RAG) han demostrado ser efectivas para integrar información actualizada, reducir alucinaciones y mejorar la calidad de las respuestas en dominios especializados. Sin embargo, estas técnicas a menudo enfrentan desafíos en su implementación y tiempos de respuesta.

En (Wang, y otros, 2024), se analizan los enfoques RAG existentes y sus combinaciones para identificar prácticas óptimas que equilibren rendimiento y eficiencia (ver Ilustración 9. Diagrama de componentes de un sistema RAG). Se proponen estrategias para la implementación de RAG, destacando que las técnicas de recuperación multimodal pueden mejorar significativamente las capacidades de respuesta a preguntas sobre entradas visuales y acelerar la generación de contenido multimodal.

Además, concluyen que los RAG permiten aplicaciones rápidas en organizaciones específicas sin necesidad de actualizaciones en los parámetros del modelo, siempre que se proporcionen documentos relacionados con la consulta.

En esta sección, entraremos en detalle para describir los componentes que forman el flujo de trabajo de un RAG. Para cada módulo, revisaremos cada enfoque y los mejores métodos para cada componente, según (Wang, y otros, 2024).

11 Clasificación de consulta

No todas las consultas requieren usar el RAG por la naturaleza del LLM. Mientras que el RAG puede mejorar la calidad de la información devuelta y reducir el riesgo de alucinaciones, un uso frecuente de la recuperación aumentada puede resultar en un aumento del tiempo de respuesta. Por tanto, empezamos clasificando las consultas para

comprobar la necesidad de recuperación. Si no lo requieren, simplemente se usará el LLM por defecto.
La recuperación se necesita cuando el conocimiento requerido va más allá del que posee el modelo. Sin embargo, la necesidad de recuperación varía según la tarea. Por ejemplo, un LLM al que se pide una traducción, podrá responder sin problemas, pero si se le pregunta una duda técnica sobre una tecnología que no conoce, necesitará recuperar información. Por tanto, se propone clasificar las consultas por tareas. La división se ha establecido en 15 tipos diferentes de tareas. Aquellas tareas que se basen en información otorgada por el usuario, serán marcadas como “suficiente”, si no es así, serán marcadas como “Insuficiente” y es posible que se necesite la recuperación. Se ha entrenado un clasificador para automatizar este proceso.

12 Descomposición en *chunks*

La división de documentos en segmentos más pequeños es crucial para mejorar la precisión de la recuperación y evitar problemas de longitud de respuesta en los LLM. Este proceso se puede hacer a distintos niveles de tamaño:

-Nivel Token: es simple pero puede llegar a dividir frases, afectando a la calidad de recuperación.
-Nivel Semántico: usa el LLM para determinar los puntos de división, manteniendo el contexto pero más caro computacionalmente.
-Nivel frase: equilibrado entre mantener la semántica del texto y ser eficiente.
Entraremos en más detalle en función de 4 factores relativos a los *chunks*.

1) Tamaño de los *chunks*:

El tamaño del *chunk* tiene un gran impacto en el rendimiento. Si el *chunk* es más grande, provee mayor contexto, mejorando la comprensión del modelo, pero aumentando el tiempo de respuesta. Lo contrario ocurre a medida que el *chunk* es más pequeño.

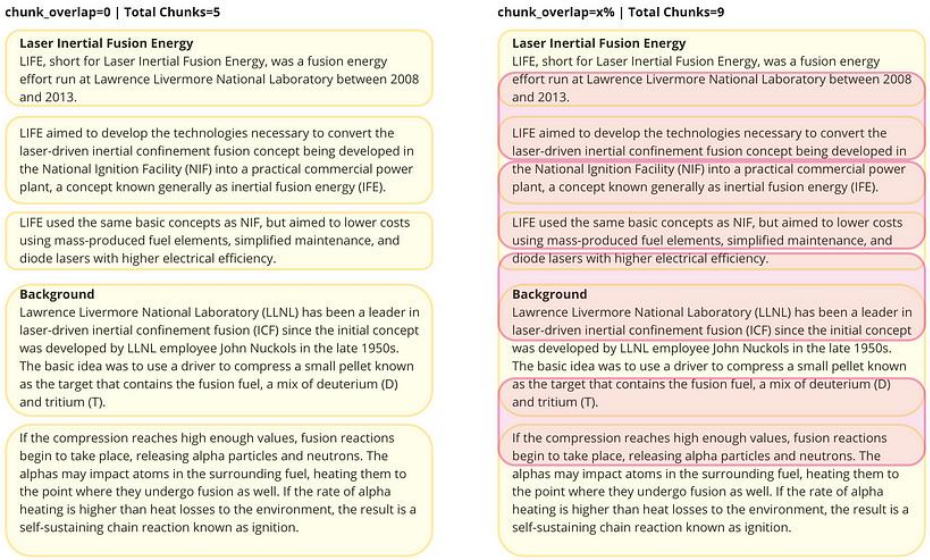
Para obtener el tamaño óptimo, se usan varias métricas como **fidelidad** o **relevancia**. La **fidelidad** comprueba que la respuesta sea una alucinación o que se corresponda con el texto recuperado. La **relevancia** comprueba que el texto recuperado y las respuestas están relacionadas con las consultas.

En las pruebas, se ha usado una superposición de *chunks* de **20 *tokens***. El tamaño que resulta en mayor fidelidad es **512 tokens**, mientras que el tamaño con mayor relevancia es **256**.

2) Técnicas de descomposición de *chunks*:

Cuando los textos se dividen en *chunks*, es posible que se pierda información relevante al hacer el corte entre un *chunk* y otro, por ejemplo, si el corte se efectúa justo en la mitad de la frase.

Existen técnicas avanzadas como *small-to-big* y *sliding-window* para mejorar la calidad de recuperación organizando relaciones en los bloques de *chunks*.
Small-to-big consiste en unir *chunks* que son demasiado pequeños en *chunks* más grandes para poder para poder dar contexto más relevante. *Sliding-window* consiste en de dejar una solapación entre *chunks*, de forma que cada *chunk* empiece ocupando parte del final del *chunk* anterior.



Para demonstrar la efectividad de estas técnicas avanzadas, se han realizado una serie de pruebas. En estas pruebas, el tamaño de *chunk* más pequeño es de **175 *tokens***, y el más grande es de **512**, y la superposición es de **20 *tokens***. Dentro de las técnicas, ***sliding-window*** es el más efectivo tanto para mantener la relevancia como la fidelidad.

3) Selección de modelo *embedding*:

Elegir el modelo de *embedding* correcto es esencial para mantener la semántica de las consultas y los chunks. Se ha usado el modelo de *FlagEmbedding* para evaluar los resultados. El que ha obtenido mejores resultados es el modelo ***LLM-Embedder***.

4) Añadir metadatos:

Se puede mejorar la recuperación añadiendo información adicional a los *chunks*, como títulos, palabras clave y preguntas hipotéticas, para que el LLM entienda mejor la información recuperada.

13 Base de datos de vectores

Las bases de datos de vectores guardan los vectores de *embedding* con sus metadatos asociados, permitiendo la recuperación eficiente de documentos relevantes mediante varios métodos como ANN (aproximación al vecino más cercano).

Para seleccionar una base de datos adecuada a la investigación, se han elegido 4 criterios: múltiples tipos de índice, soporte de vectores de escala millonaria, búsqueda híbrida y capacidad de ser guardado en la nube. Si tiene múltiples tipos de índice, se asegura la

flexibilidad para optimizar búsquedas basadas en distintos tipos de datos. El soporte de vectores de escala es importante si se quieren manejar conjuntos de datos muy grandes. La búsqueda híbrida combina búsqueda de vectores con búsqueda tradicional de palabras claves.

En la investigación realizada por (Wang, y otros, 2024), **Milvus** es el único que cumple todos los requisitos.

14 Métodos de recuperación

Dada una consulta de un usuario, el modelo de recuperación selecciona, los documentos *n*-primeros con mayor relevancia dentro de una base de conocimiento basada en la similitud de la consulta con los documentos. Después, el modelo de generación usa estos documentos para formular una respuesta apropiada a la consulta. Sin embargo, las consultas por defecto a menudo no devuelven el resultado esperado debido a una expresión poco precisa y una falta de información por parte del LLM. Para resolver estos problemas, hemos evaluado 3 métodos para transformar las consultas:

- Reescritura de consultas: mejora las consultas para que se acerquen más a los documentos relevantes.
- Descomposición de consultas: Este enfoque implica recuperar documentos según unas preguntas derivadas de la consulta original, lo cual resulta más complejo.
- Generación de pseudo-documentos: Este enfoque genera un documento hipotético basado en la consulta del usuario y usa un *embedding* de respuestas hipotéticas para recuperar documentos similares. Una implementación destacable es **HyDE**.
Algunos estudios recientes indican que combinando una búsqueda basada en el léxico con una basada en vectores mejora considerablemente el rendimiento.

15 Métodos de reorganización de *chunks*

Después de la recuperación inicial, se ejecuta una fase de reorganización para mejorar la relevancia de los documentos recuperados, asegurando que la mejor información se encuentra la primera de la lista. Esta fase usa métodos más precisos e intensivos para funcionar eficazmente.
Consideramos dos enfoques en nuestro módulo de reorganización. **Reorganización DLM**, que emplea clasificación y **Reorganización TILDE**, que se centra en el objetivo de la consulta. Estos enfoques priorizan el rendimiento y la eficiencia, respectivamente.

- DLM: Este método aprovecha modelos de lenguaje profundos (DLM) para la reorganización. A estos modelos se les ha hecho un *fine-tuning* para clasificar la relevancia de los documentos con respecto a la consulta según un criterio de verdadero o falso.

- TILDE: Este calcula la probabilidad de cada término de la consulta de manera independiente prediciendo las probabilidades de los tokens a lo largo del vocabulario del modelo. Los documentos se puntúan sumando las probabilidades logarítmicas previamente calculadas de los tokens de la consulta, lo que permite un reordenamiento rápido durante la inferencia.

Una vez realizadas las pruebas, se descubre que **RankLLaMa** es el que devuelve mejor rendimiento, mientras que **TILDEv2** es el ideal para un proceso rápido con una colección fija.

16 Reensamblado de documentos

El rendimiento de subprocesos, como la generación de respuesta del LLM, puede estar afectado por el orden en el que se suministran los documentos. Para solucionar esto, se incorpora un módulo de reensamblado en el flujo de trabajo después de reorganizar, que involucra tres métodos, “directo”, “inverso” y “lateral”.
El método directo los ordena de forma descendente, según su puntuación de relevancia en la fase de reorganización, mientras que el método inverso los ordena de forma ascendente. Hay estudios que demuestran que la mayor eficiencia se encuentra al posicionar los documentos al principio o al final de la lista, resultando en el método lateral.

17 Síntesis

Los resultados de la recuperación pueden contener información redundante o innecesaria, lo que impide al modelo de lenguaje de generar respuestas útiles. Además, los *prompts* largos pueden ralentizar el proceso, por ello, son cruciales los métodos para sintetizar los documentos recuperados en el flujo de trabajo del RAG.
Las tareas de síntesis pueden ser extractivas o abstractivas. Los métodos extractivos dividen el texto en frases, y luego los puntúan y ordenan según su importancia. Los abstractivos sintetizan información de múltiples documentos para reorganizarla y generar un resumen coherente. Estas tareas pueden ser basadas en consultas o no. En este estudio, ya que el RAG devuelve información relativa a las consultas, nos centramos exclusivamente en métodos basados en consultas.
-Recomp: tiene sintetizadores extractivos y abstractivos. El sintetizador extractivo selecciona frases útiles, mientras que el sintetizador abstractivo recoge información de varios documentos.
-LongLLMLingua: mejora LLMLingua centrándose en información clave relativa a la consulta.
-Contexto selectivo: mejora la eficiencia del LLM identificando y retirando información redundante en el contexto de entrada.

Se recomienda **Recomp** por su destacable rendimiento.

18 *Fine-tuning* generador

En esta sección el foco del estudio está en hacer *fine-tuning* al generador mientras dejamos el *fine-tuning* del recuperador para otro momento. Se busca investigar el impacto del *fine-tuning*, sobre todo la influencia de contexto relevante o no, en el rendimiento del generador. Las pruebas sugieren que mezclar contextos aleatorios con contextos relevantes durante el entrenamiento puede mejorar la robustez del generador a información irrelevante mientras se asegura el uso efectivo de contexto relevante. Por ello, se identifica que la práctica de suministrar una mezcla de documentos relevantes con otros aleatorios durante el entrenamiento es el mejor enfoque.

19 Evaluación

En el estudio de (Wang, y otros, 2024), se han realizado una serie de pruebas para evaluar el rendimiento de los sistemas RAG. Estas pruebas son: **Razonamiento de sentido común, Comprobación de hechos, Preguntas y respuestas de dominio abierto, Preguntas y respuestas de multi-salto, y Preguntas y respuestas médicas.**
La métrica usada para la evaluación de Razonamiento de sentido común, Comprobación de hechos y Preguntas y respuestas médicas ha sido la precisión, para el resto, se han usado la puntuación F1 a nivel de token y puntuación *Exact Match*. La puntuación final del RAG fue calculada mediante el promedio de las 5 cualidades mencionadas anteriormente.

1) Resultados y análisis

Basándonos en los resultados de las pruebas, los aspectos más destacables son:

- Módulo de clasificación de consultas: Este módulo se referencia y contribuye tanto a la efectividad como a la eficiencia, dando lugar a una mejoría en la puntuación general de 0,428 a 0,443, y una reducción en latencia de 16,41 a 11,58 segundos por consulta.
- Módulo de recuperación: Mientras que el método HyDE híbrido obtuvo la mejor puntuación RAG de 0,58, también requiere un alto coste computacional de 11,71 segundos por consulta. Además, los métodos híbridos u originales son los recomendados, puesto que reducen el tiempo de respuesta manteniendo una eficacia parecida.
- Módulo de reordenación o reranking: Si no existe este módulo, se nota una caída en la eficacia de la aplicación, señalando su importancia. MonoT5, acumuló la puntuación más alta, demostrando su eficacia a la hora de aumentar la relevancia de los documentos recuperados.
- Módulo de reempaquetado o repacking: La configuración inversa demostró un rendimiento superior. Esto demuestra que posicionar contenido relevante cerca de la consulta aumenta la calidad de las respuestas.
- Módulo de resumen o sumarización: Recomp fue el que demostró mejor rendimiento, aunque se pueden conseguir parecidos con menor latencia eliminando este módulo.

Los resultados de los experimentos demuestran que cada módulo contribuye de forma única al rendimiento del sistema RAG al completo. El módulo de clasificación mejora la precisión y reduce la latencia, mientras que los de recuperación y reordenación mejoran la capacidad del sistema para responder una mayor variedad de preguntas. Los de reempaquetado y síntesis afinan la salida, asegurando una mayor calidad de respuestas.

20 Tutorial simple de construcción de un sistema RAG explicado mediante un notebook comentado

En este apartado nos vamos a adentrar de manera pedagógica en la creación de una aplicación RAG simple en la nube usando *Google Colab* y con escasos documentos. Pese a su simpleza, consigue demostrar que las aplicaciones RAG son más eficientes que los modelos de lenguaje el contexto se escapa del conocimiento preentrenado del modelo.

El primer paso dentro del tutorial sería establecer las claves de API de los servicios que vayamos a usar (en este caso usamos la clave de OpenAI) y la instalación de dependencias.

```
!pip install langchain-community unstructured pypdf tiktoken faiss-cpu openai -q
```

Lo siguiente sería declarar el corpus que usará nuestro sistema RAG. En este caso, está compuesto por 11 artículos médicos que tratan sobre el síndrome de Rett descargados desde el portal PubMed en formato PDF, los cuales cargamos desde un directorio de Google Drive usando el *framework* Langchain.

```
from google.colab import drive
drive.mount('/content/drive')

dir="/content/drive/MyDrive/Colab Notebooks/TFG Alejandro/Rett"

from langchain.document_loaders import PyPDFLoader, DirectoryLoader
loader = DirectoryLoader(
    dir,
    glob="**/*.pdf",
    loader_cls=PyPDFLoader,
)

documents = loader.load()
```

Una vez hayamos escogido los documentos que queramos que use nuestra aplicación, debemos pasarlo a un formato que el modelo de lenguaje pueda incorporar.

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter

# Dividir los documentos en fragmentos más pequeños
text_splitter = CharacterTextSplitter(
    separator="\n",
    chunk_size=512,
    chunk_overlap=20,
    length_function=len
)
texts = []
for doc in documents:
    texts.extend(text_splitter.split_text(doc.page_content))

# Crear los embeddings utilizando OpenAI
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")
```

En este código, se divide cada documento en *chunks* con un tamaño de 512 *tokens* y una solapación de 20 *tokens*, siguiendo la recomendación de (Wang, y otros, 2024), y se usa el modelo de *embeddings* “ada-002” para transformar cada token a su equivalente en un sistema de representación vectorial. En la siguiente figura se muestra como ejemplo uno de los vectores que conforman la base de datos, creada por medio de FAISS (*Facebook AI Similarity Search*).

```
from langchain.vectorstores import FAISS
db = FAISS.from_texts(texts, embeddings)
```

```
# Obtener los embeddings del índice FAISS
embeddings_array = db.index.reconstruct_n(0, db.index.ntotal)

# Convertir a lista para facilitar la lectura (opcional)
embeddings_list = embeddings_array.tolist()

# Mostrar el número de embeddings generados y el primero como ejemplo
print(f"Número de embeddings generados: {db.index.ntotal}")
print(f"Primer embedding (dimensión {len(embeddings_list[0])}):
{embeddings_list[0]}")
```

```
Número de embeddings generados: 1353
Primer embedding (dimensión 1536): [-0.029281478375196457, 0.002531148726120591,
0.03168327733874321, -0.038990870118141174, -0.007735574152320623, …
```

Falta ajustar un detalle de la configuración de la base de datos de vectores, que es el recuperador. En este caso, le decimos que realice búsquedas por similitud y recoja los 2 *chunks* con mayor relevancia.

```
retriever = db.as_retriever(search_type="similarity", search_kwargs={"k": 2})
```

Con esto ya queda configurada la base de datos de vectores y podemos comenzar a usar la aplicación de forma práctica, por medio de la consulta *"¿Qué puedes decir sobre el uso de trofenitide y cannabis en pacientes de sindrome de Rett"?:*

```
query="¿Qué puedes decir sobre el uso de trofenitide y cannabis en pacientes de
sindrome de Rett?"

docs = retriever.get_relevant_documents(query)

for i, doc in enumerate(docs):
    print(f"Documento {i + 1}:")
    print(doc.page_content)
    print("\n" + "="*80 + "\n")  # Separador para claridad

# Combina los documentos recuperados en un contexto único
context = "\n\n".join([doc.page_content for doc in docs])

# Construye el prompt para el modelo
prompt = f"""
Usa la siguiente información relevante para responder la pregunta:
{context}

Pregunta: {query}
Respuesta:
"""

from openai import OpenAI
import os

client = OpenAI(
    api_key=os.environ.get("OPENAI_API_KEY"),
)

response = client.chat.completions.create(
    model="gpt-3.5-turbo",  # Modelo a utilizar
    messages=[
        {"role": "user", "content": prompt}  # Mensaje del usuario
```

```
    ],
    temperature=0.3,          # Control de creatividad
    max_tokens=256,           # Máximo de tokens en la respuesta
    top_p=1,                  # Nucleus sampling
    frequency_penalty=0,      # Penalización por repetición de palabras
    presence_penalty=0        # Penalización por introducir nuevos temas
)

print(response.choices[0].message.content)
```

Aquí se declara la consulta, se crea una plantilla de *prompt* en la que se añade el contexto recuperado a la consulta, y se configura el modelo de lenguaje que se usará para generar la respuesta.

```
Documento 1:
o más. La trofinetida reduce la inflamación en el cerebro, aumenta la cantidad de una
proteína protectora en el cerebro y evita que algunas células lleguen a ser
demasiado activas.
Otros tratamientos se centran en el tratamiento de síntomas o comportamientos
específicos que se observan en el síndrome de Rett. Los medicamentos pueden ayudar
con los problemas del movimiento y la respiración, y con las crisis convulsivas.

================================================================================

Documento 2:
las estrategias de comunicación efectiva. El apoyo psicosocial a las familias es
esencial. Los enfoques farmacológicos tienen como objetivo mejorar los trastornos
del sueño, los problemas respiratorios, las crisis epilépticas, los movimientos
estereotipados y el bienestar general. Dado que los pacientes con RTT presentan un
mayor riesgo de sufrir arritmias potencialmente mortales asociadas a un intervalo
QT prolongado, se deberá evitar una serie de fármacos (como los antibióticos
macrólidos).

================================================================================
```

Aquí se muestra de forma estructurada los 2 *chunks* que recoge para dar contexto al *prompt*.
La respuesta del modelo en este caso es la siguiente:

```
El uso de trofenitide en pacientes con síndrome de Rett parece ser beneficioso, ya
que reduce la inflamación en el cerebro, aumenta la cantidad de una proteína
protectora y evita que algunas células se vuelvan demasiado activas. Por otro lado,
el cannabis también puede ser útil en el tratamiento de algunos síntomas del
síndrome de Rett, como problemas de movimiento, respiración y crisis convulsivas.
Sin embargo, es importante consultar con un médico especialista antes de iniciar
cualquier tratamiento con cannabis, ya que puede tener efectos secundarios y
contraindicaciones en ciertos pacientes.
```

En el siguiente fragmento de código se llama al modelo sin RAG, es decir, sin contexto adicional en el *prompt*, para comparar la respuesta.

```
from openai import OpenAI

client = OpenAI(
    api_key=os.environ.get("OPENAI_API_KEY"),
)

response = client.chat.completions.create(
    model="gpt-3.5-turbo",  # Modelo a utilizar
    messages=[
        {"role": "user", "content": query}  # Mensaje del usuario
```

```
    ],
    temperature=0.3,          # Control de creatividad
    max_tokens=256,           # Máximo de tokens en la respuesta
    top_p=1,                  # Nucleus sampling
    frequency_penalty=0,      # Penalización por repetición de palabras
    presence_penalty=0        # Penalización por introducir nuevos temas
)

print(response.choices[0].message.content)
```

En este caso, la respuesta que genera el modelo es la siguiente:

```
No hay suficiente evidencia científica para respaldar el uso de trofenitide y
cannabis en pacientes con síndrome de Rett. Siempre es importante consultar con un
médico especializado antes de comenzar cualquier tratamiento, ya que cada paciente
es único y puede responder de manera diferente a diferentes terapias. Es
fundamental seguir las recomendaciones médicas y realizar un seguimiento adecuado
para garantizar la seguridad y eficacia del tratamiento.
```

Como se puede apreciar, el modelo genérico da una respuesta poco relevante que no aporta demasiada información, mientras que el modelo es mucho más atrevido y es capaz de dar datos concretos de su uso.

21 Construcción de un RAG Biomédico.

21.1 Metodología

Lo que se trata lograr en este proyecto es crear una aplicación definitiva que sea práctica y útil para su propósito: generar un sistema RAG que funcione como un modelo de lenguaje capaz de responder de forma eficaz a consultas relativas al síndrome de Rett, ejecutándose de manera offline en local.

Para ello será necesario recoger una lista de documentos relacionados con el tema lo más actualizados posibles, los cuales convertiremos a *embeddings* y guardaremos en una base de datos vectorial de FAISS mediante el uso de un script de *python*.

Posteriormente, se usará *Ollama* desde el subsistema de Linux para Windows (WSL) para poder usar la GPU local. Desde *Ollama* descargaremos el LLM *Mistral-7b* y lo usaremos en conjunción con la base de datos vectorial, para apoyar al modelo con documentos relevantes y la librería *Chainlit*, la cual nos dará una interfaz gráfica de página web para poder trabajar fácilmente con la aplicación.

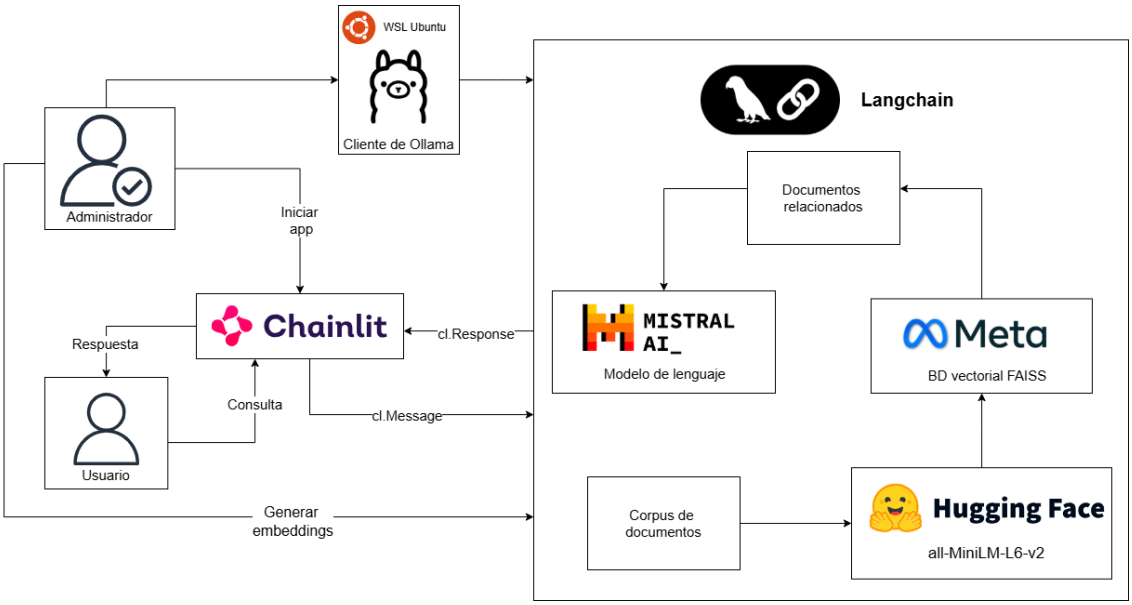


Ilustración 10. Diagrama de flujo de la aplicación definitiva

21.2 Selección de componentes del pipeline

Para poder construir nuestra aplicación RAG, debemos seleccionar el conjunto o corpus de documentos sobre los cuales se apoyará nuestro modelo de lenguaje para darnos respuestas más eficientes, así como el modelo de lenguaje en sí mismo, la estructura de sus *embeddings* y la base de datos de vectores donde alojarlos.
El modelo escogido para generar los *embeddings* de la aplicación *es* ***all-MiniLM-L6-v2***, de código abierto, extraído a partir de la comunidad ***HuggingFace*** (HuggingFace, 2024). Se basa en la arquitectura *sentence-transformers*, emplea 384 dimensiones para generar los vectores, y pesa únicamente 80Mb.

A partir de los datos resultantes en el estudio realizado en (Sbert, 2025), podemos extraer que este modelo es el más equilibrado, ofreciendo una calidad peor que otros modelos como *all-mpnet-base-v2*, pero siendo 5 veces más rápido.
Para el tamaño de *chunks* y el solapamiento entre ellos, se ha escogido el estándar publicado en el estudio realizado en (Wang, y otros, 2024), que es de 512 *tokens* por *chunk*, con un solapamiento de 20 *tokens*.

En cuanto a la base de datos vectorial, se ha decidido usar ***FAISS*** debido a su sinergia con el resto de librerías y su facilidad para usarse una vez generada, sin tener que gastar recursos computacionales cada vez que quiera emplearse. Además, se puede usar totalmente en local, y funciona mediante archivos, por lo que es fácilmente exportable de un equipo a otro.

En este caso particular, resulta más interesante que usar Milvus, la cual requiere el uso de servicios de contenedores externos, como Docker o Kubernetes, por lo que no es tan fácilmente exportable, pero además FAISS funciona mejor con proyectos a una escala pequeña, como este, mientras que la gran ventaja de Milvus es su facilidad para el escalado de proyectos a un nivel mucho más extenso.

En cuanto al volumen de *chunks* recuperados por consulta, se ha seguido el estándar recomendad para este tipo de proyectos, y se ha establecido en 4. Es decir, en cualquier pregunta, se seleccionarán los 4 documentos más relevantes, y se anexionarán a nuestra consulta. En un principio, se aconseja modificar este volumen si no se consiguen los resultados apropiados ya que, si es demasiado escaso, nuestro modelo no recibirá suficiente información relevante, pero si es demasiado extenso, recibirá demasiado ruido y las respuestas no serán acertadas.

Lógicamente, resulta interesante el uso de librerías externas que permitan gestionar y enlazar los distintos servicios que se usan en la aplicación. En este caso, la librería principal elegida es Langchain (más información en 21.4 Framework para diseño del RAG: LangChain), ya que permite integración con una variedad de servicios y es fácil de implementar debido a sus abstracciones como los *prompt templates*.

21.3 Selección del corpus de documentos

Para poder recoger una gran cantidad de datos médicos relevantes y estudios actuales, hemos decidido usar PubMed como motor de búsqueda de artículos médicos.

Pubmed es una base de datos de libre acceso que permite buscar revistas médicas de la base de datos MEDLINE y otros repositorios. Debido a su eficiencia de búsqueda, su cobertura temática, su rigurosidad y constante actualización, resulta el motor de búsqueda más empleado por los expertos médicos.
Para poder usar al máximo provecho un sistema RAG, resulta interesante tocar un tema muy concreto y del cual no haya extensa información. El síndrome de Rett es un buen tema que tratar, debido a que es una enfermedad rara, un trastorno genético neurológico que provoca la pérdida progresiva de las capacidades motoras y del habla, y afecta principalmente a niñas. Este síndrome no tiene cura, pero en la actualidad se siguen estudiando tratamientos efectivos para mejorar el movimiento, la comunicación y suavizar el efecto de las convulsiones. Precisamente, es debido a estos estudios recientes por lo que un sistema RAG es útil, ya que algunos modelos más típicos, como *gpt-3.5*, no tienen acceso a toda la información reciente.

En nuestro caso, hemos establecido un filtro de búsqueda con los siguientes parámetros. Queremos recoger artículos relacionados con el síndrome de Rett (*Rett syndrome*), que hayan sido publicados en los últimos 5 años, que el artículo completo esté disponible de forma gratuita, y seleccionamos como tipo de artículo Informes de casos, Estudio clínico, Prueba clínica, Revisión y Revisión sistemática.

Ordenamos los resultados para obtener al principio de la lista los artículos más recientes, y recogemos los 20 primeros. Esto nos da como resultado el siguiente enlace: https://pubmed.ncbi.nlm.nih.gov/?term=rett+syndrome&filter=datesearch.y_5&filter=simsearch2.ffrft&filter=pubt.casereports&filter=pubt.clinicalstudy&filter=pubt.clinicaltrial&filter=pubt.review&filter=pubt.systematicreview&sort=date

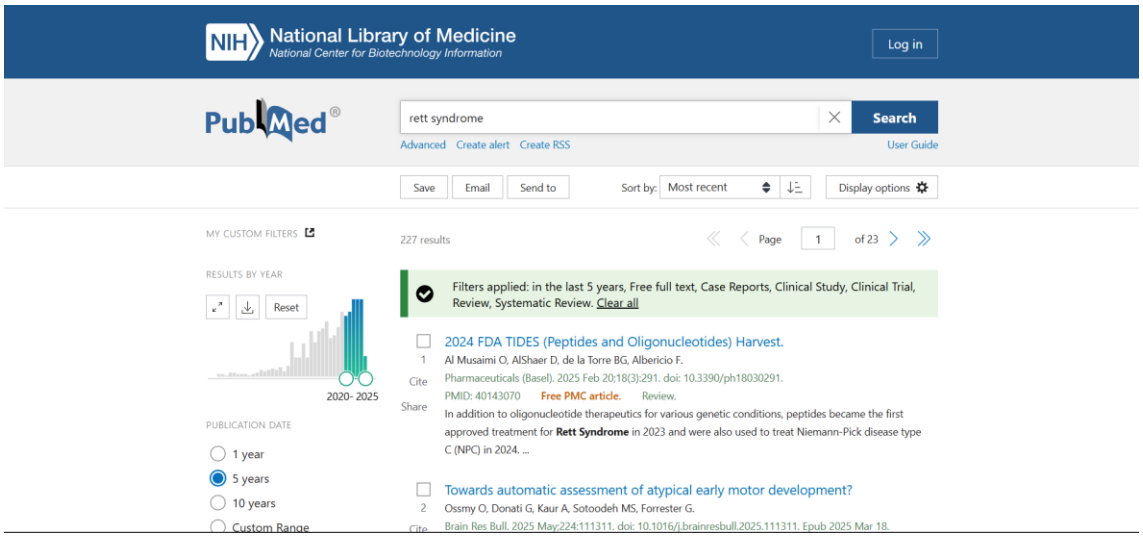


Ilustración 11. Resultado de la búsqueda con los parámetros especificados en PUBMED

21.4 Framework para diseño del RAG: LangChain

Langchain es un *framework* diseñado para desarrollar aplicaciones que usan *LLMs* (modelos de lenguaje a gran escala).
Langchain es útil para simplificar cada fase de producción de una aplicación que use *LLM*, tanto para el desarrollo, como la producción y el despliegue, ya que permite analizar la eficiencia de nuestras aplicaciones, y permite convertir nuestras aplicaciones en *APIs*, aunque en este documento nos centraremos en la parte del desarrollo[2] .

La construccion de aplicaciones mediante *Langchain* se construye por medio de cadenas *(chains)*, que son el principio fundamental que sustenta la construcción de objetos más complejos en *Langchain*. Una cadena es una serie de componentes que funcionan en secuencia para integrar LLMs junto a otros elementos para lograr una tarea más compleja.
A continuación, se enumeran los principales componentes que disponibles en una aplicación Langchain:
- Interfaz LLM: Langchain proporciona una API de fácil uso que los desarrolladores pueden usar para conectar una gran variedad de modelos LLM con el código de nuestra aplicación.
- Plantillas de peticiones *(prompt template)*: Consisten en estructuras de peticiones prediseñadas que los desarrolladores pueden usar para formatear de una forma adecuada para los LLMs, además de poder construir plantillas en las que luego sustituir palabras clave con elementos de nuestro código.
- Agentes *(agents)*: Un *agente* es una cadena especial que hace que el modelo de lenguaje decida la mejor secuencia en respuesta a una consulta. Al utilizar un agente, los desarrolladores proporcionan las entradas del usuario, las herramientas disponibles y los posibles pasos intermedios para lograr los resultados deseados. Luego, el modelo de lenguaje devuelve una secuencia viable de acciones que la aplicación puede realizar.
Por ejemplo, es bien sabido que los LLMs tienen dificultades para responder a preguntas que requieran conocimientos de aritmética, pero podemos usar Langchain para corregir

[2]https://python.langchain.com/docs/introduction

esto, implementando una función calculadora para apoyar al LLM. A continuación, veremos mediante diversos fragmentos de código como implementarlo.

```
@tool
def calculadora(query: str) -> str:
    """Resuelve operaciones matemáticas simples."""
    try:
        return str(eval(query))
    except Exception as e:
        return f"Error: {str(e)}"

tools = [calculadora]
```

En este fragmento de código se crea una función calculadora que servirá para devolver la solución a una operación aritmética se usa el decorador *@tool* y se añade a una lista de herramientas (*tools*) que usaremos más tarde.

```
llm = ChatOpenAI(
    temperature=0,
    model_name="gpt-3.5-turbo"
)

agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True
)
```

Luego seleccionamos el modelo de lenguaje *gpt-3.5-turbo* e inicializaremos el agente con los parámetros establecidos. Entre los parámetros definidos se encuentra la variable *booleana verbose*, que, si activamos a verdadero, nos describirá el proceso mientras lo realiza.

```
respuesta = agent.run("Calcula la raíz cuadrada de 144, luego súmale 3 y
multiplícalo por 7")
print(respuesta)
```

Lo que queda después es ejecutar el agente y comprobar su salida:

```
> Entering new AgentExecutor chain...
Primero calcularemos la raíz cuadrada de 144, luego sumaremos 3 y finalmente
multiplicaremos por 7.
Action: calculadora
Action Input: "sqrt(144)"
Observation: Error: name 'sqrt' is not defined
Thought:It seems that the calculator function does not recognize the "sqrt"
function. We can instead calculate the square root by raising 144 to the power of
0.5.
Action: calculadora
Action Input: "144**0.5"
Observation: 12.0
Thought:Ahora que tenemos la raíz cuadrada de 144, podemos continuar con los
siguientes pasos.
Action: calculadora
Action Input: "12.0 + 3"
```

```
Observation: 15.0
Thought:Hemos sumado 3 a la raíz cuadrada de 144. Ahora multiplicaremos por 7.
Action: calculadora
Action Input: "15.0 * 7"
Observation: 105.0
Thought:Hemos completado todas las operaciones solicitadas.
Final Answer: 105.0

> Finished chain.
105.0
```

- Memoria: Algunas aplicaciones de modelos de lenguaje conversacional refinan sus respuestas con información que se recupera de interacciones anteriores, como por ejemplo *Chatbots*, es decir, aplicaciones en las que se pone a disposición un modelo de lenguaje y, guardando el histórico de mensajes tanto del usuario como del modelo, se puede usar como agente conversacional. LangChain permite a los desarrolladores incluir capacidades de memoria en sus sistemas.
- Devoluciones de llamada *(callbacks)*: Son códigos que los desarrolladores pueden colocar en sus aplicaciones para registrar, monitorear y transmitir eventos específicos en las operaciones de Langchain. Por ejemplo, los desarrolladores pueden rastrear cuándo se llamó por primera vez a una cadena y si se encontraron errores.
Y por supuesto, lo que resulta más interesante para esta aplicación son las herramientas que ofrece Langchain para **Recuperación aumentada** o **modelos RAG**. Entre ellas encontramos herramientas para **cargar documentos de diversos formatos** (pdf, txt, csv, docx, etc..), ***text splitters*** para dividir los documentos en *chunks*, integración con diversos algoritmos de ***embeddings*** y **bases de datos vectoriales**, que son tremendamente útiles para integrar todas estas tecnologías en un flujo de trabajo eficaz[3] .

21.5 Generación de Embeddings. Huggingface.

Una vez hayamos construido el corpus de documentos que se suministrarán al sistema RAG, debemos convertirlos a un formato de *embeddings* para que el modelo de lenguaje lo comprenda. Esto se realiza en un script de Python independiente al *script* en el que se usa el modelo, para no tener que realizar la conversión cada vez que quiera usarse el sistema.

Langchain, entre sus múltiples funciones, nos permite usar una libería auxiliar llamada *PyPDF* para convertir los documentos PDF fácilmente, como se puede apreciar en el siguiente fragmento de código:

```
dir="./pdf"

from langchain_community.document_loaders import PyPDFLoader, DirectoryLoader
loader = DirectoryLoader(
    dir,
    glob="**/*.pdf",
    loader_cls=PyPDFLoader,
)

docs = loader.load()
```

[3] https://aws.amazon.com/what-is/langchain

En este código cargamos en la aplicación todos los documentos en formato PDF del directorio especificado.

Una vez se hayan cargado los documentos, debemos convertirlos al formato de *embeddings* con los parámetros de *chunking* explicados anteriormente.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(chunk_size=512, chunk_overlap=20)
split_docs = splitter.split_documents(docs)

texts = [doc.page_content for doc in split_docs]

embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-
v2")

from langchain_community.vectorstores import FAISS
db = FAISS.from_texts(texts, embeddings)
db.save_local("faiss_index")
```

Para ello usamos el modelo *all-miniLM-L6*-v2, de Huggingface, como ya vimos en la sección Selección de componentes del pipeline, para luego guardarlos en una base de datos vectorial en FAISS.

21.6 Construcción. Uso de Ollama.

Una vez generados los embeddings y guardados en la base de datos vectorial de FAISS, es necesario importarlos a otro archivo ejecutable en el que se usará modelo de lenguaje local. Sin embargo, es necesario usar una plataforma en la que descargar el modelo y ejecutarlo cuando se requiera. Para ello hemos decidido usar Ollama[4].

Este consiste en una herramienta de código abierto que permite ejecutar modelos de lenguaje de forma local fácilmente. En este proyecto también se ha usado el subsistema de Linux para Windows, WSL, con una distribución Ubuntu, para que Ollama pueda acceder a la GPU del equipo, y para ello también es necesario el kit de herramientas CUDA de Nvidia. Para instalarlo desde el subsistema de Windows para Linux, solo se requiere escribir este comando en la terminal de Ubuntu.

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

Después es necesario seleccionar el modelo de lenguaje, que en este caso será Mistral-7B. Una vez se lanza el modelo de lenguaje, Ollama está disponible como servicio en el puerto 11434 del equipo local, al cual accederemos dentro del código cuando se requiera usar el modelo de lenguaje.

21.7 Uso de LLM Open Source. Mistral.

[4] https://ollama.com/library/mistral-small3.1

En *Huggingface* (Open_llm_leaderboard, 2025), uno de los sitios web más importantes en el campo de la computación y la inteligencia artificial, ofrecen una clasificación disponible para todo el mundo de los LLM más usados y en *Ollama* (Library, s.f.), que es la plataforma donde se pueden descargar LLM locales, también disponemos de una lista de los modelos de lenguaje más populares entre los usuarios.

En estas listas, podemos encontrar a Mistral-7B, el cual goza de unas 13 millones de descargas desde el sitio web de *Ollama*, dato registrado en agosto de 2024.
Mistral es un LLM diseñado por Mistral AI, una startup francesa dedicada a la inteligencia artificial en 2023.
Su estrategia de diseño destaca por la intención de desarrollar un producto eficiente y utilizable por el usuario particular, no necesariamente por un sistema con muchos recursos (Acerca de, s.f.).
Esto se aprecia claramente si observamos las características de este modelo: utiliza únicamente 7 mil millones de parámetros, 32 capas, una ventana de atención de 4096 tokens y un contexto total de 8192 tokens.
En el estudio realizado por (Q. Jiang, y otros, 2023), se realizó una comparativa con otros LLM, LLaMa 2(7B y 13B), LLaMa 1(34B) y Code-LLaMa en diversas tareas como: razonamiento, conocimiento global, matemáticas, programación, comprensión lectora, etc...
Mistral acabó superando a estos modelos, aunque algunos tienen el triple de parámetros. Es por su eficacia relativa a su tamaño lo que lo hace muy interesante para proyectos como éste, donde no disponemos de grandes sistema distribuidos para manejar una alta complejidad de análisis de consultas.

21.8 Interfaz de usuario del sistema RAG. Chainlit.

Chainlit es una librería de código abierto en *python* diseñada para crear interfaces página web de forma sencilla para aplicaciones de chat de inteligencia artificial, usando librerías auxiliares para trabajar con modelos de lenguaje, como *Langchain*, *Langsmith*, *LlamaIndex*, etc...
Es ampliamente personalizable, tanto en la lógica de la aplicación como en la interfaz de esta, y se despliega por medio del navegador, la cual permanece disponible para su uso en el puerto 8000 del equipo local.
En el aspecto gráfico de la interfaz, se puede personalizar mediante CSS, además de poder modificar las imágenes e iconos de la aplicación. La interfaz dispone de 2 modos de presentación: oscuro y claro, los cuales también son personalizables mediante la creación y modificación de un archivo *theme.json*.

Ilustración 12. Interfaz de bienvenida de Chainlit

En cuanto a la lógica de la aplicación, hay algunos aspectos que son modificables mediante JavaScript, como botones adicionales o integrarla con otras aplicaciones web, pero la gran mayoría de funciones internas, se modifican mediante código en *python*.
Para construir una aplicación de *Chainlit*, es necesario crear un archivo de *python* para su posterior ejecución. *Chainlit* presenta una serie de decoradores para definir funciones ligadas a eventos. Los más importantes son *@cl.on_chat_start* , que define el comportamiento al iniciar la aplicación y *@cl.on_message,* que define el comportamiento de la aplicación el recibir un mensaje.
En realidad, *Chainlit* no requiere necesariamente el uso de un modelo de lenguaje, ya que se pueden definir funciones como la siguiente, que responde de una forma u otra según el contenido del mensaje del usuario:

```
@cl.on_message
async def handle_message(message: cl.Message):
    user_input = message.content

    if user_input.lower() == "hola":
        respuesta = "¡Hola! ¿Cómo estás?"
    elif user_input.lower() == "adiós":
        respuesta = "¡Hasta luego!"
    else:
        respuesta = f"No entendí: '{user_input}'"

    await cl.Message(content=respuesta).send()
```

A continuación, se explicará el código definitivo usando *Ollama* y los *embeddings* de FAISS.

```
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-
v2")

db = FAISS.load_local("faiss_index", embeddings,
allow_dangerous_deserialization=True)

retriever = db.as_retriever(search_type="similarity", search_kwargs={"k": 4})
```

En estas 3 primeras líneas, se definen los mismos *embeddings* que en el código anterior, se extraen los mismos desde el directorio *faiss_index*, y se establece el recuperador para que recoja los 4 *chunks* más relevantes.
En el siguiente código definimos un *prompt template*, es decir una plantilla para que nuestros prompts siempre tengan la siguiente estructura.

```
prompt_template = ChatPromptTemplate.from_messages([
    ("system",
        "Eres un asistente de IA que responde correctamente y en español. "
        "Si no sabes la respuesta, di 'No tengo información sobre su pregunta'. "
        "Usa la siguiente información relevante para responder la
pregunta:\n\n{context}"),
    ("human", "{history}\nUsuario: {query}")
])
```

Las palabras clave *query, context* y *history* se definirán posteriormente en el método para definir el comportamiento de la aplicación al recibir un mensaje.
El siguiente fragmento de código se ejecuta cuando se inicia la aplicación.

```
@cl.on_chat_start
async def on_chat_start():
    model = ChatOllama(
        base_url="http://localhost:11434
        model="mistral",
        temperature=0.0,
        max_tokens=500
    )
    runnable = prompt_template | model | StrOutputParser()
    cl.user_session.set("runnable", runnable)
```

Aquí se define el modelo de lenguaje, que es *Mistral*, que se recoge a partir el puerto 11434 de nuestro subsistema Ubuntu, y sus parámetros, como la temperatura y el máximo de tokens por respuesta.
También se establecen el modelo y el *prompt template* como *runnable*, es decir el ejecutable que usará la aplicación para construir la respuesta.
Este es el código que se ejecutará cada vez que el usuario envíe un mensaje.

```
@cl.on_message
async def on_message(message: cl.Message):
    runnable = cast(Runnable, cl.user_session.get("runnable"))

    history = cl.user_session.get("history", [])

    history_str = "\n".join(
        [f"{'Usuario' if m['role']=='user' else 'Asistente'}: {m['content']}" for m
in history]
    )

    query = message.content
    docs = retriever.invoke(query)
    context = "\n\n".join([doc.page_content for doc in docs])

    msg = cl.Message(content="")
    response_text = ""
    async for chunk in runnable.astream(
```

```
        {"query": query, "context": context, "history": history_str},
        config=RunnableConfig(callbacks=[cl.LangchainCallbackHandler()]),
    ):
        response_text += chunk
        await msg.stream_token(chunk)
    await msg.send()

    history.append({"role": "user", "content": message.content})
    history.append({"role": "assistant", "content": response_text})
    cl.user_session.set("history", history)
```

Lo primero, es recoger el *runnable* a partir del código anterior. Se define una variable *query* para sustituirla en el *prompt template*, que guardará el contenido del mensaje.
Creamos una lista de string llamada *docs*, a partir del recuperador, que transformará nuestra consulta a un formato de embeddings, lo comparará con los embeddings que generamos en el fichero anterior y nos devolverá los 4 *chunks* más cercanos o similares. Luego guardamos esta lista en un único *string* llamado *context*, que introduciremos en el *prompt template*.

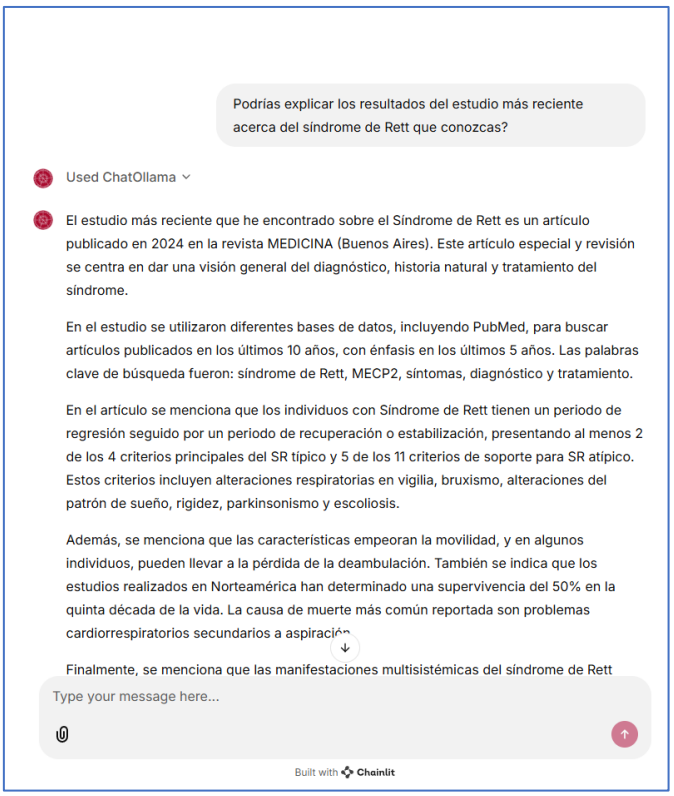


Ilustración 13. Demostración de la interfaz de Chainlit mediante mensaje

También es importante el uso de la clase *history* en el *prompt template*, ya que por defecto, el modelo de lenguaje no tiene en cuenta los mensajes anteriores en la conversación. Para que funcione como un *Chatbot*, es necesario generar un historial de mensajes e insertarlos en el *prompt template* para que acceda en cada consulta.

Para generar el historial, se recoge una lista a partir del historial de usuario de chainlit, y luego se crea un único string que se sustituirá por la clave *history*, con un formato tipo:

```
"Usuario: ¿Qué es la fotosíntesis?\n
Asistente: La fotosíntesis es un proceso mediante el cual...\n
```

A continuación, definimos un mensaje vacío, y configuramos el *runnable*, es decir, el ejecutable de la aplicación, para que genera cada *token* de la respuesta acorde a la entrada, es decir, nuestro *prompt template* personalizado, y a continuación lo envíe a la interfaz gráfica.
En cada consulta, se guardan los valores de la consulta del usuario y la respuesta del modelo para mantener el historial de la conversación.

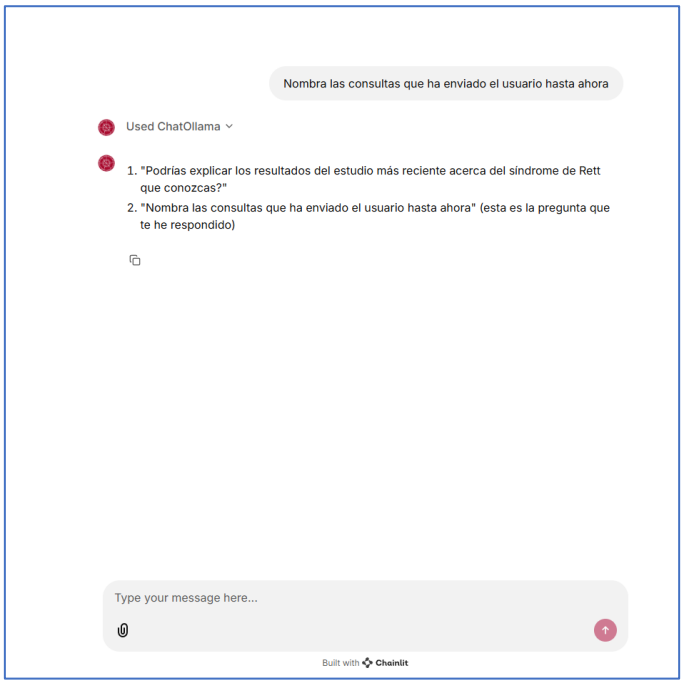


Ilustración 14. Demostración del histórico de Chainlit

21.9 Evaluación de resultados. RAGAS.

Uno de los objetivos iniciales era corroborar cuantitativamente la hipótesis de que un RAG mejora la eficacia de un LLM en contextos altamente especializados o actuales. Para ello existen librerías que nos ayudan en el proceso de evaluación.

RAGAS[5] es una librería de Python diseñada para evaluar las características de los sistemas RAGs usando métricas, para asegurarnos que la recuperación y la generación funcionan de manera óptima. También es capaz de generar *datasets* basados en consultas y respuestas a partir de documentos de entrada.

Mediante el uso de diferentes tipos de consultas, ya sean ***single-hop*** (preguntas basadas en un hecho específico) o ***multi-hop*** (preguntas más complejas en las que se necesitan usar varios documentos para encontrar la respuesta), RAGAS nos ayuda a identificar áreas de mejora o comparar diferentes modelos o sistemas entre sí.

[5] https://docs.ragas.io/en/stable/

1) Métricas de RAGAS:

- Context precision (precisión de contexto): evalúa cómo de eficiente es el sistema para organizar los documentos de forma que los más relevantes se encuentren al principio de cola de recuperación.
Para calcular esta métrica, se utiliza otra métrica auxiliar llamada **Precision@K**, que mide la proporción de fragmentos relevantes respecto al total de fragmentos recuperados en cada posición del ranking. RAGAS emplea modelos de lenguaje para identificar afirmaciones relevantes dentro de los documentos recuperados, comparándolas con la consulta del usuario. A cada resultado se le asigna un indicador de relevancia -1 si es relevante y 0 si no lo es.
La puntuación general de precisión de contexto se obtiene promediando estos valores de precisión ponderados entre los K mejores resultados. Esta métrica garantiza que los usuarios reciban la información más útil de la manera más accesible.

$$\text{Context Precision@K} = \frac{\sum_{k=1}^{K} (\text{Precision@k} \times v_k)}{\text{Total number of relevant items in the top } K \text{ results}}$$

$$\text{Precision@k} = \frac{\text{true positives@k}}{(\text{true positives@k} + \text{false positives@k})}$$

Where K is the total number of chunks in `retrieved_contexts` and $v_k \in \{0, 1\}$ is the relevance indicator at rank k.

Ilustración 15. RAGAS: Fórmula de Context Precision (Young, 2024)

- Context recall (recuperación de contexto): mide la eficacia del sistema a la hora de recuperar los documentos más relevantes con respecto a la consulta dentro de los documentos aportados. RAGAS compara el contexto recuperado con la GT o *ground truth* (verdad absoluta) para calcular esta métrica.

Para cada afirmación de la GT, el sistema verifica si puede inferirse a partir del contenido recuperado. La puntuación de recuperación de contexto se calcula como la proporción de afirmaciones relevantes encontradas en el contenido recuperado respecto al total de afirmaciones en la GT. Esta métrica es especialmente importante en campos como derecho, medicina e investigación, donde se requiere que la información sea lo más precisa posible.

$$\text{context recall} = \frac{|\text{GT claims that can be attributed to context}|}{|\text{Number of claims in GT}|}$$

Ilustración 16. RAGAS: Fórmula de Context Recall (Young, 2024)

- Response relevancy (relevancia de la respuesta): calcula la similitud entre la respuesta generada y la consulta del usuario. Esta métrica asegura que la respuesta es útil y que no se expone información innecesaria. RAGAS usa LLMs para generar un conjunto de preguntas artificiales a partir de la respuesta del sistema.

Estas preguntas se contruyen mediante un proceso de ingeniería inversa para simular cual podría ser la consulta original. La similitud coseno entre estas consultas generadas y las consultas originales se miden para calcular la puntuación de relevancia.
Una puntuación alta significa que la respuesta efecivamente se corresponde con las necesidades del usuario, haciendo que la métrica sea particularmente útil en aplicaciones como servicio al consumidor, donde las respuestas concisas y relevantes son cruciales.

$$\text{answer relevancy} = \frac{1}{N}\sum_{i=1}^{N} cos(E_{g_i}, E_o)$$

$$\text{answer relevancy} = \frac{1}{N}\sum_{i=1}^{N} \frac{E_{g_i} \cdot E_o}{\|E_{g_i}\|\|E_o\|}$$

Ilustración 17. RAGAS: Fórmula de Response Relevancy (Young, 2024)

- Faithfulness (fidelidad): asegura que la respuesta generada se mantiene rigurosa y no genera alucinaciones o contradicciones. RAGAS divide la respuesta generada en varias afirmaciones y las contrasta con el texto original usando LLMs para comprobar si se puede extraer la respuesta a partir del documento original.
Por cada afirmación, el sistema determine si se puede inferir a partir del contexto generado. La puntuación se calcula mediante la proporción de afirmaciones verificadas frente al número total de afirmaciones en la respuesta. Esta métrica es clave en campos como periodismo y finanzas, donde la rigurosidad es muy importante.

$$\text{Faithfulness score} = \frac{|\text{Number of claims in the generated answer that can be inferred fror}}{|\text{Total number of claims in the generated answer}|}$$

Ilustración 18. RAGAS: Fórmula de Faithfulness (Young, 2024)

- Noise sensitivity (sensibilidad al ruido): calcula la frecuencia del sistema de generar errores dando respuestas incorrectas, ya sea basándose en documentos relevantes o irrelevantes. Cuanto menor sea el valor de esta métrica, mejor será el rendimiento. Esta métrica resulta crucial para asegurar que la aplicación no sea susceptible a información falsa.

Para calcular esta métrica, el sistema comprueba cada afirmación en la respuesta generada frente a la GT (verdad absoluta) para determinar si es preciso y si se corresponde con los contextos recuperados más relevantes. Un sistema ideal tendría todas las consultas apoyadas por contextos relevantes, reduciendo las afirmaciones incorrectas.

$$\text{noise sensitivity (relevant)} = \frac{|\text{Total number of incorrect claims in response}|}{|\text{Total number of claims in the response}|}$$

Ilustración 19. RAGAS: Fórmula de Noise Sensitivity (Young, 2024)

- Factual correctness (rigurosidad con los hechos): sirve para comparar y evaluar la similitud entre una respuesta generada y una respuesta de referencia. Esta métrica es útil para poder conocer cuán cercana es la respuesta generada con respecto a la información que se requiere para responder. RAGAS la calcula dividiendo la respuesta y la referencia entre diversas afirmaciones y las compara por medio de un LLM.

Para ello se apoya en falsos positivos (cantidad de afirmaciones en la respuesta que no aparecen en la referencia), positivos verdaderos (cantidad de afirmaciones en la respuesta que aparecen en la referencia), y falsos negativos (cantidad de afirmaciones en la referencia que no están presentes en la respuesta). Una puntuación más alta equivale a una mayor precisión del modelo y menor riesgo de respuestas incorrectas.

$$\text{True Positive (TP)} = \text{Number of claims in response that are present in reference}$$

$$\text{False Positive (FP)} = \text{Number of claims in response that are not present in reference}$$

$$\text{False Negative (FN)} = \text{Number of claims in reference that are not present in response}$$

The formula for calculating precision, recall, and F1 score is as follows:

$$\text{Precision} = \frac{TP}{(TP + FP)}$$

$$\text{Recall} = \frac{TP}{(TP + FN)}$$

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{(\text{Precision} + \text{Recall})}$$

Ilustración 20. RAGAS: Fórmula de Factual Correctness (Young, 2024)

Una vez explicados los conceptos básicos de la librería RAGAS, lo siguiente es ponerlo en práctica.

2) Código necesario para la evaluación y resultados consiguientes:

Este apartado tratará de explicar cómo construir un módulo de RAGAS para evaluar las capacidades de nuestro sistema RAG en un ***notebook*** de *Google Colab*, para evitar problemas relacionados con incompatibilidad de librerías en el equipo local.

Después de importar las librerías pertinentes en el entorno virtual, y cargar el modelo de lenguaje y modelo de *embeddings* que usamos anteriormente, escribiremos las siguientes líneas de código para generar de manera automática nuestro ***testset***.

```
import nest_asyncio
nest_asyncio.apply()

from ragas.testset import TestsetGenerator

generator = TestsetGenerator(llm=generator_llm,
embedding_model=generator_embeddings)
dataset = generator.generate_with_langchain_docs(docs, testset_size=30)
```

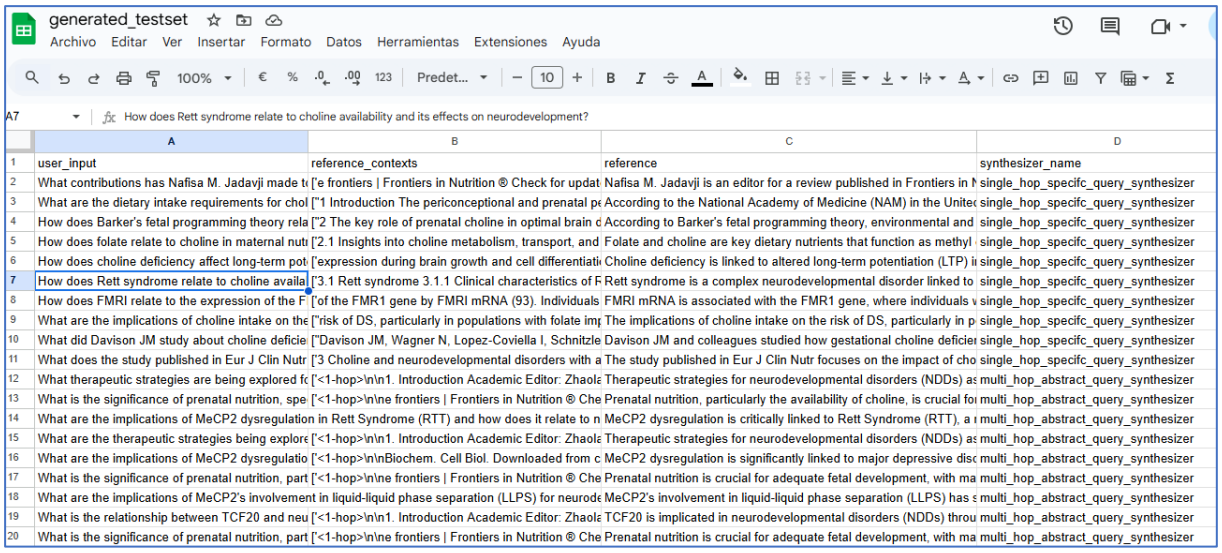
Esto consiste en un archivo *.csv* con 4 columnas: *user_input, reference contexts, reference, y synthesizer_name*, que se explicarán en mayor detalle, generado a partir del modelo de lenguaje escogido, los documentos que representan el corpus, y el modelo de *embeddings* utilizado. Tal y como aparece en la última línea del fragmento de código, se especifica que se generen 30 tuplas.

- User input (entrada del usuario): Es el *prompt* que introduciría el usuario para obtener las siguientes respuestas. Ejemplo: *¿Cómo se relaciona el síndrome de Rett con la disponibilidad de colina y sus efectos en el neurodesarrollo?*

- Reference contexts (contexto de referencia): Los fragmentos del corpus que deberían ser recuperados para responder de forma adecuada a la pregunta. Es la *ground truth* o verdad absoluta sobre la que se espera que se base la respuesta del modelo de lenguaje. Ejemplo: *[…] 3.1.3 La colina y su posible interacción con el síndrome de Rett*
Una baja disponibilidad de colina altera la metilación global de las islas CpG en el cerebro en desarrollo. La metilación de los CpG es llevada a cabo de forma de novo por la metiltransferasa de ADN 3A, con estabilización por las metiltransferasas 3L y 3B (DNMT3A/3B) […].

- Reference (referencia): La respuesta esperada que el modelo de lenguaje debería dar en una situación ideal dado el contexto recuperado. Ejemplo: *El síndrome de Rett es un trastorno neurodesarrollativo complejo asociado a mutaciones en el gen MECP2, el cual es fundamental para el desarrollo cerebral. Una baja disponibilidad de colina puede alterar la metilación global de las islas CpG en el cerebro en desarrollo, afectando la expresión génica. [...]*

- Synthesizer name (nombre del sintetizador): Se refiere al tipo de consulta generada de forma automática. Se catalogan en 4 tipos distintos en función de si son consultas que requieran respuestas de datos concretos, o si requieren una respuesta más abstracta *(specific vs abstract)* y también en función de si requieren información de una única fuente, o si por el contrario requieren varias *(single_hop vs multi_hop)*. Ejemplo: *single_hop_specific_query_sinthesizer*



generated_testset

Archivo Editar Ver Insertar Formato Datos Herramientas Extensiones Ayuda

A7 How does Rett syndrome relate to choline availability and its effects on neurodevelopment?

	A	B	C	D
1	user_input	reference_contexts	reference	synthesizer_name
2	What contributions has Nafisa M. Jadavji made t	['e frontiers | Frontiers in Nutrition ® Check for updat	Nafisa M. Jadavji is an editor for a review published in Frontiers in N	single_hop_specifc_query_synthesizer
3	What are the dietary intake requirements for chol	["1 Introduction The periconceptional and prenatal pe	According to the National Academy of Medicine (NAM) in the Unite	single_hop_specifc_query_synthesizer
4	How does Barker's fetal programming theory rela	["2 The key role of prenatal choline in optimal brain d	According to Barker's fetal programming theory, environmental and	single_hop_specifc_query_synthesizer
5	How does folate relate to choline in maternal nut	['2.1 Insights into choline metabolism, transport, and	Folate and choline are key dietary nutrients that function as methyl	single_hop_specifc_query_synthesizer
6	How does choline deficiency affect long-term pot	['expression during brain growth and cell differentiati	Choline deficiency is linked to altered long-term potentiation (LTP) i	single_hop_specifc_query_synthesizer
7	How does Rett syndrome relate to choline availa	['3.1 Rett syndrome 3.1.1 Clinical characteristics of R	Rett syndrome is a complex neurodevelopmental disorder linked to	single_hop_specifc_query_synthesizer
8	How does FMRI relate to the expression of the F	['of the FMR1 gene by FMRI mRNA (93). Individuals	FMRI mRNA is associated with the FMR1 gene, where individuals v	single_hop_specifc_query_synthesizer
9	What are the implications of choline intake on the	["risk of DS, particularly in populations with folate im	The implications of choline intake on the risk of DS, particularly in p	single_hop_specifc_query_synthesizer
10	What did Davison JM study about choline deficie	["Davison JM, Wagner N, Lopez-Coviella I, Schnitzle	Davison JM and colleagues studied how gestational choline deficier	single_hop_specifc_query_synthesizer
11	What does the study published in Eur J Clin Nutr	['3 Choline and neurodevelopmental disorders with a	The study published in Eur J Clin Nutr focuses on the impact of cho	single_hop_specifc_query_synthesizer
12	What therapeutic strategies are being explored fo	['<1-hop>\n\n1. Introduction Academic Editor: Zhaole	Therapeutic strategies for neurodevelopmental disorders (NDDs) a	multi_hop_abstract_query_synthesizer
13	What is the significance of prenatal nutrition, spe	['<1-hop>\n\ne frontiers | Frontiers in Nutrition ® Che	Prenatal nutrition, particularly the availability of choline, is crucial fo	multi_hop_abstract_query_synthesizer
14	What are the implications of MeCP2 dysregulation in Rett Syndrome (RTT) and how does it relate to n		MeCP2 dysregulation is critically linked to Rett Syndrome (RTT), a	multi_hop_abstract_query_synthesizer
15	What are the therapeutic strategies being explore	['<1-hop>\n\n1. Introduction Academic Editor: Zhaole	Therapeutic strategies for neurodevelopmental disorders (NDDs) a	multi_hop_abstract_query_synthesizer
16	What are the implications of MeCP2 dysregulatio	['<1-hop>\n\nBiochem. Cell Biol. Downloaded from c	MeCP2 dysregulation is significantly linked to major depressive disc	multi_hop_abstract_query_synthesizer
17	What is the significance of prenatal nutrition, part	['<1-hop>\n\ne frontiers | Frontiers in Nutrition ® Che	Prenatal nutrition is crucial for adequate fetal development, with ma	multi_hop_abstract_query_synthesizer
18	What are the implications of MeCP2's involvement in liquid-liquid phase separation (LLPS) for neurode		MeCP2's involvement in liquid-liquid phase separation (LLPS) has s	multi_hop_abstract_query_synthesizer
19	What is the relationship between TCF20 and neu	['<1-hop>\n\n1. Introduction Academic Editor: Zhaole	TCF20 is implicated in neurodevelopmental disorders (NDDs) throu	multi_hop_abstract_query_synthesizer
20	What is the significance of prenatal nutrition, part	['<1-hop>\n\ne frontiers | Frontiers in Nutrition ® Che	Prenatal nutrition is crucial for adequate fetal development, with ma	multi_hop_abstract_query_synthesizer

Ilustración 21. RAGAS: Extracto del Testset generado de forma automática.

Después de obtener el *dataset*, debemos comprobar cómo de eficaz es nuestro sistema RAG, tanto a la hora de buscar cuáles son los documentos más relevantes, para anexarlos al *prompt*, como para devolver una respuesta adecuada en función de la consulta proporcionada y los fragmentos de documento extraídos. Para ello sirve la siguiente celda del *notebook*:

```
dataset = []

for index, row in df.iterrows():
    query = row['user_input']
    reference = row['reference']

    relevant_docs = rag.get_most_relevant_docs(query)
    response = rag.generate_answer(query, relevant_docs)
    dataset.append(
        {
            "user_input":query,
            "retrieved_contexts":relevant_docs,
            "response":response,
            "reference":reference
        }
    )
```

Consiste en recorrer iterativamente la tabla del *dataset*, obtener las columnas *user_input* y *reference* de cada una de las 30 filas, utilizar el código del RAG para recuperar los documentos más relevantes, y generar una respuesta a partir de estos documentos. Se crea una nueva tupla con estos 4 datos, y se añade al final de la tabla.

Este es el *prompt* usado en la primera iteración de la evaluación por medio de RAGAS:

```
prompt_template = ChatPromptTemplate.from_messages([
        ("system",
          "Eres un asistente experto en temas médicos que responde siempre en
español. "
          "Debes usar únicamente la información proporcionada en el contexto para
responder. "
```

```
          "NO menciones artículos, fuentes, documentos ni autores. "
          "NO digas frases como 'el objetivo de este artículo' o 'según el
documento'. "
          "No digas frases como 'dentro del contexto proporcionado."
          "Responde de forma directa, como si el conocimiento fuera tuyo. "
          "Si no tienes información suficiente en el contexto, responde: 'No tengo
información sobre su pregunta'.\n\n"
          "Contexto:\n{relevant_doc}"),
        ("human", "\nUsuario: {query}")
      ])
```

Lo siguiente será obtener las métricas de RAGAS en función del desempeño de la aplicación RAG. Se debe usar un modelo de lenguaje auxiliar como evaluador, en este caso se ha optado por *gpt-4o-mini*,.

```
evaluation_dataset = EvaluationDataset.from_list(dataset)

llm = ChatOpenAI(model="gpt-4o-mini")

evaluation_dataset = EvaluationDataset.from_list(dataset)
evaluator_llm = LangchainLLMWrapper(llm)

result = evaluate(dataset=evaluation_dataset,metrics=[LLMContextRecall(),
Faithfulness(), FactualCorrectness(),LLMContextPrecisionWithReference(),
ResponseRelevancy(), NoiseSensitivity()],llm=evaluator_llm)
```

Estos son los resultados obtenidos de la primera iteración:

```
{'context_recall': 0.5539,
'faithfulness': 0.7634,
'factual_correctness(mode=f1)': 0.3947, 'llm_context_precision_with_reference':
0.8667,
'answer_relevancy': 0.6762,
'noise_sensitivity(mode=relevant)': 0.2553}
```

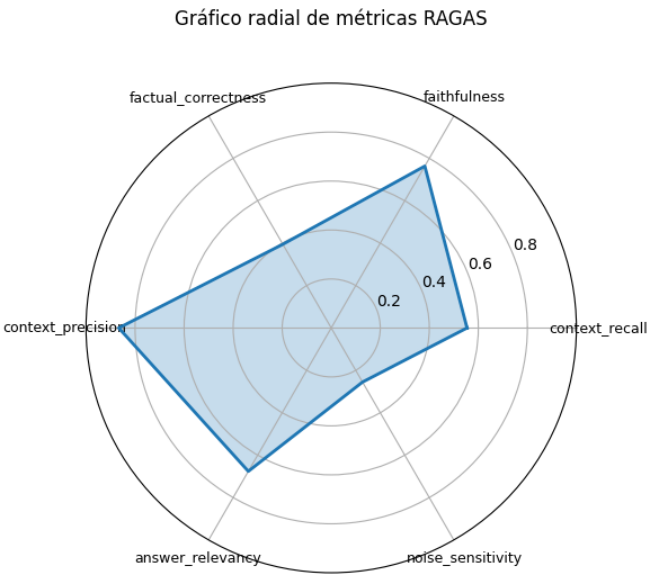


Ilustración 22. Gráfico radial de métricas RAGAS del sistema RAG (primera iteración).

Después de un proceso de optimización del sistema, se modificó el prompt para buscar una mejora de las respuestas generadas por el modelo de lenguaje, especialmente para que sea capaz de reflejar cuando la información suministrada no es suficientemente adecuada para responder de forma correcta y así evitar alucinaciones, pero al mismo no tiempo no cohibir al modelo y no obligarle a que sólo responda si el contexto es perfectamente óptimo, además del uso de etiquetas # para definir con mayor claridad los objetivos que se quieren conseguir.
En definitiva, este es el *prompt* final:

```
("system",
         "#rol Eres un experto asistente en temas médicos, especialmente en el
sindrome de Rett, que responde en español preguntas con fin pedagógico basándose en
el contexto."
         "Si los materiales no son relevantes o lo suficientemente completos para
responder con confianza la pregunta del usuario, la mejor respuesta es: 'los
materiales no parecen"
         "ser suficientes para proporcionar una buena respuesta."
         "#resultados esperados La información proporcionada debe ser clara,
concisa, precisa y actual y responder de manera directa como si el conocimiento
fuera tuyo."
         "#contexto Estos extractos de textos biomédicos en Pubmed son los más
recientes y relevantes para responder: {relevant_doc}"),
        ("human", "{query}")
```

Después de este cambio, los resultados obtenidos son los siguientes:

```
{'context_recall': 0.5622,
'faithfulness': 0.7608,
'factual_correctness(mode=f1)': 0.4993, 'llm_context_precision_with_reference':
0.8898,
'answer_relevancy': 0.6432,
'noise_sensitivity(mode=relevant)': 0.2414}
```

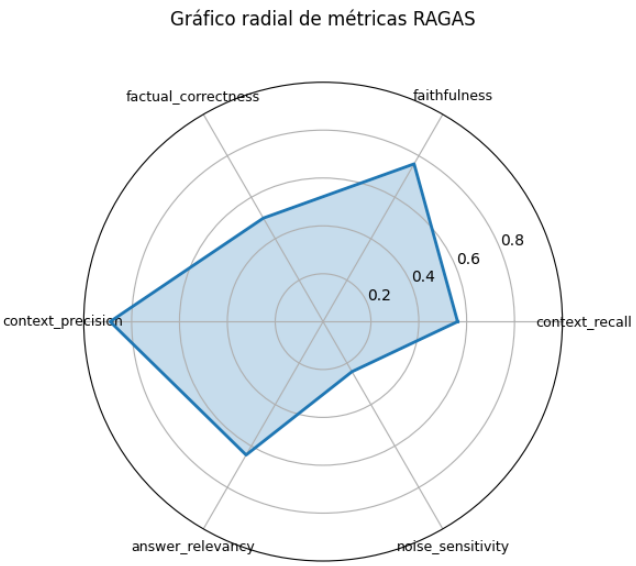


Ilustración 23. Gráfico radial de métricas RAGAS del sistema RAG (segunda iteración).

Comparando los resultados de esta iteración con respecto a la anterior, podemos destacar un aumento de un 20.94% *[(0.4993 - 0.3947) / 0.4993]* en la métrica *factual_correctness*, que representa la rigurosidad del modelo a la hora de dar datos concretos. Esta será la iteración que usaremos en el siguiente apartado.

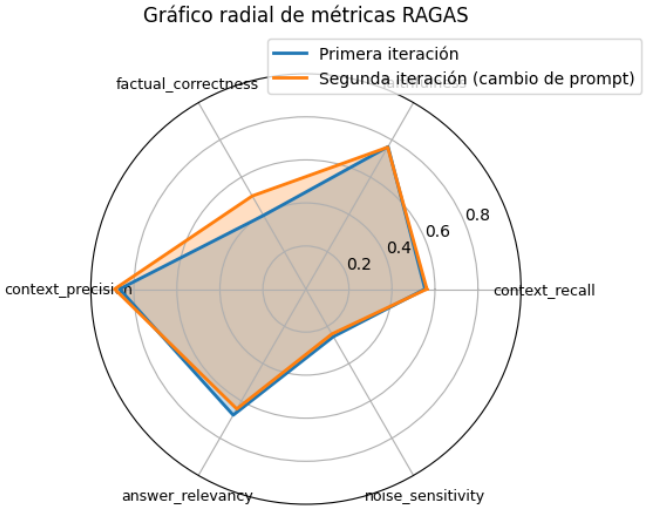


Ilustración 24. Gráfico radial de métricas RAGAS del sistema RAG (comparación entre primera y segunda iteración).

Con el objetivo ofrecer una comparación, para comprobar si el sistema RAG funcionaba daba respuestas de mayor calidad que el modelo de lenguaje base, se hizo la misma prueba con el modelo de lenguaje (*Mistral-7b)* sin el apoyo del contexto, sustituyendo la clave de fragmentos recuperados por texto vacío.

Los resultados fueron los siguientes:

```
{'context_recall': 0.0667,
 'faithfulness': 0.0339,
 'factual_correctness(mode=f1)': 0.5077,
 'llm_context_precision_with_reference': 0.0667,
 'answer_relevancy': 0.5839,
 'noise_sensitivity(mode=relevant)': 0.0000}
```

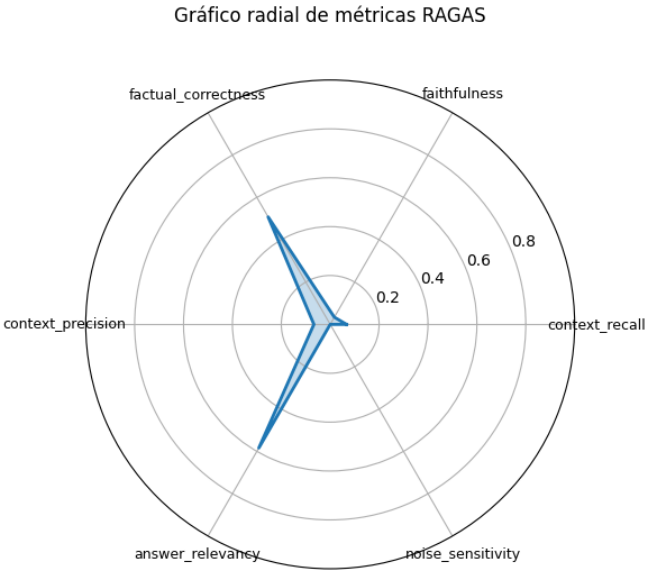


Ilustración 25. Gráfico radial de métricas RAGAS (modelo de lenguaje por defecto).

Cómo es lógico, en algunas métricas como *context_recall*, *context_precision* y *noise_sentivity*, las puntuaciones son anormalmente bajas debido a que estas métricas

evalúan la recuperación de fragmentos relevantes del sistema RAG, pero aquí esa función se ha deshabilitado.

Sin embargo, si comprobamos el resto de métricas que sí evalúan la generación de la respuesta, la métrica *factual_correctness* se muestra prácticamente igual a la del RAG, *faithfulness* es mucho más baja y *answer_relevancy* es ligeramente inferior. Para ser más exactos, la relevancia de la respuesta o AR, baja de 0,6432 a 0,5839. Eso es (0,6432-0,5839)/0,6432=0,0921=9,21%. En nuestra experimentación el LLM básico es un 9,21% peor que nuestro RAG.

Esto sirve como conclusión para demostrar que un sistema RAG puede dar mejores respuestas que un modelo de lenguaje por defecto.

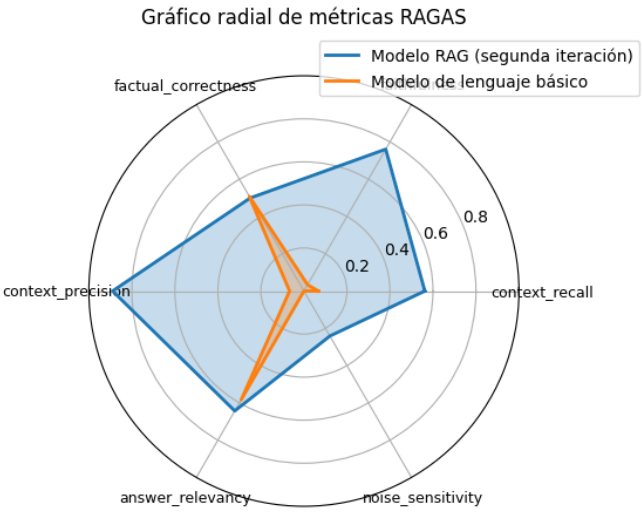


Ilustración 26. Gráfico radial de métricas RAGAS (Comparación entre modelo base y RAG)

22 Conclusiones y trabajos futuros

Este trabajo se ha dividido en dos partes principales. La primera corresponde a una sección teórica en la que se introducen los modelos de lenguaje, su funcionamiento interno, el concepto de *Retrieval-Augmented Generation* (RAG) y algunas de sus principales técnicas de optimización. La segunda parte se ha centrado en el desarrollo práctico, describiendo el proceso de creación de un sistema RAG básico y, especialmente, presentando un *chatbot* RAG ejecutado en local, diseñado para responder preguntas relacionadas con el síndrome de Rett. También se ha realizado una comparación con el mismo modelo de lenguaje sin incorporar el corpus auxiliar de documentos.

Entre las posibles mejoras que se podrían implementar en el proyecto, se destacan las siguientes:

- Profundizar en la investigación para optimizar las capacidades del sistema RAG, con el fin de mejorar la calidad y precisión de las respuestas generadas.

- Ampliar el acceso a la aplicación, ya sea mediante la creación de un repositorio que facilite su descarga e instalación como software de escritorio local, o mediante su publicación en un dominio web para uso online.
- Incorporar un mecanismo automático de extracción y actualización de documentos, que permita recoger periódicamente información desde un repositorio en línea, así como eliminar los documentos que hayan quedado obsoletos.
- Mejorar el sistema de extracción de texto desde archivos PDF. Actualmente, se recupera el contenido textual completo de los artículos médicos, pero resultaría útil añadir la capacidad de extraer información contenida en imágenes mediante herramientas de reconocimiento óptico de caracteres (OCR), como *Tesseract*.

Desde una perspectiva personal, este proyecto ha representado una valiosa introducción a un campo que desconocía, al venir desde un trasfondo de formación universitaria (rama de software) no relacionada con el tema a tratar, además de ser un tema de investigación innovador y de creciente relevancia.

Considero que el perfeccionamiento de los sistemas RAG podría contribuir significativamente a su aplicabilidad en contextos reales. En la actualidad, existen proyectos como *AnythingLLM*, que permiten a los usuarios crear sistemas RAG personalizados de forma accesible. Además, diversas empresas han comenzado a implementar sus propios *chatbots* RAG privados, alimentados con documentación especializada relacionada con su ámbito de actividad.

Estos desarrollos ponen de manifiesto tanto la utilidad de este tipo de soluciones como su creciente popularidad en distintos sectores.

23 Agradecimientos

Ha sido un largo camino desde que entré por primera vez al Campus de la Universidad de Huelva hasta que me encuentro aquí, escribiendo el final de mi trabajo de fin de grado.
A lo largo de mi trayectoria, ha habido momentos difíciles, momentos en los que pensaba que no sería capaz de manejar las circunstancias, momentos en los que pensaba que no podría continuar. Es por esto que quiero agradecer a todos aquellos que me apoyaron en este viaje.

A mis padres, Alejandro y Pilar, por siempre creer en mí, invitarme a destacar, incentivar mi desarrollo como alumno y como persona, además de ayudarme a continuar cuando dudaba de si hacerlo y aconsejarme cuando más problemas tuve.

A mi pareja, Paula, por hacerme la persona más feliz del mundo y por conseguir detener el tiempo para poder olvidarme de las situaciones estresantes cuando permanece a mi lado, por demostrarme todo el cariño que se puede otorgar a una persona, y ser mi fiel compañera de vida.

A los amigos más importantes que hice a lo largo del grado, como son Francisco de Asís, Gonzalo, Ismael, Cristian Delgado, Juan y Cristian González, por enseñarme lo que es trabajar en equipo y por todos los momentos divertidos que hemos pasado juntos.

A mi tutor Manuel de la Villa, por ayudarme a elegir un tema tan interesante como la IA generativa, además de hacer de mentor, ya que me ayudó a comprender una materia con la que no estaba familiarizado antes de empezar el proyecto.

Y, por último, a la universidad en general, por darme la oportunidad de cursar este grado, formarme como ingeniero informático, y desarrollar habilidades útiles para el mundo laboral.

24 Bibliografía

Acerca de. (s.f.). Obtenido de Mistral: https://mistral.ai/about

AWS. (s.f.). Obtenido de ¿Qué es LangChain?: Explicación sobre LangChain: AWS: https://aws.amazon.com/what-is/langchain/

Gao, Y. X. (2023). Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

HuggingFace. (5 de Enero de 2024). Obtenido de sentence-transformers/all-MiniLM-L6-v2 · Hugging Face: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

Langchain. (s.f.). Obtenido de Introduction: https://python.langchain.com/docs/introduction/

Library. (s.f.). Obtenido de Ollama: https://ollama.com/library?sort=popular

Open_llm_leaderboard. (06 de 2025). Obtenido de Huggingface: https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard#/

Q. Jiang, A., Sablayrolles , A., Mensch , A., Bamford , C., Singh Chaplot, D., de las Casas, D., . . . El Sayed, W. (2023). Mistral 7B. *arXiv preprint arXiv:2310.06825*. Obtenido de https://arxiv.org/abs/2310.06825

Sbert. (Mayo de 2025). Obtenido de Pretrained Models: https://www.sbert.net/docs/sentence_transformer/pretrained_models.html#model-overview

Vaswani, A., Parmar, N., Shazeer, N., Uszkoreit, J., Jones, L., Gómez, A., . . . Polosukhin, I. (2017). Attention is All you Need. *31st Conference on Neural Information Processing Systems (NIPS)* (pág. https://arxiv.org/abs/1706.03762). Advances in Neural Information Processing Systems. Vol. 30. Curran Associates, Inc.

Wang, X., Wang, Z., Gao, X., Zhang, F., Wu, Y., & Xu, X. (2024). Searching for best practices in retrieval-augmented generation. *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, (pp. 17716-17736).

Yang, J., Hongye Jin, Ruixiang Tang, & Xiaotian Han . (2023). *Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond.* Obtenido de https://arxiv.org/abs/2304.13712

Young, B. (30 de octubre de 2024). *How to evaluate a Langchain RAG system with RAGAs.* Obtenido de Weights & Biases Fully Connected: https://wandb.ai/byyoung3/ML_NEWS3/reports/How-to-evaluate-a-Langchain-RAG-system-with-RAGAs--Vmlldzo5NzU1NDYx