Universidad de Cantabria

Facultad de Ciencias

DESARROLLO DE UN ASISTENTE VIRTUAL LOGÍSTICO BASADO EN INTELIGENCIA ARTIFICIAL GENERATIVA

(Development of a Logistics Virtual Assistant Based on Generative Artificial Intelligence)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Rubén Martínez Amodia

Directora: Cristina Tîrnăucă

Co-Directora: Marta Zorrilla

Junio - 2025

A mis padres, por su apoyo.

A Cristina y Marta, por su dirección.

Gracias.

Resumen

Este trabajo presenta el diseño, implementación y evaluación de un asistente virtual basado en inteligencia artificial generativa con técnicas de generación aumentada por recuperación, orientado a suplir la falta de información de los modelos de lenguaje en dominios privados. El sistema responde a la necesidad de brindar asistencia técnica especializada a usuarios no técnicos que deben consultar documentación extensa sin acceso inmediato a soporte humano.

La investigación adopta un enfoque integral que equilibra la solidez teórica con la aplicación práctica de un caso real. Explora en profundidad la arquitectura *Transformer* de los modelos GPT tipo *decoder-only*, la representación vectorial de palabras en espacios de alta dimensionalidad y la identificación de hiperparámetros clave para optimizar el rendimiento. Además, se realiza una evaluación comparativa de distintos *frameworks* y modelos de lenguaje, definiendo criterios de selección y metodologías de integración según el caso de uso.

La metodología propuesta contempla tanto la calidad de recuperación como la calidad de generación de respuestas. Los resultados experimentales, obtenidos mediante un dataset de 27 preguntas representativas elaborado en colaboración con expertos del dominio, demuestran mejoras significativas, mostrando incrementos de 8.8 puntos en la métrica BLEU y 25.4 puntos en ROUGE-L comparado con configuraciones sin generación aumentada por recuperación. La evaluación revela que el sistema alcanza un 81 % de capacidad de respuesta efectiva sin presencia de alucinaciones, con un 54.5 % de respuestas clasificadas como excelentes o buenas cuando dispone de contexto relevante.

En particular, el análisis de configuraciones paramétricas identifica puntos óptimos: temperatura de 0.6 para equilibrar precisión y naturalidad, fragmentos textuales de 2000–3000 caracteres para maximizar cobertura contextual, y búsqueda por similitud tradicional sobre algoritmos de máxima relevancia marginal en dominios especializados. Se evidencian múltiples *trade-offs* inherentes entre diversidad de recuperación y precisión de respuestas, cantidad de contexto y capacidad de integración semántica.

Como contribución adicional, se desarrolla `ragbot`, una herramienta de terminal que permite la experimentación sistemática con configuraciones de parámetros para diferentes casos de uso, facilitando la evaluación automatizada mediante métricas cuantitativas (BLEU, ROUGE-L, similitud semántica) y cualitativas (relevancia contextual, relevancia de respuesta y fidelidad factual).

PALABRAS CLAVE
Asistente conversacional; generación aumentada por recuperación; modelo de lenguaje de gran escala; inteligencia artificial generativa

Abstract

This work presents the design, implementation, and evaluation of a virtual assistant based on generative artificial intelligence with retrieval-augmented generation techniques, aimed at addressing the lack of information in language models within private domains. The system responds to the need to provide specialized technical assistance to non-technical users who must consult extensive documentation without immediate access to human support.

The research adopts a comprehensive approach that balances theoretical robustness with the practical application of a real-world case. It explores in depth the Transformer architecture of decoder-only GPT models, the vector representation of words in high-dimensional spaces, and the identification of key hyperparameters to optimize performance. In addition, it conducts a comparative evaluation of different frameworks and language models, defining selection criteria and integration methodologies according to the use case.

The proposed methodology considers both retrieval quality and response generation quality. The experimental results, obtained using a dataset of 27 representative questions developed in collaboration with domain experts, demonstrate significant improvements, showing increases of 8.8 points in the BLEU metric and 25.4 points in ROUGE-L compared to configurations without retrieval-augmented generation. The evaluation reveals that the system achieves 81 % effective response capability without hallucinations, with 54.5 % of responses rated as excellent or good when relevant context is available.

In particular, the analysis of parametric configurations identifies optimal points: a temperature of 0.6 to balance accuracy and naturalness, text fragments of 2000–3000 characters to maximize contextual coverage, and traditional similarity search over maximal marginal relevance algorithms in specialized domains. Multiple inherent trade-offs are evident between retrieval diversity and response accuracy, amount of context and semantic integration capability.

As an additional contribution, `ragbot` is developed—a terminal tool that enables systematic experimentation with parameter configurations for different use cases, facilitating automated evaluation through quantitative metrics (BLEU, ROUGE-L, semantic similarity) and qualitative ones (contextual relevance, response relevance, and factual fidelity).

KEYWORDS
Conversational assistant; retrieval augmented generation; large language model; generative artificial intelligence

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1

Introducción

"Solo sé que no sé nada".

Sócrates

SECCIÓN 1.1

Motivación

Vivimos una etapa de efervescencia tecnológica en la que la inteligencia artificial (IA) generativa está adquiriendo un papel protagonista, impulsada en gran medida por el auge de los modelos de lenguaje de gran escala (LLM). Plataformas como ChatGPT se han popularizado rápidamente, no solo por sus capacidades, sino también por la conveniencia que ofrecen. Frente a una búsqueda convencional que retorna una lista de enlaces, estos modelos permiten al usuario dialogar en lenguaje natural y obtener respuestas directas, articuladas y contextuales. En pleno 2025, ya existen modelos capaces de acceder a internet y recuperar información actualizada para ofrecer respuestas más precisas. Por ejemplo, si preguntamos por la última novela publicada por un autor determinado, el modelo consultará la red y devolverá la información deseada. Pero si le preguntamos por el contenido de un documento privado almacenado en nuestro escritorio, ¿podrá responder?

Este tipo de reflexión abre una cuestión central en el desarrollo y uso de la IA: su conocimiento está acotado a la información con la que ha sido entrenada o aquella a la que se le da acceso explícito. En este contexto, la incorporación de estos modelos en productos digitales representa una ventaja competitiva significativa. Las grandes empresas tecnológicas ya los integran en sus ciclos de desarrollo, optimizando tiempos y recursos. Debido a las limitaciones inherentes de los LLM, se ha impulsado la creación de agentes, sistemas capaces de realizar tareas complejas que requieren razonamiento secuencial, utilizando herramientas externas en combinación con procesamiento del lenguaje natural (NLP). Un ejemplo paradigmático es Github Copilot, que colabora activamente en el entorno de desarrollo del programador, sugiriendo y escribiendo código en tiempo real.

Retomando la pregunta anterior: ¿sabe ChatGPT qué contiene nuestro documento privado? La respuesta es clara: no. El modelo base, como veremos a lo largo de este trabajo, se corresponde con un *Generative Pretrained Transformer* (GPT), es decir, un modelo entrenado con grandes volúmenes de datos públicos. En [1], se introdujo el concepto de generación aumentada por recuperación (RAG), una arquitectura que permite a estos modelos acceder a bases de conocimiento específicas mediante técnicas de recuperación de información. Así, se suple la falta de conocimiento puntual del modelo y se le dota de la capacidad de generar respuestas en contextos que le serían, de otro modo, inaccesibles.

SECCIÓN 1.2

Contexto y objetivo

El principal objetivo de este trabajo es integrar un asistente virtual en una aplicación de gestión de flotas, cubriendo de esta forma la necesidad de una empresa cuya documentación interna es privada (y, por tanto, desconocida por los LLM). La aplicación en cuestión se debe usar muchas veces por personal sin conocimientos técnicos avanzados, lo que hace que a menudo los usuarios sean incapaces de realizar algunas acciones específicas. Para aclarar sus dudas, dichos usuarios disponen de un manual de uso de la aplicación, además de vídeos explicativos. El manual en cuestión supera las 150 páginas y cada uno de los vídeos tiene al menos media hora de duración. Esto deriva en consultas por parte de los usuarios al personal de soporte de la empresa, que no siempre pueden ser atendidas en tiempo real.

Con el objeto de dar una solución a este problema, implementamos un asistente virtual que dé respuesta a las preguntas de los usuarios sin necesidad de la intervención de un equipo de soporte técnico. Debido al carácter privado del manual y los vídeos de la aplicación, utilizamos el método de RAG para desarrollarlo. La integración del asistente supone no solo una optimización de la experiencia del usuario sino que además contribuye a una mejora en la gestión de los recursos humanos de la empresa, permitiendo asignar al personal de soporte tareas de mayor valor estratégico.

Más allá del diseño del asistente, este trabajo persigue un segundo objetivo fundamental: ofrecer una comprensión profunda del funcionamiento de los sistemas basados en RAG. Se pone especial énfasis en desentrañar los principios que sustentan el comportamiento de los LLM, y en presentar un enfoque metodológico riguroso para el diseño de asistentes virtuales inteligentes. No se trata únicamente de aplicar una técnica, sino de comprender, componente por componente, cómo y por qué funciona cada elemento del sistema. Por ello, se incluye una descripción teórica detallada de todos los conceptos y tecnologías relevantes involucradas. Asimismo, se realiza un análisis exhaustivo de los hiperparámetros más influyentes, evaluando su impacto en el rendimiento del asistente en el contexto específico de este caso de estudio.

SECCIÓN 1.3

Estructura del trabajo

Después de esta introducción, el resto del documento se organiza como sigue. En el capítulo 2 se lleva a cabo una revisión histórica de los antecedentes del desarrollo de agentes conversacionales y se realiza un estudio comparativo de *frameworks* y LLM para la implementación de los mismos. En el capítulo 3 se sientan las bases para comprender el funcionamiento de los LLM, describiendo con detalle su arquitectura subyacente y dando una intuición sobre la representación multidimensional de las palabras. En el capítulo 4 se define una metodología para el diseño de agentes conversacionales basados en RAG y se estudian los parámetros trascendentales. En el capítulo 5 se describen el diseño y la implementación del sistema. En el capítulo 6 se evalúa el asistente virtual implementado y se discuten los resultados obtenidos. Finalmente, en el capítulo 7 se presentan las conclusiones y las posibles líneas de trabajo futuro.

CAPÍTULO 2

Estado del arte

"Sapiens can cooperate in extremely flexible ways with countless numbers of strangers. That's why Sapiens rule the world, whereas ants eat our leftovers and chimps are locked up in zoos and research laboratories."

Yuval Harari

SECCIÓN 2.1

Revisión histórica

2.1.1. Desarrollo de la lingüística

Yuval Harari, en su famoso superventas *Sapiens*, describe fielmente por qué los seres humanos hemos conseguido dominar la Tierra. A diferencia del resto de seres vivos, los humanos hemos desarrollado la capacidad de cooperar en grandes grupos gracias a un lenguaje extremadamente complejo y versátil. Como él mismo señala, «si hay algún secreto para nuestro éxito, y recuerden, éxito en la supervivencia en la naturaleza, este es que nuestros cerebros se desarrollaron para comunicarse». La complejidad de nuestro lenguaje es inmensa y desde el siglo XX despertó el interés de los lingüistas.

Una de las primeras teorías del lenguaje se remonta a comienzos del siglo XX. Ferdinand de Saussure definió el lenguaje natural como una estructura de elementos mutuamente enlazados, similares o contrapuestos entre sí, lo que dio lugar a una corriente lingüística conocida como estructuralismo. Durante el periodo de 1920 a 1950, Leonard Bloomfield y otros estructuralistas americanos consideraron el orden de las palabras en una oración como la herramienta principal para estructurar las oraciones, dando lugar al método de separación en constituyentes sintácticos.

En 1950 Noam Chomsky introducía una teoría puramente matemática que consolidaba una jerarquía de gramáticas generativas de diferente complejidad. En particular, las gramáticas libres de contexto se convirtieron en la herramienta fundamental para describir el lenguaje humano. La estructura sintáctica de una oración se describía entonces mediante un árbol sintáctico, esto es, una estructura anidada que identifica cada una de sus partes jerárquicamente. Como es evidente, no todas las oraciones podían definirse de esta manera, así que el propio Chomsky proponía una nueva manera de describirlas, las gramáticas transformacionales, que posibilitaban la transformación de una oración descriptible en una que no lo es.

Más tarde, se introdujo la subcategorización de los verbos en función de sus posibles complementos. Para lidiar con idiomas especialmente complicados, como el español, en los que por ejemplo existe

una correspondencia entre la persona gramatical y el número en lo que se refiere a sujeto y verbo, se comenzó a utilizar un nuevo enfoque basado en restricciones, el conocido como *Generalized Phrase Structure Grammar* (GPSG). En la mayoría de los constituyentes sintácticos, uno de sus subconstituyentes es más trascendente que los demás. Por esta razón, se ampliaba el método GPSG al método *Head-Driven Phrase Structure Grammar* (HPSG). Este último introduce el concepto de núcleo del constituyente, lo que facilitó significativamente el análisis sintáctico de las oraciones. A finales de 1960, se proponía en Rusia la teoría *Meaning* $\Longleftrightarrow$ *Text Theory* (MTT), que consideraba el lenguaje como un transformador multietapa de significado a texto y viceversa. Esta teoría es capaz de describir cualquier lenguaje natural y cualquier nivel lingüístico en él.

2.1.2. Evolución del procesamiento del lenguaje natural

El NLP es un subcampo de la IA que tiene por objetivo dotar a los computadores de la capacidad de procesar datos codificados en lenguaje natural. Este campo está estrechamente relacionado con la lingüística computacional y abarca tareas diversas como la recuperación de información, el análisis de sentimiento, la generación de lenguaje, el reconocimiento de voz, entre otras. Sin embargo, es en la traducción automática donde podríamos situar el origen del interés por este campo.

A pesar de que ya en 1913 Markov propuso modelos basados en *n*-gramas de letras para modelar el lenguaje [2], los precursores del NLP fueron quienes se interesaron en el diseño de sistemas de traducción automática. Los primeros sistemas de este tipo aparecieron en 1933, con el otorgamiento de dos patentes independientes de diccionarios mecánicos en Francia y Rusia. Por un lado, el francés Georges Artstrouni presentó un diccionario mecánico bilingüe conocido como el «cerebro mecánico», un dispositivo con la capacidad de realizar traducciones básicas palabra por palabra. Por otro lado, el ruso Petr Troyanskii propuso el diseño de una máquina de tres etapas dedicada a la traducción de texto. En la primera etapa, un editor humano conocedor del idioma de origen debía analizar el texto de forma que todas las palabras que intervinieran en él fueran reemplazadas por su forma canónica. En la segunda etapa, la máquina transformaría estas secuencias de formas canónicas del texto original en secuencias de formas canónicas en el idioma objetivo. Por último, en la tercera etapa, un editor entendido del idioma de destino convertiría esta secuencia en las formas normales adecuadas.

Warren Weaver, que había dedicado su tiempo al estudio de los procesos de comunicación durante la Segunda Guerra Mundial, establecía en 1947 una analogía entre la traducción automática y la criptografía. En una carta dirigida a Norber Wiener, reconocido lingüista del Instituto de Tecnología de Massachussets, escribía: «uno naturalmente se pregunta si el problema de la traducción podría, en principio, tratarse como un problema de criptografía. Cuando veo un artículo en ruso, digo: esto en realidad está escrito en inglés, pero ha sido codificado en unos símbolos extraños. Ahora procederé a decodificar», abriendo una vía de investigación que culminaría siendo refutada. Dos años más tarde Weaver escribió la parte introductoria de la *Teoría Matemática de la Comunicación* de Claude Shannon, que fue el primero en generar modelos de *n*-gramas de palabras en inglés [3].

En 1950, Alan Turing, conocido como el padre de la informática teórica, proponía un experimento reconocido como el test de Turing con objeto de definir un estándar para determinar la capacidad intelectual de una máquina [4] que, entre otros aspectos, incluía la evaluación de la interpretación y generación automáticas de lenguaje natural. El 7 de enero de 1954, una demostración del experi-

mento Georgetown-IBM supuso uno de los eventos más importantes de la historia de la traducción automática. El experimento mostraba al público la capacidad de una máquina para traducir más de sesenta frases en ruso al inglés y, por ende, estimuló el optimismo sobre la viabilidad de un sistema de traducción automática a corto plazo.

En 1956, Chomsky señaló que «los modelos probabilísticos no proveen ningún indicador particular en algunos de los problemas básicos de estructura sintáctica», observación que dio lugar a que se dejasen de lado los modelos estadísticos durante décadas, recobrando su importancia con su utilización en el campo del reconocimiento de voz en 1976 [5]. En los años 60 se diseñó ELIZA, uno de los primeros sistemas de procesamiento del lenguaje natural capaces de mantener una conversación con el usuario. ELIZA asumía el rol de un psicoterapeuta rogeriano (centrado en la persona) que se limitaba a contestar a los usuarios basándose en un programa notablemente sencillo de detección de patrones mediante expresiones regulares [6]. Sorprendentemente, las personas que interactuaron con ELIZA realmente sintieron que eran entendidas y valoraban la capacidad del programa para comprender sus problemas.

El término «modelo de lenguaje» fue acuñado por Jelinek y sus compañeros de IBM, aportándole la significación de un conjunto de influencias lingüísticas que afectaban la probabilidad de una determinada secuencia de palabras, lo que incluía la gramática, la semántica y el discurso, en lugar de exclusivamente referirse al modelo particular de *n*-gramas. Múltiples estrategias de modelización del lenguaje y de *smoothing* se propusieron entre el año 1980 y el año 2000, incluyendo el Good-Turing *smoothing* [7] y el descuento de Witten-Bell [8], y a finales de siglo, el modelo *modified interpolated* Kneser-Ney [9] se convirtió en el estándar de referencia para los modelos de lenguaje basados en *n*-gramas.

A finales de la década de los 80, tuvo lugar una revolución del NLP con la introduccion de los algoritmos de aprendizaje automático, capaces de aprender a partir de conjuntos de datos, lo que implicó que la dominancia de las teorías chomskianas fuera viéndose reducida paulatinamente. En los años 90 comenzaron a aparecer aplicaciones prácticas de procesamiento del lenguaje basadas en redes neuronales, así como el reconocimiento de voz [10]. Las primeras redes neuronales recurrentes (RNN) y las redes *Long Short-Term Memory* (LSTM) se entrenaron durante estos años. Las RNN son un tipo de red neuronal capaz de procesar datos secuenciales, de manera que la salida de una capa puede realimentarse sobre la misma capa, permitiendo mantener un estado; lo que las hace muy interesantes para su uso en aplicaciones de NLP, en las que el contexto en que se encuentra una palabra juega un papel muy importante para comprender su significado. No obstante, a pesar de su interés teórico, su progreso en la práctica se vio impedido por la dificultad de entrenarlas y la complejidad de gestión contextual de largas secuencias de texto. Con la introducción de las redes LSTM en 1997, esta situación cambiaría por completo, ya que fueron diseñadas para dar solución al problema del desvanecimiento de gradiente presente en las RNN [11].

A comienzos del siglo XXI, mejoras en el hardware de los computadores y avances en las técnicas de optimización y entrenamiento hicieron posible el desarrollo de redes neuronales más grandes, dando lugar al nacimiento del aprendizaje profundo [12]. El interés en la aplicación de redes neuronales a aplicaciones prácticas de NLP tiene su origen en las contribuciones de Collobert en 2008 y 2011 [13, 14], demostrando un rendimiento increíble en numerosas tareas de NLP. En 2013, Tomas

Mikolov introdujo el modelo word2vec [15], una red neuronal capaz de aprender *word embeddings*, es decir, representaciones vectoriales de palabras en un espacio vectorial de alta dimensión, lo que permitió a los modelos de lenguaje aprender las relaciones semánticas y sintácticas entre palabras. En 2014 se introducía un modelo similar, GloVe [16]. La combinación de LSTM con *word embeddings* preentrenados se convirtió rápidamente en el modelo dominante en diversas tareas de NLP.

En 2015, Bahdanau introdujo los modelos *Sequence-to-Sequence* (Seq2Seq), redes neuronales capaces de mapear secuencias de entrada y salida de longitud variable [17]. No obstante, los modelos Seq2Seq mostraron limitaciones al tener que enfrentarse a secuencias de gran longitud en tareas de traducción automática. Finalmente, un hallazgo innovador en la evolución del NLP fue la arquitectura *Transformer*, introducida por un grupo de investigadores de Google en su insólito artículo *Attention Is All You Need* [18]. En él describían una arquitectura basada en un codificador y un decodificador, que supondría un punto de inflexión en el rendimiento de aplicaciones de traducción automática. Al año siguiente, Google presentaba el modelo *Bidirectional Encoder Representations from Transformers* (BERT) [19], un nuevo modelo de lenguaje diseñado para entrenar representaciones bidireccionales a partir de texto no etiquetado, lo que permitiría hacer *fine-tuning* sobre el modelo BERT preentrenado, añadiendo una única capa de salida adicional. De esta forma, el modelo podía ser configurado para construir otros modelos de gran rendimiento en diferentes tareas de NLP.

2.1.3. Modelos de lenguaje generativos y RAG

OpenAI es la empresa más disruptiva en los últimos años en términos de desarrollo de modelos de lenguaje generativos. En 2018 introducía la primera propuesta de GPT, que denominaron GPT-1[1], demostrando que se pueden obtener mejoras significativas en tareas de *Natural Language Understanding* (NLU) mediante el preentrenamiento generativo de un modelo de lenguaje en un texto no etiquetado, seguido de una fase de *fine-tuning* discriminativo dedicada a tareas específicas [20]. Este modelo se basa en la arquitectura *Transformer* y precisa de 117 millones de parámetros. Los modelos que fueron introduciendo después se basan en la misma arquitectura, introduciendo ligeras modificaciones pero incrementando considerablemente la cantidad de información utilizada en la fase de entrenamiento.

Así como GPT-1 fue entrenado con solamente 4.5 GB de texto de libros no publicados extraídos del *dataset* BookCorpus [21], GPT-2[2] lo fue con 40 GB de texto provenientes del *dataset* WebText, formado por 8 millones de documentos extraídos de 45 millones de páginas web, constituyendo un modelo de 1500 millones de parámetros [22]. La tercera generación de estos modelos, GPT-3, fue publicada en 2020 y su versión de mayor tamaño alcanzaba los 175 mil millones de parámetros [23]. Entre otras fuentes de datos, para entrenar este modelo OpenAI utilizó 570 GB de texto del *dataset* CommonCrawl[3], WebText y la Wikipedia inglesa. Otros modelos con gran adopción entre 2020 y 2022 fueron T5 [24], *BigScience Large Open-science Open-access Multilingual Language Model* (BLOOM) [25] y PaLM [26].

Hasta entonces estos modelos eran exclusivamente conocidos por investigadores y entusiastas del campo de la IA. Sin embargo, en 2022, el lanzamiento de ChatGPT captó la atención del mundo

[1] https://github.com/openai/finetune-transformer-lm
[2] https://github.com/openai/gpt-2
[3] https://commoncrawl.org/

entero, alcanzando el mayor pico de visitas en un día de la historia. Un año después se publicó GPT-4, un modelo que ya no solo tenía la capacidad de generar texto, sino que desempeñaba funciones multimodales como el procesamiento de imágenes [27]. A finales de 2024, OpenAI publicaba su modelo más potente hasta el momento, OpenAI o1, que presentaba una arquitectura innovadora que dotaría al LLM de la capacidad de razonar [28]. Para ello, este modelo genera largas cadenas de pensamiento antes de responder al usuario, dando lugar a respuestas estructuradas de rigor deductivo. Los modelos de razonamiento son los más avanzados hasta el momento y ya existen otros modelos de este tipo que difieren de los de OpenAI y que han demostrado capacidades similares incurriendo en un coste de entrenamiento significativamente menor, como es el caso del modelo de código abierto DeepSeek R1 publicado por un grupo de investigación chino en enero de 2025 [29].

Mientras tanto, con el objetivo de aunar la capacidad generativa de los LLM y la recuperación de información, se introdujo la estrategia RAG. Este concepto se popularizó en [1], donde un grupo de investigadores de Meta proponía un modelo que, combinando un recuperador vectorial con un modelo Seq2Seq, mostró mejoras significativas en tareas intensivas en conocimiento.

El concepto de recuperación de información hace referencia a la cuantización de la similitud textual y fue fuertemente influenciado por el nacimiento de la *World Wide Web* en los años 90. El modelo *Vector Space* junto con la métrica *Term Frequency–Inverse Document Frequency* (TF-IDF), introducidos en la década de los 80, sentaron las bases para el desarrollo de sistemas de recuperación de información. Los algoritmos de RAG se introdujeron por primera vez como una forma de modelar el lenguaje [30]. Más tarde, comenzaron a dedicarse a la mejora de respuestas de propósito específico, como en sistemas de diálogo de dominio cerrado. La idea subyacente de la integración de recuperación de información en estos sistemas consiste en la ampliación del contexto con que se alimenta al LLM, aprovechando al máximo la capacidad de su ventana contextual. En determinados dominios, los LLM carecen de la información pertinente para responder preguntas de los usuarios, luego una solución es recabar toda aquella información relevante que no necesariamente tiene que estar codificada en los parámetros del modelo. A raíz de esta idea surgen numerosas líneas de investigación, así como el número de documentos que deben recuperarse o la posición de los mismos en el espacio contextual del *prompt* del LLM. En [31], se examina detalladamente la evolución de los diferentes paradigmas de generación aumentada por recuperación y, en [32], se enfatiza que la utilización de *datasets* públicos para la evaluación de la recuperación puede dar lugar a un diseño subóptimo de sistemas RAG. Por su parte, en [33], se estudia la variación del rendimiento de los sistemas RAG en función de la posición de la información relevante dentro de grandes contextos, introduciendo el problema «*Lost in the Middle*». Finalmente, un estudio reciente examina el impacto de la introducción de ruido en el *prompt* del LLM [34], lo que resulta fundamental a la hora de definir el valor de los parámetros que intervienen en el funcionamiento de un sistema RAG.

SECCIÓN 2.2

Estudio comparativo: *frameworks* y modelos

A la hora de enfrentar la construcción de un asistente virtual, una opción es construirlo totalmente desde cero, implementando cada uno de los componentes que lo constituyen. No obstante, una buena práctica en el desarrollo de sistemas es utilizar *frameworks* que proveen una serie de módulos que podemos integrar en nuestra aplicación, sin necesidad de «reinventar la rueda». En esta sección introducimos las diferentes maneras de abordar la construcción de un *chatbot* y estudiamos los diferentes LLM disponibles.

2.2.1. *Frameworks*

La elección del *framework* a utilizar debe ajustarse al caso de uso en cuestión. Para compararlos, definimos una serie de criterios que nos permitirán observar con precisión las diferencias más significativas entre los diferentes *frameworks* estudiados (ver Tabla 2.1).

Tabla 2.1: Métricas para la evaluación de *frameworks* de desarrollo de *chatbots*.

Criterio	Descripción
Sencillez de uso	Facilidad para construir, entrenar y mantener el *chatbot*.
Soporte de integración	Capacidad para integrarse con API de terceros, bases de datos y diferentes plataformas.
Escalabilidad	Capacidad para manejar grandes volúmenes de usuarios y responder eficientemente.
Flexibilidad de despliegue	Lugares donde se puede desplegar el *chatbot*.
Monitorización	Capacidad para monitorizar las acciones de los usuarios.
Soporte multimodal	Soporte para texto, reconocimiento de voz y procesamiento de imágenes.
Coste	Precio incurrido en su utilización.
Comunidad y soporte	Disponibilidad de recursos de aprendizaje y soporte.

A continuación, presentamos un análisis comparativo de algunos de los *frameworks* más conocidos de acuerdo con los criterios establecidos.

Google Dialogflow[4]. Este *framework* presenta una interfaz intuitiva que facilita significativamente la creación y mantenimiento de *chatbots*, con herramientas visuales que reducen la necesidad de programación. En términos de integración, ofrece API REST y cliente que permiten conexiones robustas con sistemas externos y bases de datos. Además, demuestra excelente escalabilidad bajo el respaldo de la infraestructura de Google Cloud, manejando eficientemente grandes volúmenes de tráfico con opciones automáticas de escalado. Su flexibilidad de despliegue abarca desde aplicaciones web y móviles hasta asistentes de voz, complementada por herramientas sólidas de analítica que permiten monitorizar patrones de uso, tasas de abandono y efectividad de respuestas en tiempo real. Consta además de capacidad multimodal, con reconocimiento de voz preciso mediante la tecnología Speech-to-Text de Google, modelos de NLU basados en BERT con soporte para más de 30 idiomas

[4] https://cloud.google.com/products/conversational-agents

y capacidad para procesar imágenes cuando se integra con otras API de Google Cloud. En cuanto a costes, opera bajo un modelo *freemium*, con una versión *Essentials* con un coste de 0.002€ por consulta, mientras que la versión *Enterprise* ofrece mayor rendimiento y funciones avanzadas con precios que escalan según el uso. Finalmente, la comunidad y soporte que rodea a Dialogflow constituye uno de sus mayores activos, contando con documentación detallada, tutoriales oficiales, foros activos y amplia presencia en plataformas como Stack Overflow, GitHub y YouTube, además de la opción de soporte técnico directo de Google para clientes empresariales.

Wit.ai[5]. Se presenta como una solución accesible para el desarrollo de *chatbots* con un enfoque en la simplicidad y democratización de la tecnología conversacional. Su interfaz visual intuitiva permite definir intenciones y entidades mediante ejemplos, reduciendo significativamente la barrera de entrada técnica para desarrolladores sin experiencia previa en IA. En cuanto a integración, ofrece API REST que facilitan la conexión con diversas plataformas, aunque su vinculación con el ecosistema de Meta favorece particularmente las integraciones con productos de esta compañía. Respecto a escalabilidad, aunque puede manejar volúmenes moderados de tráfico, presenta algunas limitaciones en comparación con soluciones empresariales más robustas, si bien su adquisición por Meta ha mejorado su infraestructura subyacente para soportar implementaciones de mayor escala. La flexibilidad de despliegue se concentra principalmente en aplicaciones web, móviles y plataformas de mensajería, con herramientas básicas pero funcionales para la monitorización de interacciones que permiten analizar patrones de uso y la efectividad de las respuestas del *chatbot*. Asimismo, dispone de soporte para 132 idiomas, su uso no presenta ningún coste y precisa de una comunidad de más de 350000 desarrolladores.

Amazon Lex[6]. Destaca como una plataforma empresarial sólida para el desarrollo de interfaces conversacionales, aprovechando la misma tecnología que impulsa Alexa. Su entorno de desarrollo estructurado facilita la creación de *chatbots* sofisticados mediante un sistema de definición de intenciones, *slots* y flujos de diálogo, aunque requiere cierta curva de aprendizaje para aprovechar todo su potencial. En el aspecto de integración, sobresale por su perfecta conexión con el ecosistema AWS, permitiendo vincular fácilmente servicios como Lambda, DynamoDB y Amazon Connect, además de ofrecer API versátiles para integración con plataformas externas. Su capacidad de escalamiento es excepcional gracias a la infraestructura de AWS, con escalado automático que maneja eficientemente picos de tráfico sin degradación del rendimiento, mientras que su flexibilidad de despliegue abarca desde aplicaciones web y móviles hasta centros de contacto y asistentes de voz, complementada con robustas herramientas analíticas que ofrecen métricas detalladas sobre rendimiento, patrones de uso y tasas de conversión a través de Amazon CloudWatch. AWS ofrece un nivel gratuito para nuevos usuarios que incluye hasta 10000 solicitudes de texto y 5000 solicitudes de voz por mes durante el primer año, permitiendo experimentación y desarrollo inicial sin costes. Si bien AWS consta de una enorme comunidad de desarrolladores, también integra un asistente virtual empotrado con ánimo de ayudar al usuario en el desarrollo del *chatbot*.

IBM watsonx Assistant[7]. Constituye un *framework* robusto para el desarrollo de *chatbots* con un notable equilibrio entre sofisticación técnica y accesibilidad. Su interfaz combina elementos visuales

[5] https://wit.ai/
[6] https://aws.amazon.com/lex/
[7] https://www.ibm.com/products/watsonx-assistant

intuitivos con opciones avanzadas de configuración. En el ámbito de integración, ofrece conectividad empresarial completa mediante API REST y *Software Development Kit* (SDK) para diversos lenguajes de programación, permitiendo conexiones fluidas con sistemas *Cutomer Relationship Management* (CRM) y bases de datos corporativas, además de una integración nativa con otros servicios de IBM Cloud que potencia sus capacidades analíticas y cognitivas. Presenta una arquitectura diseñada para entornos empresariales que soporta implementaciones globales con millones de interacciones, mientras que su flexibilidad de despliegue abarca desde sitios web corporativos y aplicaciones móviles hasta centros de contacto y sistemas telefónicos. Además, dispone de funcionalidades de monitorización avanzadas que incluyen paneles detallados sobre rendimiento y patrones de uso. Tiene soporte multilingüe, reconocimiento de voz y posibilidad de procesamiento de imágenes. En términos de costes, opera bajo un modelo por niveles que incluye un plan *Lite* gratuito con limitaciones significativas. Finalmente, el soporte y comunidad que rodea a watsonx Assistant es sólido y orientado al entorno empresarial, con documentación técnica extensa, recursos educativos de alta calidad y opciones de soporte técnico.

Rasa[8]. Una plataforma de código abierto que prioriza la personalización y el control total sobre el desarrollo de *chatbots*. Su arquitectura modular basada en Python permite a los desarrolladores construir *chatbots* sofisticados con total control sobre el procesamiento del lenguaje natural y la gestión del diálogo. En términos de integración, ofrece una flexibilidad excepcional mediante conectores predefinidos para plataformas populares como Slack, Telegram y Facebook Messenger, además de un sistema de canales personalizables que facilita la conexión con prácticamente cualquier interfaz o *backend*, permitiendo integraciones profundas con sistemas empresariales existentes y bases de datos a través de API REST o conexiones directas. Su capacidad de escalabilidad es notable para una solución gratuita, con posibilidad de implementación en contenedores Docker y compatibilidad con Kubernetes para orquestación y escalado horizontales. Rasa presenta un plan de suscripciones basado en niveles, permitiendo hasta 1000 conversaciones mensuales en el plan gratuito. La flexibilidad de despliegue constituye uno de sus mayores atractivos, permitiendo implementaciones *on-premise* que garantizan la soberanía total de los datos, despliegues en nubes privadas o públicas, e incluso en dispositivos *edge* con requisitos de conectividad limitada, complementado con herramientas de monitorización que, aunque menos sofisticadas que las soluciones comerciales, ofrecen opciones de registro, seguimiento de conversaciones y evaluación de modelos.

LangChain[9]. Emerge como un *framework* innovador diseñado específicamente para desarrollar aplicaciones potenciadas por LLM, diferenciándose por su enfoque en la construcción de experiencias conversacionales basadas en cadenas de componentes modulares. Su filosofía de desarrollo facilita la creación de *chatbots* avanzados mediante la composición de elementos como modelos de lenguaje y recuperación de documentos, aunque exige conocimientos sólidos de programación y comprensión conceptual de las arquitecturas de LLM. En el ámbito de integración, destaca por su capacidad para conectarse con una amplia variedad de proveedores de modelos de lenguaje como OpenAI, Google Vertex AI y Anthropic, además de ofrecer opciones de integración con múltiples bases de datos, modelos de *embeddings* y API externas, permitiendo la creación de sistemas que combinan conocimiento específico del dominio con la potencia generativa de los LLM. Su escalabilidad está

[8] https://rasa.com/
[9] https://www.langchain.com/

en evolución constante, con un diseño que favorece la eficiencia computacional mediante técnicas como el procesamiento por lotes y la recuperación optimizada, aunque requiere consideraciones arquitectónicas cuidadosas para implementaciones a gran escala. La flexibilidad de despliegue es considerable gracias a su naturaleza de librería de Python, permitiendo implementaciones en cualquier entorno que soporte este lenguaje, desde servidores locales hasta plataformas en la nube como AWS Lambda o Google Cloud Functions. Además, presenta herramientas para la monitorización que permiten el seguimiento de cadenas, evaluación de respuestas y depuración de razonamientos que resultan cruciales para refinar la calidad de las interacciones basadas en LLM. Langchain, al tratarse de un *framework* de código abierto, no supone ningún coste más allá del incurrido en el uso de los modelos empleados, y dispone de tutoriales, foros y una gran comunidad de desarrolladores.

LlamaIndex[10]. Se presenta como una infraestructura de datos especializada que facilita la conexión entre LLM y fuentes de datos externas, con un enfoque particular en la generación aumentada por recuperación. Su arquitectura estructurada ofrece una experiencia de desarrollo bien organizada con componentes modulares para indexación, recuperación y generación de respuestas, aunque requiere conocimientos intermedios de Python y familiaridad con conceptos de NLP para su implementación efectiva. En términos de integración, destaca por su versatilidad al conectarse con una amplia gama de fuentes de datos, incluyendo documentos PDF, sitios web y bases de datos SQL, además de soportar múltiples proveedores de LLM como OpenAI, Anthropic y modelos de código abierto, permitiendo arquitecturas híbridas que aprovechan las fortalezas de diferentes modelos. Su capacidad de escalabilidad se manifiesta principalmente a través de optimizaciones en el procesamiento de datos y técnicas de indexación avanzadas que reducen los costes computacionales, con opciones de balanceo de carga y procesamiento por lotes. Al igual que LangChain, su flexibilidad de despliegue es considerable por tratarse de una librería de Python, facilitando implementaciones en entornos diversos. Sus capacidades de monitorización incluyen herramientas emergentes para evaluación de relevancia de respuestas, análisis de recuperación y seguimiento de dependencias, complementadas con integraciones con plataformas de observabilidad para entornos de producción. Como LangChain, LlamaIndex consta de una comunidad enorme de desarrolladores y su uso no supone ningún coste.

Además de los que hemos analizado, existen muchos otros *frameworks* para el desarrollo de *chatbots*. Mientras que Google Dialogflow, wit.ai, Amazon Lex y watsonx facilitan su construcción mediante herramientas visuales, Rasa, LangChain y LlamaIndex son más complejos de utilizar y resulta necesario tener conocimiento de diseño de sistemas y programación para su implementación. No obstante, estas últimas permiten un mayor control sobre cada uno de los componentes que intervienen e incurren en un coste significativamente menor.

2.2.2. Modelos

Hasta ahora hemos mencionado varias veces la palabra LLM, pero no nos hemos detenido a explicar qué son realmente. Sabemos que son modelos de lenguaje entrenados con una enorme cantidad de datos, pero realmente son un macroconjunto de varios tipos de modelos. A continuación, describimos una taxonomía de LLM que facilitará su clasificación y el análisis del tipo de modelo más conveniente para el caso de uso objeto de estudio.

[10] https://www.llamaindex.ai/

A pesar de que la mayor parte de los LLM actuales se basan en la arquitectura *Transformer*, no todos reconstruyen su implementación exactamente. A grandes rasgos, esta arquitectura consta de dos componentes: el codificador y el decodificador. El primero procesa y representa la información de entrada en una serie de vectores latentes, capturando relaciones complejas entre las palabras mediante mecanismos de autoatención. Mientras tanto, el segundo genera secuencias de salida basándose en estos vectores, decodificando la información de manera autorregresiva para producir texto coherente y contextualizado. Según la arquitectura sobre la que se construyen, distinguimos tres tipos de LLM:

- *Encoder-only*. Modelos Seq2Seq basados en el codificador de la arquitectura *Transformer* que concentran su atención en la secuencia de entrada. En este sentido, estos modelos presentan un rendimiento mayor en tareas de NLU como la clasificación de texto. BERT [19] y *Robustly Optimized BERT Pretraining Approach* (RoBERTa) [35] son ejemplos de este tipo.
- *Decoder-only*. Modelos Seq2Seq basados en el decodificador de la arquitectura *Transformer* capaces de generar la continuación de un texto teniendo en cuenta el contexto anterior. Su comprensión de la secuencia de entrada es, por ello, menor que la de los modelos con codificador, pero presentan gran capacidad para predecir la siguiente palabra más probable. En este sentido, estos modelos adquieren la capacidad de llevar a cabo diferentes tareas durante la fase de entrenamiento, como la respuesta a preguntas, la traducción, el resumen de texto, etc. La serie GPT de OpenAI y *Large Language Model Meta AI* (LLaMA) [36] son modelos *decoder-only*.
- *Encoder-Decoder*. Modelos Seq2Seq que combinan el codificador y el decodificador de la arquitectura original. Son especialmente buenos en tareas en las que existe una relación fuerte entre las secuencias de entrada y de salida, como en la traducción y resumen de texto, donde es fundamental encontrar las relaciones entre los elementos de ambas secuencias. Algunos ejemplos son *Bidirectional and Autoregressive Transformer* (BART) [37] y T5 [24].

Según su disponibilidad, distinguimos tres tipos:

- *De código abierto*. Modelos disponibles para su uso, modificación y distribución de forma gratuita. Generalmente, su entrenamiento y arquitectura son accesibles al público, lo que permite a la comunidad investigadora y a las empresas adaptarlos a sus necesidades. Ejemplos de estos modelos incluyen la serie LLaMA, modelos de Mistral[11] y BLOOM.
- *Open-weights*. Modelos cuyos parámetros han sido publicados y pueden utilizarse libremente, pero cuyo código de entrenamiento o datos de preentrenamiento pueden no ser completamente accesibles o modificables. Este enfoque permite que los modelos sean reutilizados y ajustados por terceros sin necesidad de entrenarlos desde cero. Cabe hacer mención de las plataformas Ollama[12] y Hugging Face[13], que pueden ser utilizadas para acceder a la mayor parte de estos modelos. Un ejemplo de esta categoría es el modelo Mistral-8x7B[14].
- *Propietarios*. Modelos desarrollados y mantenidos por una organización privada que normalmente están sujetos a un coste por su utilización. Además, su código fuente es privado, por

[11] https://mistral.ai/
[12] https://ollama.com/
[13] https://huggingface.co/
[14] https://huggingface.co/mistralai/Mixtral-8x7B-v0.1/

lo que no se puede inspeccionar ni modificar libremente. Suelen estar optimizados para aplicaciones comerciales y ofrecen acceso mediante API, como ocurre con los modelos privados de OpenAI, Anthropic y Google.

Finalmente, los LLM también pueden clasificarse según la intención con la que fueron diseñados. En este sentido, existen dos grandes categorías: los modelos de propósito general y los modelos de propósito específico.

- *De propósito general*. Modelos diseñados para manejar una amplia variedad de tareas dentro del campo del NLP, como generación de texto, traducción automática, resumen de documentos, respuesta a preguntas, entre otras. Ejemplos destacados de esta categoría incluyen GPT-4o [38], Claude 3 y Gemini 2.5, que pueden adaptarse a múltiples aplicaciones gracias a su capacidad para comprender e interpretar el lenguaje en distintos contextos.
- *De propósito específico*. Modelos entrenados específicamente para desempeñar una tarea concreta con un alto grado de precisión. Dentro de esta categoría, se encuentran modelos como BioGPT [39], optimizado para el procesamiento de textos biomédicos, FinGPT [40], centrado en el análisis financiero, o Codex [41], diseñado para generación de código. Además, algunos modelos han sido ajustados mediante *fine-tuning* para entornos conversacionales, como los modelos optimizados para chat, que mejoran su capacidad de interacción con los usuarios en diálogos fluidos y contextuales; por ejemplo, ChatGPT.

Hemos descrito una taxonomía para la clasificación de LLM. Ahora bien, una vez elegido el tipo de modelo que necesitamos, ¿cómo podemos valorar cuál de todos ellos es el adecuado? Para evaluar la destreza de los LLM existen varios *benchmarks* relevantes. Uno de los rankings más populares es el ranking OpenLLM de Hugging Face[15], que compara los LLM de la plataforma en función de varios *benchmarks*. Entre ellos, en la Tabla 2.2 destacamos *Instruction-Following Evaluation* (IFEval), que evalúa la capacidad del modelo para seguir una serie de instrucciones determinada; *Big Bench Hard* (BBH), que evalúa el desempeño del modelo en tareas de NLP en varios dominios, como el razonamiento matemático y la cultura general; y el *Multistep Soft Reasoning* (MuSR), que evalúa tanto la capacidad del modelo para entender el lenguaje como su capacidad de razonamiento. Por su parte, la puntuación media se calcula como la media ponderada de los resultados normalizados de todos los *benchmarks* de Hugging Face, siendo estos más de los ya descritos.

Tabla 2.2: Adaptación del ranking OpenLLM de HuggingFace filtrado por proveedores oficiales y truncado a las cinco primeras posiciones hasta la fecha.

Posición	Modelo	Media	IFEval	BBH	MuSR
1	Qwen/Qwen2.5-72B-Instruct	47.98 %	86.38 %	61.87 %	11.74 %
2	Qwen/Qwen2.5-32B-Instruct	46.60 %	83.46 %	56.49 %	13.50 %
3	mistralai/Mistral-Large-Instruct-2411	46.52 %	84.01 %	52.74 %	17.22 %
4	meta-llama/Llama-3.3-70B-Instruct	44.85 %	89.98 %	56.56 %	15.57 %
5	Qwen/Qwen2-72B-Instruct	43.59 %	79.89 %	57.48 %	17.17 %

[15] https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard

Otro ranking interesante es LiveBench [42], que además de los modelos abiertos, también incluye modelos propietarios y, por tanto, resulta ser una referencia a la hora de decidir qué modelo elegir y compararlos en función de la tarea a realizar. LiveBench contiene un conjunto de 18 tareas de entre seis categorías diferentes donde cada pregunta es correspondida con respuestas de referencia objetivas que permiten la automatización de la evaluación con precisión sin necesidad de utilizar un LLM como juez. En la Tabla 2.3 se muestran los cinco modelos con mayor puntuación en los *benchmarks* de este ranking.

Tabla 2.3: Adaptación del ranking LiveBench truncado a las cinco primeras posiciones hasta la fecha.

Posición	Modelo	Media	Razonamiento	Matemáticas	IFEval
1	gemini-2.5-pro-exp-03-25	82.35 %	89.75 %	90.20 %	80.59 %
2	claude-3-7-sonnet-thinking	76.10 %	87.83 %	79.00 %	81.25 %
3	o3-mini-2025-01-31-high	75.88 %	89.58 %	77.29 %	84.36 %
4	o1-2024-12-17-high	75.67 %	91.58 %	80.32 %	81.55 %
5	qwq-32b	71.96 %	83.50 %	77.82 %	81.83 %

Finalmente, cabe destacar ChatbotArena [43], una plataforma para evaluar los LLM en función de las preferencias humanas. Este ranking se basa en comparaciones anónimas y aleatorias entre modelos, donde los usuarios eligen qué respuesta prefieren. De esta forma, ChatbotArena ofrece un ranking dinámico y orientado a la experiencia real de los usuarios, complementando las métricas automatizadas. En la Tabla 2.4 se muestran las cinco primeras posiciones hasta la fecha. El intervalo de confianza al 95 % representa un rango estimado en el que, con un 95 % de probabilidad, se encuentra el valor real del *Arena Score* del modelo. Este intervalo refleja la incertidumbre asociada al ranking, debido a la variabilidad inherente en las votaciones humanas y la cantidad de comparaciones realizadas.

Tabla 2.4: Adaptación del ranking ChatbotArena truncado a las cinco primeras posiciones hasta la fecha.

Posición	Modelo	Arena Score	95 % CI	Votos
1	Gemini-2.5-Pro-Exp-03-25	1443	+11/-8	3474
2	ChatGPT-4o-latest (2025-03-26)	1408	+11/-12	2676
3	Grok-3-Preview-02-24	1404	+6/-6	10397
4	GPT-4.5-Preview	1398	+6/-7	10907
5	Gemini-2.0-Flash-Thinking-Exp-01-21	1381	+4/-5	22987

Por otro lado, también merece mención el Holistic Evaluation of Language Models[16], desarrollado por Stanford, que busca integrar múltiples dimensiones de evaluación —como capacidad, alineación y eficiencia— en un único marco comparativo. Este enfoque pretende ofrecer una visión más completa y equilibrada del rendimiento de los LLM. Finalmente, para obtener una perspectiva general y actualizada sobre la diversidad de modelos existentes, resulta útil consultar la lista de LLM recopilada en Wikipedia[17], que agrupa y compara decenas de modelos disponibles.

[16] https://crfm.stanford.edu/helm/classic/latest/
[17] https://en.wikipedia.org/wiki/List_of_large_language_models

CAPÍTULO 3

Fundamentos

"Puesto que ignoras lo que te reserva el mañana, procura ser feliz hoy. Coge un ánfora de vino, siéntate a la luz de la luna y bebe, mientras te dices que quizás mañana te busque, en vano, el astro de la noche".

Omar Khayyam

SECCIÓN 3.1

Arquitectura *Transformer*

Como se introdujo en el capítulo anterior, en 2017 se publicó el influyente artículo *Attention is All You Need* [18], que marcó un hito en el desarrollo de modelos de procesamiento del lenguaje natural. En dicho trabajo, los autores propusieron una nueva arquitectura denominada *Transformer*, diseñada específicamente para tareas de traducción automática. Esta arquitectura se basa en un esquema de codificador-decodificador, y su principal innovación radica en el uso exclusivo de mecanismos de atención, prescindiendo por completo de las redes neuronales recurrentes y convolucionales que predominaban hasta ese momento, y dando lugar a la aparición de los modelos que hoy en día conocemos como LLM.

El mecanismo de atención, núcleo del modelo, permite capturar de forma eficiente las dependencias contextuales entre palabras, incluso cuando estas se encuentran alejadas en la secuencia, lo que se traduce en una mayor capacidad para modelar relaciones lingüísticas complejas.

En la arquitectura *Transformer*, el codificador recibe como entrada una secuencia $(w_1, w_2, \ldots, w_n)$ de símbolos en el idioma de origen, y la transforma en una secuencia de vectores $(\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n)$. Estos vectores encapsulan la información contextual de la secuencia y son posteriormente utilizados por el decodificador, que genera, paso a paso, los símbolos en el idioma de destino $(y_1, y_2, \ldots, y_n)$. En cada paso, el decodificador tiene en cuenta tanto la información codificada como los símbolos previamente generados, con el fin de predecir la palabra más probable en la secuencia traducida.

En esta sección introducimos una variante de la arquitectura *Transformer* que consiste en olvidarse por completo del codificador y emplear exclusivamente el decodificador de la arquitectura original para la generación de texto. Y de ahí su nombre, *decoder-only*. Esta es la estrategia empleada por gran parte de los LLM que conocemos. Por ejemplo, los primeros modelos de OpenAI, como GPT, GPT-2 y GPT-3 se basan en esta arquitectura, pero con ligeras modificaciones. A partir de ahora denominamos *Transformer* al modelo basado en decodificador que describimos.

Como se puede observar en la Figura 3.1, a partir de una secuencia de entrada, se obtiene una distribución de probabilidad de todas las palabras de un vocabulario V. Intuitivamente, podemos pensar que para una oración determinada, digamos «Don Quijote de la [...]», la palabra que el modelo debería producir (esto es, asignar la mayor probabilidad) es «Mancha». De este modo, podríamos añadir la palabra «Mancha» al final de la secuencia anterior, reintroducir la secuencia completa para generar la siguiente palabra y así, sucesivamente.

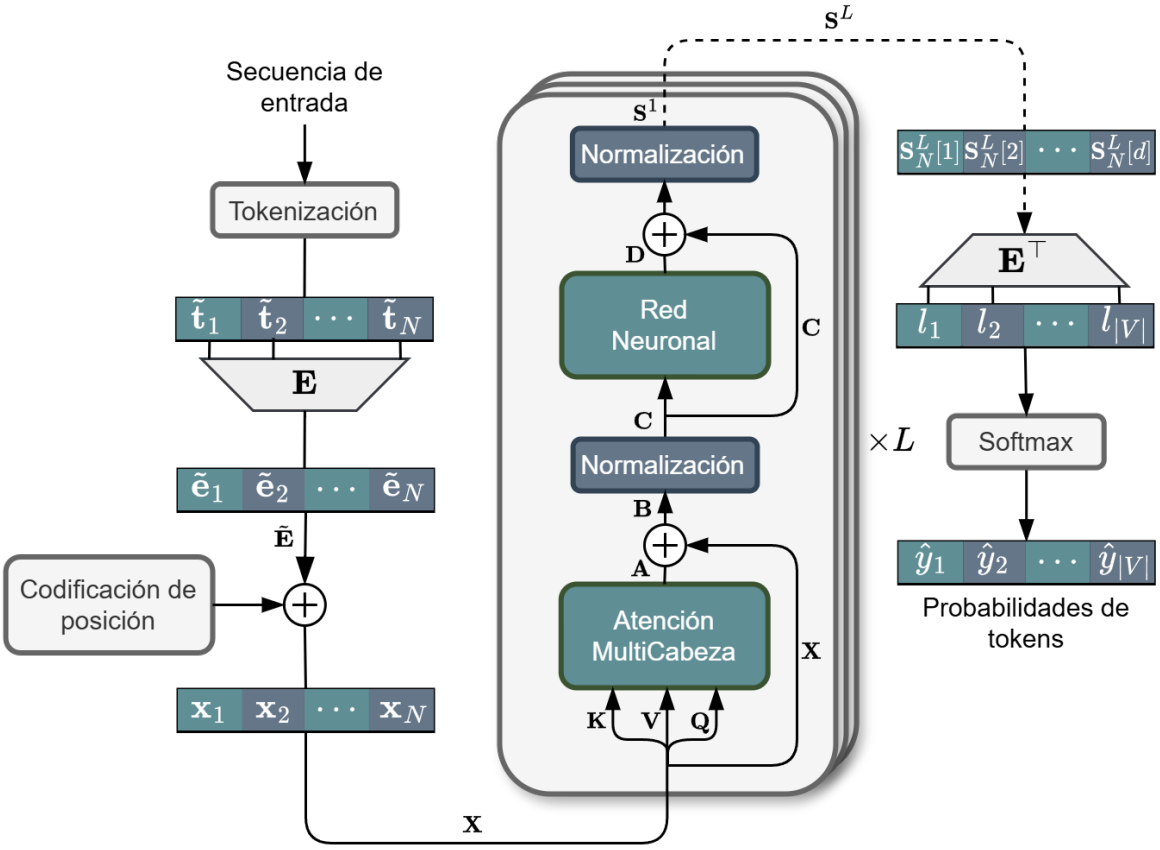

Figura 3.1: Arquitectura de un *Transformer decoder-only* de L capas. La línea discontinua indica que la salida de la primera capa ($\mathbf{S}^1$) actúa como entrada de la siguiente capa, esta última de la siguiente y así, sucesivamente, hasta la L-ésima capa, que retorna $\mathbf{S}^L$.

Formalmente, dados V un vocabulario de tokens y $\mathbf{t} = (t_1, t_2, \ldots, t_N)$ una secuencia de tokens tal que $t_i \in V \ \forall i \in \{1, 2, \ldots, N\}$, se tiene que $(\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{|V|}) = \text{Transformer}(\mathbf{t})$, donde $\sum_{i=1}^{|V|} \hat{y}_i = 1$. Es decir, el *Transformer* genera una distribución de probabilidad sobre V de manera que $\hat{y}_j = P(V_j \mid \mathbf{t})$, donde V_j denota el token que ocupa la posición j-ésima del vocabulario. Asimismo, el token producido por el *Transformer* será $\hat{t} = argmax_{t \in V} P(t \mid \mathbf{t})$. La elección del token más probable se denomina decodificación voraz. En la práctica, este no es el método más habitual, ya que incurre en resultados deterministas, produciendo siempre el mismo token para secuencias de entrada equivalentes. En el próximo capítulo, hablaremos de técnicas de muestreo para modificar esta elección, pero por ahora nos limitaremos a esta definición.

En la Figura 3.1 distinguimos tres partes. En la parte izquierda, se codifica una secuencia de entrada en una secuencia $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ de N vectores de dimensión d. En la parte central encontramos las capas de decodificación, donde a partir de la representación vectorial de la entrada, se producen

otros N vectores de dimensión d. Finalmente, en la parte derecha, se transforman estos N vectores en $|V|$ números, comúnmente denominados *logits*, que representan una puntuación para cada token del vocabulario y que, tras aplicarse la función softmax : $\mathbb{R}^n \rightarrow (0,1)^n$, donde $\text{softmax}(\mathbf{z})_i = e^{z_i} / \sum_{j=1}^{n} e^{z_j}\ \forall i \in \{1, 2, \ldots, n\}$, conforman una distribución de probabilidad sobre V.

A continuación, describimos con detalle cada uno de estos componentes de manera secuencial, esto es, tras describir el componente c_{i-1} con entrada e_{i-1} y con salida e_i, describimos el componente c_i con entrada e_i y salida e_{i+1}. De esta forma, a diferencia de gran parte de la literatura, que se restringe a explicar cada componente de manera independiente, aquí se pretende aunar todos ellos para comprender su funcionamiento claramente de principio a fin.

3.1.1. Tokenización

Hasta ahora, al definir el modelo *Transformer* hemos hablado de tokens, cuando quizás lo más intuitivo hubiese sido hablar de palabras. En la práctica, una secuencia de entrada arbitraria, por ejemplo, «resbalaban las sombras por el muro húmedo del amanecer» se descompone en tokens como sigue: [res, ##bala, ##ban, las, sombra, ##s, por, el, muro, h, ##ú, ##med, ##o, del, ama, ##nec, ##er]. Evidentemente, esta descomposición depende del tokenizador que se utilice (en este caso `bert-base-multilingual-cased`) y del tipo de tokenización. De momento, solo nos interesa tener esta intuición de token, por lo que de ahora en adelante prescindiremos en la medida de lo posible del concepto de palabra. Debemos, por ende, pensar que el vocabulario del lenguaje del modelo, que denotamos V, es el conjunto de tokens que definimos, incluyendo palabras, subpalabras, letras, signos de puntuación, números, etc.

Resumiendo, una secuencia de entrada se traduce en una secuencia de N tokens $\mathbf{t} = (t_1, t_2, \ldots, t_N)$ tal que $t_i \in V\ \forall i \in \{1, 2, \ldots, N\}$.

3.1.2. Representación vectorial

Cada uno de los tokens de la secuencia $\mathbf{t}$ pertenece a V, luego podemos pensar en cada uno de ellos como el índice que le corresponde dentro del vocabulario. Por ejemplo, si el primer token es «res» y este se encuentra en la posición 1244 del vocabulario, entonces $t_1.id = 1244$. Para conseguir que el modelo aprenda a partir de estos tokens, debemos codificarlos de forma que cada token suponga algo más que un escalar. La forma habitual de lograrlo consiste en representar el token como un vector en un espacio vectorial de alta dimensión. Estas representaciones vectoriales reciben el nombre de *embeddings*. Para ilustrar este concepto podemos pensar en un elemento enriquecido por la información inherente a un token, que inicialmente puede ser arbitraria, y que a lo largo de las diferentes capas del *Transformer* va dotándose de información de los tokens que lo rodean. En la siguiente sección se explicará más detalladamente en qué consiste, pero de momento es suficiente con tener esta intuición.

La representación inicial de cada uno de los $|V|$ tokens del vocabulario reside en una matriz de *embeddings* $\mathbf{E} \in \mathbb{R}^{|V| \times d}$, donde la fila i-ésima representa el i-ésimo token de V. Ahora bien, como hemos visto que cada token t_i de la secuencia de entrada se puede representar mediante el índice que le corresponde en el vocabulario, entonces podemos obtener el *embedding* $\tilde{\mathbf{e}}_i \in \mathbb{R}^{1 \times d}$ asociado a $t_i \in \mathbf{t}$ como el producto del vector *one-hot* $\tilde{\mathbf{t}}_i \in \{0,1\}^{1 \times |V|}$ asociado al índice $t_i.id$ (o sea con la

componente $t_i.id$-ésima 1 y el resto ceros) y la matriz $\mathbf{E}$, como se ilustra en la ecuación 3.1.

$$\tilde{\mathbf{t}}_i = (0, \ldots, 0, 1, 0, \ldots, 0) \text{ con } \tilde{\mathbf{t}}_i(j) = \begin{cases} 1 & \text{si } j = t_i.id \\ 0 & \text{en otro caso} \end{cases}$$
$$\tilde{\mathbf{e}}_i = \tilde{\mathbf{t}}_i \cdot \mathbf{E} \tag{3.1}$$

Si extendemos la expresión concatenando los N vectores $\tilde{\mathbf{t}}_i$ asociados a los tokens $t_i \in \mathbf{t}$ en una matriz $\tilde{\mathbf{t}} \in \{0,1\}^{N \times |V|}$, obtenemos una matriz de *embeddings* $\tilde{\mathbf{E}} \in \mathbb{R}^{N \times d}$ asociada a los tokens de entrada, como se ve en la ecuación 3.2.

$$\tilde{\mathbf{E}} = \tilde{\mathbf{t}} \cdot \mathbf{E} \tag{3.2}$$

3.1.3. Codificación de posición

A diferencia de las RNN, el mecanismo de atención que utiliza el *transfomer* no es sensible a la posición que ocupan los tokens dentro de la secuencia. Por tanto, codificar la posición de los tokens dentro de la misma es esencial para que este sea capaz de captar la estructura del lenguaje. Sin esta información, el modelo trataría una frase como un conjunto de palabras sin orden, perdiendo nociones clave como proximidad gramatical o dependencias sintácticas. Tradicionalmente, esto se aborda introduciendo vectores de posición que se suman directamente a los *embeddings* de cada token, lo cual permite que la información posicional influya en los cálculos desde la primera capa de atención. Para ello existen varias alternativas, pero la que se introdujo originalmente se basa en una superposición de funciones seno y coseno que oscilan a distintas frecuencias a lo largo de la dimensión de los *embeddings* (ver ecuación 3.3). En el artículo original, los autores eligieron esta formulación con la hipótesis de que permitiría al modelo aprender con facilidad a atender en función de posiciones relativas. Esto se debe a que, para un desplazamiento fijo k, la codificación en la posición $i+k$ puede expresarse como una función lineal de la codificación en la posición i [18]. Este efecto se ilustra en la Figura 3.2.

$$\mathbf{p}_{ij} = \begin{cases} sin(i/10000^{j/d}) & \text{si } j \text{ es par} \\ cos(i/10000^{(j-1)/d}) & \text{si } j \text{ es impar} \end{cases} \quad \forall i \in \{1, \ldots, N\}\ \forall j \in \{1, \ldots, d\} \tag{3.3}$$

donde i es la posición del token en la secuencia y d es la dimensión de los *embeddings*.

De este modo, el elemento i-ésimo de entrada de la primera capa de decodificación, $\mathbf{x}_i \in \mathbb{R}^{1 \times d}$, se calcula como la suma del i-ésimo *embedding* $\tilde{\mathbf{e}}_i \in \mathbb{R}^{1 \times d}$ y el i-ésimo vector de posición $\mathbf{p}_i \in \mathbb{R}^{1 \times d}$, como se observa en la ecuación 3.4.

$$\mathbf{x}_i = \tilde{\mathbf{e}}_i + \mathbf{p}_i \tag{3.4}$$

Asimismo, aunando todos los vectores de posición $\mathbf{p}_i$ en una matriz $\mathbf{P} \in \mathbb{R}^{N \times d}$ es posible calcular los N valores simultáneamente (ver ecuación 3.5).

$$\mathbf{X} = \tilde{\mathbf{E}} + \mathbf{P} \tag{3.5}$$

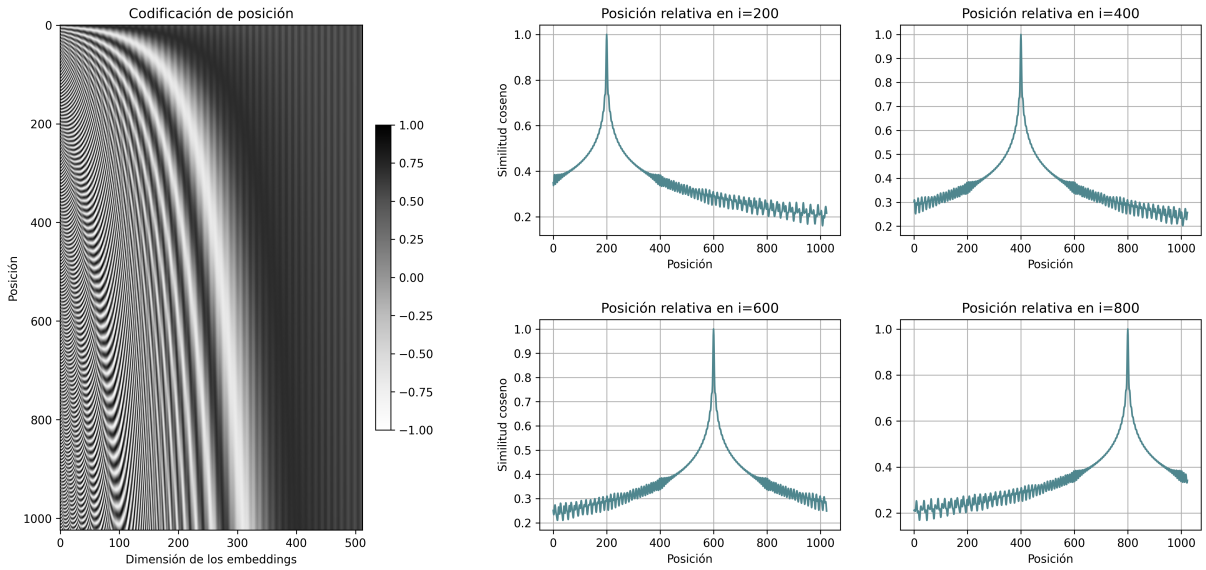

Figura 3.2: Codificaciones posicionales para $N = 1024$ y $d = 512$: a la izquierda, la matriz $\mathbf{P}$; a la derecha, la similitud del coseno entre vectores $\mathbf{p}_i$ y el resto para $i \in \{200, 400, 600, 800\}$. Se observa mayor similitud con posiciones cercanas, que disminuye gradualmente con la distancia.

Cabe destacar que, aunque esta práctica se ha adoptado de forma casi axiomática, su eficacia y uso interno dentro del modelo no son tan transparentes. Si bien hay evidencias empíricas de que los modelos utilizan la información posicional para tareas como traducción, clasificación o generación de texto, también se ha observado que estos no siempre aprovechan estos vectores de posición tal como se plantearon originalmente. En [44], se demuestra empíricamente que el *Transformer* aprende información posicional de forma localizada y dependiente del entrenamiento, es decir, que no generaliza bien esa información entre tareas sin pérdida de rendimiento y, a veces, la reaprende internamente en lugar de usar directamente los vectores de posición originales.

Modelos como BERT, por ejemplo, utilizan vectores de posición aprendidos, lo que sugiere que el modelo reaprende internamente las posiciones en lugar de usar una codificación fija. Además, existen estudios relacionados que exploran enfoques alternativos para las codificaciones posicionales en el *Transformer*, como [45], donde se investiga cómo las codificaciones posicionales aleatorias pueden mejorar la generalización a secuencias de longitud no vista durante el entrenamiento. Este enfoque introduce variabilidad en las codificaciones posicionales, lo que puede interpretarse como una forma de introducir ruido estructurado, y ha demostrado mejorar el rendimiento en tareas de razonamiento algorítmico. En conclusión, incluso cuando la posición no se codifica de manera tradicional, los modelos pueden seguir aprendiendo relaciones posicionales útiles, lo que cuestiona la necesidad de un diseño explícito para este componente.

3.1.4. Atención multicabeza

Dada la matriz $\mathbf{X} \in \mathbb{R}^{N \times d}$ con la representación multidimensional de los tokens de la secuencia de entrada, el flujo alcanza la primera capa de decodificación que, junto con el resto de capas de decodificación, constituye el núcleo del decodificador, tal y como se ve en la parte central de la Figura 3.1. Como veremos en la próxima sección, los *embeddings* son representaciones vectoriales

invariantes de los tokens, es decir, codifican información inherente al token, como sus propiedades semánticas y sintácticas, pero no se ven alterados al ser expuestos a diferentes contextos. En la capa de decodificación el *Transformer* incorpora información contextual en los *embeddings*, de manera que, a medida que estos vectores van atravesando iterativamente cada una de las capas, van enriqueciendo su representación con la información de los tokens que lo preceden.

El primer componente de esta capa consiste en un mecanismo de atención multicabeza enmascarada. Distinguimos, por tanto, tres palabras: «atención», «multicabeza» y «enmascarada». En primer lugar, el concepto de atención consiste en la computación de una representación vectorial de un token atendiendo a los *embeddings* de los tokens de su izquierda calculados en una capa previa. Es decir, el cálculo de la atención $\mathbf{a}_i$ del *embedding* $\mathbf{x}_i$ se define como una función de $\mathbf{x}_{j-k}\ \forall j \in \{i, i+1, \ldots, i+k\}$, donde k es el tamaño de la ventana contextual del modelo. Por simplicidad, asumiremos que k no está acotado. En segundo lugar, la intuición de la atención multicabeza consiste en que la integración de varias cabezas permite que cada una de ellas busque patrones o relaciones diferentes entre los tokens. Por último, que sea enmascarada significa que emplea exclusivamente tokens anteriores en la secuencia, enmascarando los futuros tokens para que no interfieran en el cálculo de las cabezas de atención.

Para ilustrar el procedimiento de computación de cada una de las cabezas de atención necesitamos introducir tres matrices de parámetros $\mathbf{W}_i^Q \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d \times d_k}$ y $\mathbf{W}_i^V \in \mathbb{R}^{d \times d_v}$. Estas matrices derivan de un intento de representar los tres roles diferentes que cada *embedding* desempeña durante el proceso de atención: clave (k), valor (v) y consulta (q). Este método se puede entender por analogía con las tablas de dispersión abierta. Por ejemplo, para la consulta $q =$«mar» y la clave $k =$«color», podríamos tener el valor $v =$«azul», incorporando numéricamente el conocimiento de que la mar es azul. Como hemos señalado anteriormente, el objetivo de integrar varias cabezas radica en tratar de detectar diferentes patrones entre los tokens en cada una de ellas, proyectando linealmente las claves, valores y consultas de manera diferente tantas veces como cabezas, permitiendo al modelo atender a la información desde diferentes subespacios de representación. Por ende, cada cabeza de atención precisa de tres matrices de parámetros diferentes.

A partir de $\mathbf{X}$, se calculan las matrices relativas a las consultas $\mathbf{Q}_i \in \mathbb{R}^{N \times d_k}$, las claves $\mathbf{K}_i \in \mathbb{R}^{N \times d_k}$ y los valores $\mathbf{V}_i \in \mathbb{R}^{N \times d_v}$ de cada cabeza como se muestra en la ecuación 3.6.

$$\mathbf{Q}_i = \mathbf{X} \cdot \mathbf{W}_i^Q, \qquad \mathbf{K}_i = \mathbf{X} \cdot \mathbf{W}_i^K, \qquad \mathbf{V}_i = \mathbf{X} \cdot \mathbf{W}_i^V \tag{3.6}$$

para todo $i \in \{1, 2, \ldots, h\}$, donde h es el número de cabezas.

A continuación, se multiplican las matrices $\mathbf{Q}_i$ y $\mathbf{K}_i^\top$ y se escala el resultado dividiendo por un factor $\sqrt{d_k}$ (ver ecuación 3.7). Después, se enmascara la matriz resultante $\mathbf{valor}_i \in \mathbb{R}^{N \times N}$, tomando valor $-\infty$ en todas las posiciones por encima de la diagonal principal y se aplica la función softmax a cada fila de la matriz resultante para obtener los parámetros sobre los valores. Nótese que, al aplicar esta función, las posiciones enmascaradas tendrán valor $\text{softmax}(-\infty) = 0$. Finalmente, el resultado se multiplica por $\mathbf{V}_i$ para obtener la matriz que representa la cabeza i-ésima $\mathbf{cabeza}_i \in \mathbb{R}^{N \times d_v}$ (ver ecuación 3.8).

$$\mathbf{valor}_i = \frac{\mathbf{Q}_i \cdot \mathbf{K}_i^\top}{\sqrt{d_k}} = \frac{\mathbf{X} \cdot \mathbf{W}_i^Q \cdot (\mathbf{W}_i^K)^\top \cdot \mathbf{X}^\top}{\sqrt{d_k}} \tag{3.7}$$

$$\mathbf{cabeza}_i = \text{softmax}(\text{enmascara}(\mathbf{valor}_i)) \cdot \mathbf{V}_i \tag{3.8}$$

para todo $i \in \{1, 2, \ldots, h\}$. En aras de visualizar la atención, se ha implementado un modelo *Transformer decoder-only* entrenado con el texto de *El Quijote* y un vocabulario de 3800 tokens. En la Figura 3.3 se muestran las ocho cabezas de atención de dicho modelo para la secuencia de entrada «Don Quijote de la». Nótese el efecto de la máscara junto con la función softmax, que anula todas las posiciones por encima de la diagonal principal, obligando al modelo a hacer predicciones de forma autorregresiva. Además, véase cómo en cada una de las cabezas, los valores de atención son diferentes.

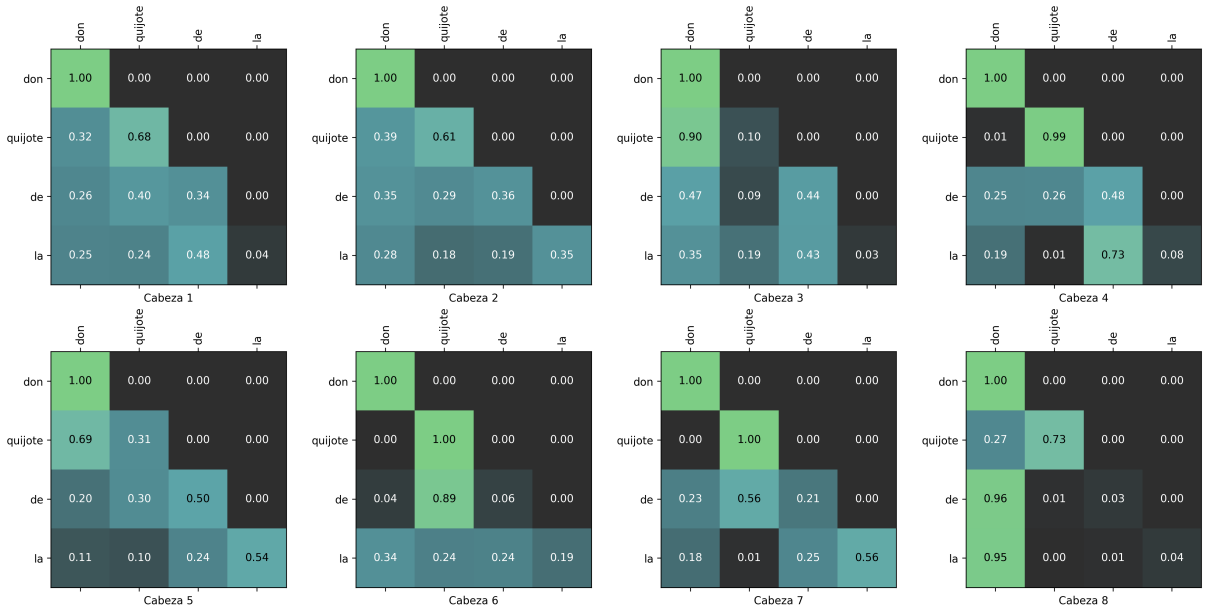

Figura 3.3: Visualización de las ocho cabezas de atención de la última capa de un modelo *Transformer decoder-only* entrenado con el texto de *El Quijote*.

Luego ya solo queda unificar todos los resultados de las h cabezas. Para ello, se concatenan todas las matrices $\mathbf{cabeza}_i$ en una matriz de dimensión $N \times hd_v$ y se multiplica por una última matriz de parámetros $\mathbf{W}^O \in \mathbb{R}^{hd_v \times d}$ para obtener la salida de la capa de atención multicabeza $\mathbf{A} \in \mathbb{R}^{N \times d}$, en la que cada fila se corresponde con un *embedding* mejorado por el mecanismo de atención (ver ecuación 3.9).

$$\mathbf{A} = (\mathbf{cabeza}_1 \mid \cdots \mid \mathbf{cabeza}_h) \cdot \mathbf{W}^O \tag{3.9}$$

3.1.5. Conexión residual y normalización

Como se observa en la Figura 3.1, inmediatamente después de la capa de atención multicabeza hay una conexión residual, proveniente de la entrada de esta capa, o sea $\mathbf{X}$, que es sumada con la matriz de *embeddings* mejorada $\mathbf{A}$ para obtener una matriz $\mathbf{B} \in \mathbb{R}^{N \times d}$, donde cada fila se corresponde con un *embedding* (ver ecuación 3.10).

$$\mathbf{B} = \mathbf{X} + \mathbf{A} \tag{3.10}$$

Esta suma es sucedida por una capa de normalización, con el objetivo de limitar el rango de valores de los *embeddings* para mejorar el rendimiento de la fase de entrenamiento. Esta normalización se realiza

de manera independiente sobre cada *embedding* (por filas), como se muestra en la ecuación 3.11.

$$\mathbf{C}_i = \text{Normalización}(\mathbf{B}_i) = \gamma \frac{\mathbf{B}_i - \mu_i}{\sigma_i} + \beta = \gamma \frac{\mathbf{B}_i - \frac{1}{d}\sum_{j=1}^{d} \mathbf{B}_{ij}}{\sqrt{\frac{1}{d}\sum_{j=1}^{d}\left(\mathbf{B}_{ij} - \frac{1}{d}\sum_{j=1}^{d}\mathbf{B}_{ij}\right)^2}} + \beta \tag{3.11}$$

para todo $i \in \{1, \ldots, N\}$, donde γ y β son parámetros de aprendizaje, μ_i es la media aritmética de los valores de $\mathbf{B}_i$ y σ_i es la desviación típica de los valores de $\mathbf{B}_i$.

3.1.6. Red neuronal

A continuación, la matriz $\mathbf{C} \in \mathbb{R}^{N \times d}$ actúa como entrada de una red neuronal totalmente conectada con una única capa oculta de dimensión d_{rn} con función de activación $\text{ReLU}(x) = \text{máx}(0, x)$, obteniendo una matriz $\mathbf{D} \in \mathbb{R}^{N \times d}$ y, al igual que antes, se aplica la suma del residuo y se normaliza el resultado, como se muestra en la ecuación 3.12.

$$\begin{aligned} \mathbf{D} &= \text{RedNeuronal}(\mathbf{C}) = \text{ReLU}(\mathbf{C} \cdot \mathbf{W}_1^\top + \mathbf{b}_1) \cdot \mathbf{W}_2^\top + \mathbf{b}_2 \\ \mathbf{S}^1 &= \text{Normalización}(\mathbf{C} + \mathbf{D}) \end{aligned} \tag{3.12}$$

donde $\mathbf{W}_1 \in \mathbb{R}^{d_{rn} \times d}$ y $\mathbf{W}_2 \in \mathbb{R}^{d \times d_{rn}}$ son las matrices de parámetros y $\mathbf{b}_1 \in \mathbb{R}^{1 \times d_{rn}}$ y $\mathbf{b}_2 \in \mathbb{R}^{1 \times d}$ son los términos independientes que son sumados mediante *broadcasting* por filas.

3.1.7. Capas de decodificación

Nuevamente en la Figura 3.1 vemos que la capa de decodificación se repite L veces. Esto significa que la salida de la última fase de normalización de la primera capa actúa como entrada de la siguiente capa de decodificación, es decir, como la matriz de *embeddings* $\mathbf{X}$ que se introduce en la capa de atención multicabeza, pero habiendo incorporado la información contextual en ella. Esto es, inicialmente la matriz de *embeddings* original $\mathbf{X}$ se introduce en la primera capa de decodificación y, a partir de entonces, si llamamos $\mathbf{S}^i \in \mathbb{R}^{N \times d}$ a la salida de la capa de decodificación i-ésima, entonces esta capa tiene como entrada $\mathbf{S}^{i-1} \in \mathbb{R}^{N \times d}$ y produce la salida $\mathbf{S}^i$, como se ve en la ecuación 3.13.

$$\begin{aligned} \mathbf{S}^1 &= \text{CapaDecodificación}_1(\mathbf{X}) \\ \mathbf{S}^i &= \text{CapaDecodificación}_i(\mathbf{S}^{i-1}) \ \forall i \in \{2, \ldots, L\} \end{aligned} \tag{3.13}$$

3.1.8. Capa de generación

Por último, dada la salida $\mathbf{S}^L \in \mathbb{R}^{N \times d}$ de la última capa de decodificación, se entra en la última etapa del *Transformer*, como se ilustra en la parte derecha de la Figura 3.1. En esta se extrae la fila N-ésima de la matriz $\mathbf{S}^L$, que representa el *embedding* contextualizado del token N-ésimo, y se utiliza para predecir el siguiente token $t_{N+1} \in V$. Recordemos, de la fase de representación vectorial, que existe una matriz de *embeddings* $\mathbf{E}$ con la representación multidimensional de cada token del vocabulario. En dicha fase obteníamos el *embedding* correspondiente al token mediante el producto del vector *one-hot* de la posición del token en V y $\mathbf{E}$. Por tanto, podemos pensar que

el *embedding* resultante es exactamente el de ese token, donde el valor no nulo del vector *one-hot* designa probabilidad 1 a ese token y probabilidad 0 a todos los demás. Análogamente, si procedemos a la inversa, resulta intuitivo que dado el *embedding* $\mathbf{S}_N^L \in \mathbb{R}^{1\times d}$, al multiplicar por $\mathbf{E}^\top \in \mathbb{R}^{d\times|V|}$, obtendremos una puntuación ($\mathbf{logits} \in \mathbb{R}^{1\times|V|}$) para cada uno los $|V|$ tokens que, tras aplicar la función softmax, dará lugar a una distribución de probabilidad $\hat{\mathbf{y}} \in (0,1)^{1\times|V|}$ sobre todos los tokens del vocabulario. Técnicamente, al multiplicar $\mathbf{S}_N^L$ por $\mathbf{E}$, se está calculando la similitud entre el *embedding* $\mathbf{S}_N^L$ y cada uno de *embeddings* iniciales, lo que justifica la idea de que realmente se está obteniendo una puntuación sobre cada uno de ellos. Este cálculo se ilustra en la ecuación 3.14.

$$\begin{aligned} \mathbf{logits} = (l_1, \ldots, l_{|V|}) &= \mathbf{S}_N^L \cdot \mathbf{E}^\top \\ \hat{\mathbf{y}} = (\hat{y}_1, \ldots, \hat{y}_{|V|}) &= \text{softmax}(\mathbf{logits}) \end{aligned} \tag{3.14}$$

donde $\sum_{i=1}^{|V|} \hat{y}_i = 1$. En la Figura 3.4 se observan los 15 tokens más probables predichos por nuestro modelo para la frase «Don Quijote de la». Cabe notar que, a excepción de «cual» y «c», el resto de tokens son sustantivos de género femenino y número singular, lo que sugiere que el modelo ha aprendido a detectar que los artículos como «la» preceden a sustantivos de este tipo.

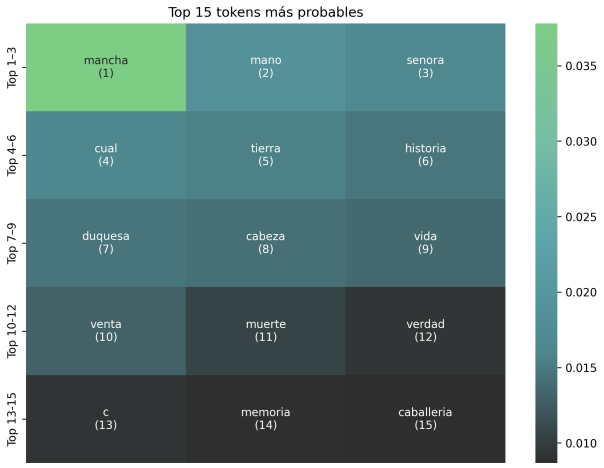

Figura 3.4: Top 15 tokens más probables producidos por un modelo *Transformer decoder-only* entrenado con el texto de *El Quijote* para la secuencia de entrada «Don Quijote de la».

Finalmente, el *Transformer* genera el token $\hat{t} \in V$ con mayor probabilidad, según la ecuación 3.15. En el ejemplo, efectivamente, nuestro *Transformer* predice la palabra que consideramos más probable al inicio de la sección («Mancha»), como se ilustra en la Figura 3.4.

$$\begin{aligned} \hat{t}.id &= \arg\max_i \hat{y}_i \\ \hat{t} &= V_{\hat{t}.id} \end{aligned} \tag{3.15}$$

3.1.9. Fase de entrenamiento

Para concluir con la descripción del *Transformer*, debemos dar una intuición de cómo se entrena este modelo, es decir, cómo se consigue ajustar los parámetros de manera que, después de varias etapas de entrenamiento, sea capaz de generar texto con coherencia, cohesión y adecuación al contexto. Dado un corpus $\mathcal{C}$ de n tokens y habiendo observado los tokens $t_{i-k}, t_{i-k+1}, \ldots, t_i$, el objetivo es encontrar

los parámetros Θ del modelo tales que se maximice la función de verosimilitud de la ecuación 3.16.

$$\mathcal{L}(\mathcal{C}) = \sum_i \log P(t_{i+1} \mid t_{i-k}, t_{i-k+1}, \ldots, t_i; \Theta) \tag{3.16}$$

donde k es el tamaño de la ventana contextual del *Transformer*. En la práctica, esto se consigue entrenando al modelo para minimizar el error de predicción del próximo token en la secuencia de entrenamiento. Como el modelo retorna una distribución de probabilidad sobre los $|V|$ tokens del vocabulario, se utiliza la función de pérdida conocida como entropía cruzada (*cross entropy loss*). Este valor evalúa cómo de cercana es la distribución de probabilidad obtenida por el *Transformer* $\hat{\mathbf{y}}$ y la distribución de probabilidad correcta $\mathbf{y}$. Nótese que esta última asigna probabilidad 1 al siguiente token correcto y 0 al resto, luego el cálculo se restringe a evaluar exclusivamente la probabilidad que el modelo asigna al siguiente token correcto, enmascarando todos los demás tokens. En consecuencia, la función de pérdida en la etapa i del entrenamiento se define según la ecuación 3.17.

$$pérdida(\mathbf{y}_i, \hat{\mathbf{y}}_i) = -\sum_{t \in V} \mathbf{y}_i(t) \log \hat{\mathbf{y}}_i(t) = -\log \hat{\mathbf{y}}_i(t_{i+1}) \tag{3.17}$$

donde cada etapa i se corresponde con la predicción del token de la posición $i+1$-ésima de la secuencia de entrada (t_{i+1}) habiendo observado los tokens $t_{i-k}, \ldots, t_i$. Esto da lugar a un procedimiento iterativo en el que, tras la etapa i, se pasa a la etapa $i+1$ usando la secuencia correcta de tokens $t_{i+1-k}, \ldots, t_{i+1}$ para estimar la probabilidad del token t_{i+2}. El valor final de la función de pérdida para una secuencia de entrenamiento es entonces la media aritmética de las pérdidas computadas en cada etapa, luego los parámetros del modelo son ajustados para minimizar este último valor mediante métodos numéricos.

SECCIÓN 3.2

Representación multidimensional de las palabras

En la sección anterior se introdujo la noción de *embedding* como una representación vectorial de una palabra sobre un espacio multidimensional y se mencionó el papel fundamental que desempeñan en diversos aspectos del NLP, como la recuperación de información y la generación de texto. En esta sección, se explica cómo son construidos.

El objetivo final de los *embeddings* es codificar el significado de las palabras mediante vectores densos de números reales, de forma que las máquinas sean capaces de comprenderlo. En lugar de tratar las palabras como entidades separadas, los *embeddings* capturan sus relaciones semánticas: una palabra puede ser sinónima, antónima, pertenecer al mismo campo semántico, tener una connotación positiva o negativa, o incluso compartir diferentes significados dependiendo del contexto. El proceso de entrenamiento permite que, en cada dimensión de estos vectores, se almacene información sobre características de la palabra. Por ejemplo, después de entrenar un modelo con una gran cantidad de texto, en una de las dimensiones de un *embedding* podría almacenarse la información de que la palabra «montaña» está relacionada con un entorno natural, mientras que en otra dimensión se codificaría si la palabra es singular o plural, o si está asociada a un tipo de sustantivo determinado, como un sustantivo concreto o abstracto, entre otras.

Lo más fascinante de los *embeddings* es que no solo representan las palabras por sí mismas, sino también las relaciones entre ellas. Tomemos, por ejemplo, las palabras «sol» y «luna». Si restamos sus *embeddings*, la diferencia entre ellos no solo nos dará una representación del contraste entre «día» y «noche», sino que también podemos utilizar esa información para establecer analogías. Es decir, del mismo modo que «sol es a día» como «luna es a noche», sus vectores reflejarán esas relaciones semánticas numéricamente.

Supongamos, por un momento, que la palabra «sol» tiene la siguiente representación: $\text{emb}(\text{«sol»}) = (0.3, 0.5, 0.7, \dots)$ y que «luna» tiene esta otra: $\text{emb}(\text{«luna»}) = (0.1, 0.5, 0.7, \dots)$. Si ahora restamos ambos vectores (ver ecuación 3.18), lo que obtendremos no será un número aleatorio, sino un vector que representa la diferencia en las características semánticas entre «sol» y «luna». Al igual que las palabras «rey» y «reina» tienen una relación similar a «hombre» y «mujer», podemos ver que los *embeddings* capturan estas relaciones profundas de una manera que va más allá de la simple comparación de palabras.

$$\text{emb}(\text{«sol»}) - \text{emb}(\text{«luna»}) \approx \text{emb}(\text{«día»}) - \text{emb}(\text{«noche»}) \tag{3.18}$$

La manera actual de codificar estas relaciones entre palabras tiene su origen en la familia de modelos word2vec [15] formada por el modelo *Continuous Bag of Words* (CBOW) y el modelo *skip-gram*, que se construyen sobre la idea de que las palabras que aparecen en contextos similares tienden a tener significados similares. A continuación, se explica el modelo *skip-gram*.

3.2.1. Modelo *skip-gram*

Desde un punto de vista práctico, el modelo *skip-gram* funciona como un clasificador binario. Dada una palabra objetivo w y una o varias palabras de su contexto $c_1, \dots, c_k$, el modelo estima la probabilidad de que esas palabras coaparezcan en el corpus. En otras palabras, intenta predecir si un par (w, c_i) es un ejemplo positivo (extraído del contexto real) o negativo (generado aleatoriamente).

Formalmente, sea V el vocabulario y sea $(w, c_1, \dots, c_k)$ una tupla compuesta por una palabra objetivo $w \in V$ y un conjunto de k palabras de contexto $c_i \in V$. El modelo estima la probabilidad de que esas palabras estén contextualmente asociadas, es decir, la probabilidad de que el ejemplo sea positivo $P(\text{positivo} \mid w, c_1, \dots, c_k)$.

Para ello, se representan las palabras como vectores en un espacio denso de dimensión d. Denotando con $\mathbf{w}$ el vector de la palabra objetivo y con $\mathbf{c}_i$ el vector de una palabra de contexto, se calcula su producto escalar $\mathbf{w} \cdot \mathbf{c}_i$, una medida de similitud que tiende a ser grande cuando ambos vectores presentan valores elevados en la misma dimensión, es decir, cuando las palabras tienen características similares, y pequeño en caso contrario. No obstante, el resultado es un escalar no acotado, luego para limitarlo entre 0 y 1 se utiliza la función logística $\sigma(u) = (1 + e^{-u})^{-1}$. Ahora bien, como el modelo a entrenar es un clasificador binario, solo existen dos clases posibles (positivo y negativo). Por lo tanto, podemos hablar de la probabilidad de que c_i sea un contexto válido para w, es decir, $P(\text{positivo} \mid w, c_i)$, y de que no lo sea $1 - P(\text{positivo} \mid w, c_i)$. Entonces, asumiendo que $c_1, c_2, \dots, c_k$ son variables independientes, se define la probabilidad de que $c_1, \dots, c_k$ esté contextualmente cerca

de w como producto de probabilidades (ver ecuación 3.19).

$$P(\text{positivo} \mid w, c_1, \ldots, c_k) = \prod_{i=1}^{k} P(\text{positivo} \mid w, c_i) = \prod_{i=1}^{k} \sigma(\mathbf{w} \cdot \mathbf{c}_i) \tag{3.19}$$

Ahora bien, ¿cómo consigue el modelo aprender los parámetros para maximizar esta probabilidad? La intuición de la fase de entrenamiento consiste en que el modelo comienza asignando un vector arbitrario a cada palabra y, con el transcurso de las etapas, va modificando su composición para parecerse más a las palabras que están cerca de ella y menos a las palabras que no lo están. De la misma manera que en el *Transfomer*, donde proyectábamos linealmente cada uno de los *embeddings* sobre tres subespacios diferentes en los que cada token adquiría un rol distinto (consulta, clave y valor), en el modelo *skip-gram* cada palabra se representa en función de si actúa como objetivo o como contexto en dos matrices diferentes $\mathbf{\Theta}^W \in \mathbb{R}^{|V| \times d}$ y $\mathbf{\Theta}^C \in \mathbb{R}^{|V| \times d}$, donde d es la dimensión del espacio de representación de los *embeddings*.

Dado un corpus $\mathcal{C}$ de n palabras y una distancia máxima $R \in \mathbb{N}$, para cada palabra objetivo $w \in \mathcal{C}$, se extraen ejemplos positivos (w, c^+) tales que la distancia entre w y c^+, definida como el número de palabras entre ambas en el corpus, es menor o igual que R. Asimismo, para cada $w \in \mathcal{C}$ también se extraen k ejemplos negativos (w, c^-), eligiendo palabras de manera arbitraria como contexto. La función objetivo que maximiza esta probabilidad es la expresada en la ecuación 3.20.

$$\begin{aligned} \mathcal{L}(w, c^+, c_1^-, \ldots, c_k^-) &= P(\text{positivo} \mid w, c^+) \cdot P(\text{negativo} \mid w, c_1^-, \ldots, c_k^-) \\ &= P(\text{positivo} \mid w, c^+) \prod_{i=1}^{k} P(\text{negativo} \mid w, c_i^-) \end{aligned} \tag{3.20}$$

Tomando el logaritmo de la función objetivo y multiplicando por -1, obtenemos la función de pérdida a optimizar (ver ecuación 3.21), que es fácilmente diferenciable respecto de $\mathbf{w}$, $\mathbf{c}^+$ y $\mathbf{c}_i^-$.

$$\begin{aligned} \mathit{pérdida}(w, c^+, c_1^-, \ldots, c_k^-) &= -\log P(\text{positivo} \mid w, c^+) - \sum_{i=1}^{k} \log P(\text{negativo} \mid w, c_i^-) \\ &= -\log P(\text{positivo} \mid w, c^+) - \sum_{i=1}^{k} \log(1 - P(\text{positivo} \mid w, c_i^-)) \\ &= -\log \sigma(\mathbf{w} \cdot \mathbf{c}^+) - \sum_{i=1}^{k} \log(1 - \sigma(\mathbf{w} \cdot \mathbf{c}_i^-)) \\ &= -\log \sigma(\mathbf{w} \cdot \mathbf{c}^+) - \sum_{i=1}^{k} \log \sigma(-\mathbf{w} \cdot \mathbf{c}_i^-) \end{aligned} \tag{3.21}$$

La función *pérdida* se minimiza iterativamente mediante descenso de gradiente, actualizando los parámetros de $\mathbf{\Theta}^W \in \mathbb{R}^{|V| \times d}$ y $\mathbf{\Theta}^C \in \mathbb{R}^{|V| \times d}$ en cada iteración. Finalmente, se suman las matrices de parámetros resultantes para obtener la matriz de *embeddings* $\mathbf{E} \in \mathbb{R}^{|V| \times d}$ (ver ecuación 3.22) con la representación vectorial del significado de cada palabra de V sobre un espacio de dimensión d.

$$\mathbf{E} = \mathbf{\Theta}^W + \mathbf{\Theta}^C \tag{3.22}$$

CAPÍTULO 4

Metodología

"ELIZA: Hello. How are you feeling today?
Me: I am worried.
ELIZA: How long have you been worried?
Me: Since I started writing this work.
ELIZA: I see. And what does that tell you?"

ELIZA y yo

La conversación presentada arriba corresponde a una interacción que mantuve con ELIZA [6], uno de los primeros asistentes virtuales desarrollados en la historia. ELIZA es un sistema muy básico, diseñado a partir de técnicas de detección de patrones, que simula el rol de un psicoterapeuta con orientación rogeriana.

En este capítulo se aborda la construcción de un asistente virtual mucho más avanzado, aprovechando las capacidades generativas de los LLM y técnicas de recuperación de información. En particular, se explica cómo diseñar un asistente virtual basado en RAG. La exposición se centra en una descripción conceptual y abstracta de las distintas fases que componen un sistema RAG, así como de los parámetros clave que influyen en su rendimiento. El objetivo es proporcionar un marco general que pueda adaptarse a diferentes contextos de aplicación. En los capítulos siguientes, se profundizará en el diseño específico, la implementación técnica y la evaluación del asistente virtual desarrollado para el caso de estudio que motiva este trabajo.

La adopción de un sistema basado en RAG responde, en general, a la necesidad de abordar casos en los que la información requerida para responder adecuadamente a las consultas del usuario proviene de fuentes privadas o de un dominio específico. Dichos contenidos no suelen estar presentes en los datos con los que fueron entrenados los LLM, por lo que el conocimiento del modelo resulta insuficiente.

A esta limitación se suma otra característica técnica clave de los LLM: su ventana de contexto, es decir, la cantidad máxima de información que pueden procesar en una sola entrada. Esta capacidad es limitada, lo que impide proporcionar al modelo la totalidad del conocimiento disponible de forma directa. Incluir grandes volúmenes de texto en cada consulta no solo es inviable por razones técnicas, sino también ineficiente desde el punto de vista computacional.

Frente a estas restricciones, el enfoque RAG ofrece una solución práctica y eficaz. En lugar de llevar a cabo un proceso de *fine-tuning* —proceso complejo y de elevado coste que implica reentrenar el modelo con datos específicos—, se opta por enriquecer dinámicamente el contexto del modelo. Esto se logra mediante la recuperación automatizada de los fragmentos más relevantes de información en función de cada consulta. De este modo, el modelo puede generar respuestas precisas y contextualizadas, sin necesidad de haber incorporado previamente dicho conocimiento en su entrenamiento.

Formalmente, lo que se pretende es desarrollar un sistema capaz de encontrar una respuesta $\mathcal{R}$ contexualmente relevante para una pregunta $\mathcal{P}$ de dominio específico. Para ello, se precisa de una base de conocimiento $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$ formada por n documentos con información relativa a dicho dominio. El objetivo es, por ende, seleccionar un subconjunto suficientemente pequeño de documentos $\{d_{i_1}, d_{i_2}, ..., d_{i_k}\} \subset \mathcal{D}$ con $i_j \in \{1, 2, \ldots, n\}$, de forma que el modelo sea capaz de responder adecuadamente a la pregunta $\mathcal{P}$.

Un sistema RAG consta de tres fases: indexación, recuperación y generación. A continuación, se describen todas ellas, introduciendo los componentes que las constituyen y sus parámetros de mayor trascendencia.

SECCIÓN 4.1

Fase de indexación

En la fase de indexación se lleva a cabo la recopilación y organización de todo el conocimiento disponible relacionado con el dominio específico para el que se pretende desarrollar el sistema. Esta información se almacena en una base de conocimiento en texto plano, ya que es el tipo de entrada que los modelos de lenguaje monomodales —que procesan y generan un único tipo de dato— están diseñados para procesar directamente.

Como se observa en la Figura 4.1, los datos que nutren esta base de conocimiento pueden proceder de fuentes con diferentes niveles de estructuración. Por un lado, se encuentran los datos no estructurados, como ficheros de texto, archivos de vídeo, grabaciones de audio o imágenes. Por otro lado, existen datos semiestructurados, como aquellos contenidos en documentos HTML, XML, JSON o PDF. Finalmente, también se consideran los datos estructurados, como los que se hallan en grafos de conocimiento o bases de datos relacionales.

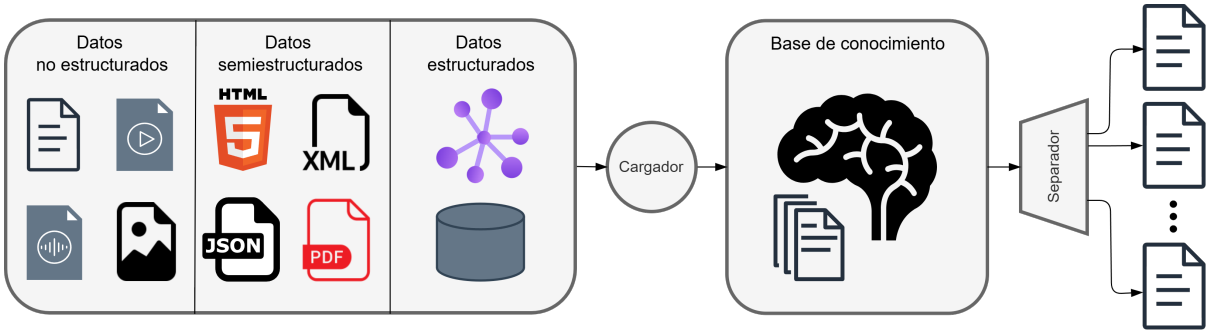

Figura 4.1: Primera parte de la fase de indexación. Se extrae el texto de las fuentes de datos y se divide en documentos.

La transformación de todos estos datos heterogéneos en texto plano recae sobre un componente denominado cargador. Este componente es una entidad abstracta encargada de adaptar cada tipo de contenido a un formato textual uniforme. Se le considera abstracto porque cada tipo de archivo requiere un proceso específico de conversión. Por ejemplo, la lectura de un archivo de texto puede resolverse simplemente mediante una codificación estándar como UTF-8, mientras que la interpretación de una imagen exige un procedimiento más complejo que incluye su descripción mediante modelos multimodales o incluso mediante intervención humana, para luego volcar esa descripción en

texto plano. En el caso de los archivos HTML, el proceso implica realizar un análisis sintáctico de las etiquetas para conservar únicamente el contenido textual.

El tratamiento de datos semiestructurados presenta ciertas complicaciones adicionales. Una de las principales dificultades radica en el proceso de segmentación textual, que se describirá más adelante, y que podría distorsionar la estructura interna de elementos como las tablas. Esta distorsión puede derivar en inconsistencias al recuperar información en fases posteriores del sistema, afectando la coherencia y precisión de las respuestas generadas por el modelo.

Una vez que toda la información ha sido transformada a texto plano, se conforma así la base de conocimiento sobre la que operará el sistema. Dado que el volumen de documentos generados suele ser considerable —pues de lo contrario no tendría sentido aplicar técnicas basadas en RAG—, se recurre a un proceso de segmentación textual. Esta técnica permite dividir los documentos en fragmentos de menor tamaño, con el fin de que cada uno pueda ser procesado eficientemente dentro de los límites de la ventana contextual del LLM utilizado. El componente que lleva a cabo este proceso se denomina separador de texto y resulta imprescindible ajustar sus parámetros para optimizar su desempeño. Entre ellos destacan el tamaño de los fragmentos y el solapamiento entre fragmentos. El tamaño de los fragmentos influye directamente en la calidad de recuperación. Fragmentos demasiado pequeños pueden ocasionar la pérdida de información contextual, afectando la precisión del modelo al responder. Por el contrario, fragmentos excesivamente grandes pueden introducir una sobrecarga de información, dificultando que el modelo se enfoque en los extractos más relevantes. El solapamiento entre fragmentos, por su parte, se emplea para evitar cortes abruptos dentro de los bloques de información, fomentando una transición más fluida entre los mismos.

Una vez que disponemos de una lista de documentos de un tamaño asumible por el modelo, podemos proceder con la segunda parte de la fase de indexación: la creación de la base de datos vectorial. El objetivo de la siguiente fase (recuperación) radica en la selección de los documentos más relevantes para un consulta concreta. Como hemos visto en el capítulo anterior, representar las palabras como secuencias de símbolos no permite a las máquinas comprender sus propiedades semánticas. En este caso, tenemos el mismo problema, pero, en lugar de con palabras, con documentos de varias palabras. Una solución pobre consiste en calcular la media de todos los *embeddings* de cada palabra del documento, conocida como *mean pooling*. Una estrategia alternativa es el *max pooling*, que genera el *embedding* del documento asignando a cada dimensión i el valor máximo entre las componentes i-ésimas de los *embeddings* de todas las palabras del mismo. Por ejemplo, para la oración «hoy es jueves», su *embedding* se calcula según la ecuación 4.1.

$$\begin{aligned} \text{emb}(\text{«hoy»}) &= (0.2, 0.4, 0.2, \mathbf{0.2}) \\ \text{emb}(\text{«es»}) &= (\mathbf{0.3}, 0.1, \mathbf{0.4}, 0.1) \\ \text{emb}(\text{«jueves»}) &= (0.2, \mathbf{0.5}, 0.3, 0.1) \\ \text{emb}(\text{«hoy es jueves»}) &= (0.3, 0.5, 0.4, 0.2) \end{aligned} \tag{4.1}$$

En la práctica, estas técnicas tienden a no utilizarse, sino que se emplean técnicas más sofisticadas como *Sentence-BERT* (SBERT) [46], una modificación de BERT que utiliza redes siamesas —una arquitectura donde dos o más ramas comparten los mismo pesos— para derivar *embeddings* de

oraciones con la suficiente información semántica para permitir compararlas mediante similitud del coseno. En la Figura 4.2 se ilustra este último proceso de la fase de indexación, donde los documentos se embeben en vectores y se almacenan en pares (documento, *embedding*) en una base de datos vectorial para su posterior recuperación.

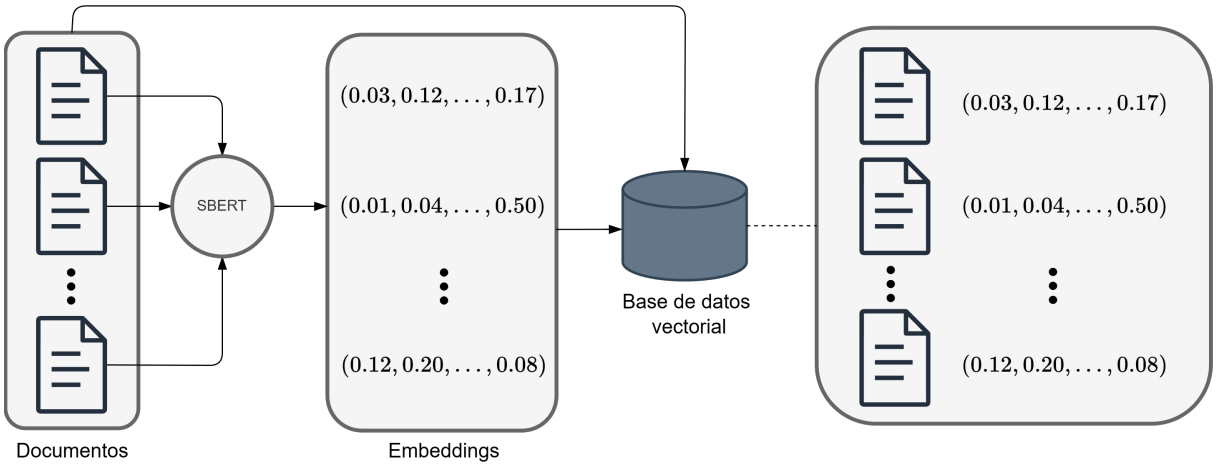

Figura 4.2: Segunda parte de la fase de indexación. A partir de los documentos, se generan sus *embeddings* y se almacenan en una base de datos vectorial.

SECCIÓN 4.2

Fase de recuperación

La fase de recuperación actúa como el puente entre la consulta del usuario y el conocimiento almacenado. Como se observa en la Figura 4.3, al igual que los documentos, la consulta se transforma en su correspondiente representación vectorial mediante un modelo de *embeddings*, como SBERT, lo que permite identificar y extraer los documentos más relevantes de la base de datos vectorial. Aquí es donde entra en juego el recuperador, usualmente integrado en la base de datos vectorial, que compara el *embedding* $\text{emb}(\mathcal{P})$ correspondiente a la pregunta con los *embeddings* de cada documento $\text{emb}(d)$ $\forall d \in \mathcal{D}$ y retorna los documentos más similares a la misma. Dos de los hiperparámetros más relevantes de este componente son el tipo de búsqueda y el número de documentos recuperados.

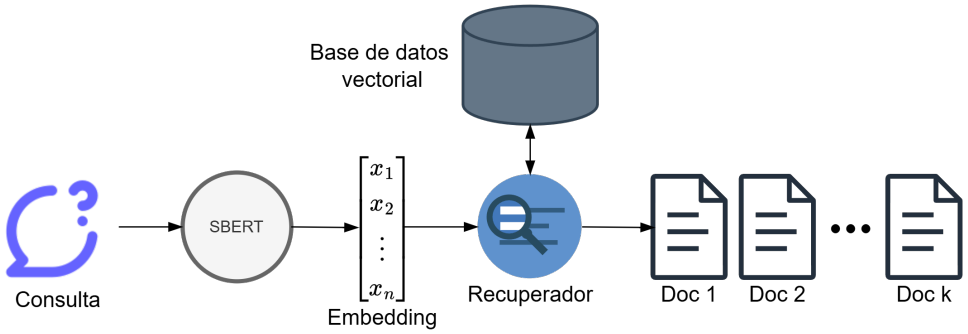

Figura 4.3: Fase de recuperación. A partir de la consulta, se buscan los documentos más relevantes por similitud entre sus *embeddings* y el *embedding* de la consulta.

El tipo de búsqueda más comúnmente utilizado es la similitud del coseno, que calcula el coseno del ángulo formado por dos vectores. En el capítulo 3 veíamos que en el modelo *skip-gram* de word2vec

se medía la similitud entre dos *embeddings* mediante su producto escalar con el pretexto de que este tiende a ser grande cuando ambos vectores presentan valores elevados en dimensiones correspondientes (ergo, características similares), y viceversa. No obstante, las palabras más frecuentes o centrales en el corpus suelen adquirir vectores de mayor norma durante el entrenamiento. Por tanto, dos *embeddings* **u** y **v** también pueden diferir en magnitud, incurriendo en que su producto escalar mezcle efectos de dirección (semejanza semántica) y de longitud (influida por frecuencia). Para eliminar este sesgo y centrarse únicamente en la orientación de los vectores, se normaliza el producto escalar dividiendo por el producto de las normas de **u** y **v**, garantizando que el valor de similitud dependa exclusivamente de la dirección y no de la magnitud de los mismos. Esta expresión es equivalente al coseno del ángulo que forman ambos vectores, luego de ahí su nombre (ver ecuación 4.2).

$$\frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \cdot \|\mathbf{v}\|} = \frac{\sum_{i=1}^{n} u_i \, v_i}{\sqrt{\sum_{i=1}^{n} u_i^2} \, \sqrt{\sum_{i=1}^{n} v_i^2}} = \cos\theta \tag{4.2}$$

donde θ es el ángulo que forman **u** y **v**, y n la dimensión de los *embeddings*.

Una alternativa es el método de búsqueda por máxima relevancia marginal (MMR), que diversifica los documentos recuperados, con objeto de reducir la redundancia e incrementar la cobertura de los diferentes aspectos de la consulta. Esto se consigue encontrando los documentos cuyos *embeddings* tienen mayor similitud del coseno con el *embedding* de la consulta, añadiéndolos iterativamente y penalizando los documentos más similares a los ya seleccionados.

En cuanto al número de documentos recuperados, resulta intuitivo pensar que una mayor cantidad de información recuperada, es decir, un mayor número de documentos seleccionados, supone una mejora en la calidad de las respuestas obtenidas por el LLM. Sin embargo, un exceso de información puede introducir más ruido, empeorando la percepción de la información relevante.

Inmediatamente después de que el recuperador seleccione los k documentos de mayor relevancia, estos se inyectan como contexto en el *prompt* del LLM. Asimismo, es fundamental proveer al modelo de las instrucciones necesarias para que realice correctamente su tarea. El LLM debe entender que tiene el rol de asistente virtual de una aplicación concreta y que debe responder a las preguntas que se le hacen de acuerdo con el contexto que se le proporciona. Estas instrucciones, junto con otras que se consideren necesarias —relativas a temas de privacidad o revelación de datos, por ejemplo—, se deben recoger en el *prompt* de sistema. En la Figura 4.4 se define un *prompt* de sistema básico, instruyendo al modelo sobre la tarea que debe realizar. Además, se muestra cómo se incorporan los k documentos como contexto.

Prompt

Sistema

```
Eres el asistente virtual de <aplicación>. Estás integrado en la aplicación para
responder preguntas del usuario acerca de <caso de uso>. Utiliza los siguientes
extractos de contexto para responder a la pregunta. Si no sabes la respuesta, limítate
a decir que no lo sabes.
```

Contexto

```
Documento 1: ...
Documento 2: ...
...
Documento k: ...
```

Consulta

```
<Pregunta del usuario>.
```

Figura 4.4: Ejemplo de un *prompt* para un asistente virtual basado en RAG.

Otro factor trascendental en la fase de recuperación es la manera en que se incorporan los documentos al *prompt* del LLM. La estrategia básica, llamada *stuff*, consiste en incorporar los documentos de forma íntegra, uno después de otro, tal cual son recuperados (como en la Figura 4.4). Una alternativa más sofisticada es la estrategia *map-reduce*, que itera sobre todos los documentos seleccionados por el recuperador, preguntando al LLM si el documento particular tiene relación con la pregunta formulada. En caso afirmativo, se resume dicho documento y se añade al *prompt*. De lo contrario, se descarta el documento. De esta forma, se obtiene un *prompt* más escueto que presenta exclusivamente la información más relevante. Otra alternativa es la estrategia *refine*, que construye la respuesta de manera iterativa, procesando los documentos uno a uno. Por cada documento, se consulta al LLM proporcionándole tanto la consulta como la última versión de la respuesta parcial, que se actualiza con cada iteración. Dado que en cada paso solo se incorpora un documento al *prompt*, esta estrategia resulta especialmente útil cuando el número de documentos supera la ventana de contexto del modelo. No obstante, puede presentar limitaciones en tareas que requieren integrar simultáneamente información dispersa o altamente interconectada entre documentos.

SECCIÓN 4.3

Fase de generación

La fase de generación es la tercera y última fase de un sistema RAG. En esta, el LLM, alimentado con la información recuperada, elabora una respuesta precisa y contextualizada. Dado que los parámetros del LLM son fijos, para una misma consulta el modelo genera la misma distribución de probabilidad en cada paso de generación, dando lugar a respuestas invariantes. En la sección 3.1 se describió este carácter determinista de los LLM que producen siempre el token más probable, es decir, que emplean la estrategia de decodificación voraz. Para diversificar las respuestas e incentivar la naturalidad de la conversación, se realiza una selección aleatoria de entre los tokens de una muestra concreta. Para determinar esta muestra existen tres técnicas de muestreo fundamentales: muestreo por temperatura, muestreo *top-p* o *nucleus sampling* y muestreo *top-k*.

El concepto de muestreo por temperatura puede entenderse mejor si se piensa en cómo actúa un sistema físico sometido a distintos niveles de energía térmica. Cuando la temperatura es alta, el sistema tiende a comportarse de forma más caótica, permitiéndose transitar libremente entre una amplia variedad de configuraciones. En cambio, a temperaturas bajas, el sistema se vuelve más estable y restringe sus movimientos, concentrándose en opciones más ordenadas y energéticamente favorables. De forma análoga, en los modelos de lenguaje, una temperatura elevada permite respuestas más diversas y menos predecibles, mientras que una baja favorece elecciones más conservadoras y repetitivas. En la práctica, el muestreo por temperatura se reduce a la alteración de un parámetro $T > 0$ que en la ecuación 3.14 se obvió por no tener ningún efecto —pues la selección no era aleatoria, sino absoluta. Antes de convertir los *logits* en una distribución de probabilidad $\hat{\mathbf{y}}$ sobre el vocabulario V del modelo, estos se dividen por este factor T, de forma que $\hat{\mathbf{y}}$ se calcula según la ecuación 4.3,

$$\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{|V|}) = \text{softmax}\left(\frac{\mathbf{logits}}{T}\right) = \text{softmax}\left(\frac{(l_1, l_2, \ldots, l_{|V|})}{T}\right) \tag{4.3}$$

donde cada componente $\hat{y}_i$ está definida como en la ecuación 4.4, de modo que $|l_i|/T \to \infty$ cuando $T \to 0$, es decir, que a medida que se reduce la temperatura, más grandes se hacen los valores que se pasan a la función softmax. Por ende, un efecto inmediato al reducir la temperatura es que los valores altos tienden a 1 y los bajos a 0, incrementando la probabilidad de los tokens más probables en detrimento de la de los menos probables. Por el contrario, si se utiliza una temperatura mayor que 1, los *logits* se suavizan, y la función softmax produce una distribución más plana. Esto significa que se asigna más probabilidad a tokens menos frecuentes, fomentando la diversidad en las respuestas del modelo.

$$\hat{y}_i = \frac{\exp\left(\frac{l_i}{T}\right)}{\sum_{j=1}^{|V|} \exp\left(\frac{l_j}{T}\right)} \quad \forall i = \{1, 2, \ldots, |V|\} \tag{4.4}$$

Por su parte, el muestreo *top-p* limita la selección a los tokens más probables cuya probabilidad agregada no supere un umbral predefinido p. Es decir, sea $\mathbf{c}$ el contexto que el LLM utiliza para predecir el siguiente token y dada una lista de todos los tokens del vocabulario ordenados de mayor a menor probabilidad, se selecciona el conjunto de tokens más grande $\{t_1, \ldots, t_n\} \subseteq V$ tal que $\sum_{i=1}^{n} P(t_i \mid \mathbf{c}) \leq p$, donde $P(t_i \mid \mathbf{c})$ es la probabilidad que el LLM asigna al token t_i en $\hat{\mathbf{y}}$.

Por último, el muestreo *top-k* limita la selección a los k tokens más probables. Para ello, estos $t_1, t_2, \ldots, t_k$ tokens se redistribuyen para formar una distribución de probabilidad sobre los k tokens, de forma que $\sum_{i=1}^{k} P(t_i \mid \mathbf{c}) = 1$. Cabe señalar que si tomamos $k = 1$, estamos en la situación original, donde se elegía el token más probable. Por el contrario, a medida que k crece, la probabilidad de elegir el token más probable se hace cada vez más pequeña, favoreciendo la generación de texto más creativo y diverso.

En conclusión, la modulación de los parámetros de generación, como la temperatura, *top-k* y *top-p*, permite controlar el grado de aleatoriedad en las respuestas generadas por el modelo de lenguaje. En general, un incremento en estos valores puede fomentar la diversidad en las respuestas, haciendo que el modelo explore un rango más amplio de posibilidades léxicas y semánticas. Sin embargo, un exceso de aleatoriedad puede comprometer la coherencia y precisión del texto generado. Específicamente, temperaturas demasiado elevadas o configuraciones poco restrictivas de *top-k* o *top-p* pueden dar lugar a secuencias carentes de sentido o inconsistentes, ya que se reduce la influencia de los tokens con mayor probabilidad en cada paso de la generación.

Por otro lado, al restringir el espacio de posibles selecciones, estos parámetros también contribuyen a generar respuestas más centradas, coherentes y alineadas con la información relevante disponible, lo cual es especialmente importante en la mitigación de alucinaciones —es decir, la producción de afirmaciones incorrectas o sin respaldo en el contexto.

En el marco de un sistema RAG, donde la calidad de las respuestas del LLM depende críticamente de su capacidad para fundamentarse en los documentos recuperados, la aparición de alucinaciones puede derivar en consecuencias indeseables. Entre ellas se incluyen interpretaciones erróneas del contexto, recomendaciones equivocadas al usuario o respuestas que contradicen la información recuperada. Por tanto, una calibración cuidadosa de los parámetros de muestreo es crucial para garantizar que la generación se mantenga fiel al contenido relevante, reduciendo así el riesgo de desinformación.

CAPÍTULO 5

Diseño e implementación

"Any program is only as good as it is useful."

Linus Torvalds

En el presente capítulo se retoma el caso particular que motiva la construcción del asistente virtual: una aplicación destinada a la gestión de flotas de vehículos. Se exponen los principales requisitos funcionales que debe satisfacer el asistente en este contexto, así como los *frameworks* y modelos seleccionados para su desarrollo, detallando igualmente su modo de operación desde una perspectiva de arquitectura de software. Finalmente, se presenta una descripción detallada de la herramienta desarrollada en el marco de este trabajo, la cual integra tanto la articulación funcional de un asistente conversacional como un *framework* de evaluación diseñado para validar distintas configuraciones de los parámetros implicados en la construcción de un sistema RAG, tal y como fueron definidos en el capítulo anterior.

SECCIÓN 5.1

Diseño

5.1.1. Requisitos del asistente virtual

En el contexto de una aplicación destinada a la gestión de flotas de vehículos, y considerando la complejidad inherente a algunas de sus funcionalidades, la aplicación consta de un manual de usuario en formato PDF, complementado por diversos vídeos tutorizados. Esta documentación tiene como propósito ofrecer a los usuarios un recurso de consulta accesible para resolver las dudas que puedan surgir durante el uso de la aplicación. Asimismo, cada usuario cuenta con el respaldo de un equipo de soporte técnico especializado.

Sin embargo, dada la considerable extensión de los materiales de ayuda, es frecuente que los usuarios prefieran recurrir directamente a la asistencia del personal de soporte. Esta situación supone una carga significativa para los equipos, cuyo tiempo y recursos son limitados. Con el fin de optimizar la gestión de estos recursos y liberar al personal para tareas de mayor valor estratégico, se plantea el desarrollo de un asistente virtual que pueda atender con eficacia las consultas de los usuarios.

El objetivo principal de este asistente consiste en ofrecer respuestas precisas y oportunas sobre el funcionamiento de la aplicación. Así, por ejemplo, ante la necesidad de configurar una alarma que se active cuando un conductor exceda un determinado tiempo sin comunicar, el usuario podrá formular su consulta al asistente y obtener la orientación necesaria. Considerando la diversidad geográfica de

los usuarios, que incluye tanto hablantes de inglés como de español, el asistente deberá ser capaz de comprender y responder en ambos idiomas.

Este proyecto persigue diseñar, implementar y evaluar el asistente virtual, con vistas a su futura integración en la aplicación. Para ello, se contempla una arquitectura que consolide todos los componentes de un sistema RAG, explorando diversas configuraciones de parámetros y evaluando la calidad de las respuestas mediante equipos de anotación humana y métricas automáticas específicas.

5.1.2. Selección del *stack* tecnológico

La naturaleza del problema abordado sugiere la elección de un sistema RAG como base para la implementación del asistente virtual. A partir del estudio comparativo expuesto en la sección 2.2, se selecciona LangChain como marco de desarrollo, dada su notable capacidad para modificar componentes a bajo nivel y su reconocida versatilidad en la integración de distintos módulos.

En cuanto a los modelos de lenguaje, se opta por las soluciones ofrecidas a través de Google AI Studio. A diferencia de otras alternativas propietarias, esta plataforma proporciona una API gratuita, con un límite diario de consultas lo suficientemente amplio para permitir la experimentación y validación del sistema sin restricciones significativas. Además, los modelos disponibles son multilingües, lo que resulta idóneo para atender los requisitos establecidos. Concretamente, se emplean los modelos Gemini 1.5 Flash y Gemini 2.0 Flash, actualmente los más potentes disponibles en modalidad gratuita. Para la generación de *embeddings*, se selecciona el modelo de Google identificado como embedding-001, garantizando así la coherencia tecnológica y optimizando la calidad de las representaciones semánticas.

5.1.3. Diseño de la herramienta propuesta

Con el objetivo de desarrollar una solución reutilizable y adaptable a diferentes dominios, se ha construido una herramienta basada en *Command Line Interface* (CLI), denominada `ragbot`, que abstrae las operaciones esenciales asociadas tanto al uso como a la evaluación de asistentes virtuales basados en arquitecturas RAG. Esta aproximación permite desacoplar la lógica de configuración, ejecución y análisis del asistente del caso de uso concreto, favoreciendo así su portabilidad y la replicabilidad experimental.

La herramienta ofrece dos comandos principales:

- **`chat`**: permite iniciar una sesión interactiva con el asistente virtual, configurando dinámicamente los parámetros del sistema RAG, tales como el modelo de lenguaje, el modelo de *embeddings*, el tamaño de los fragmentos textuales, la estrategia de recuperación y otros componentes relevantes.
- **`evaluate`**: permite ejecutar procesos de evaluación automatizada sobre configuraciones específicas del asistente, apoyándose en métricas cuantitativas y evaluaciones cualitativas mediante el uso de modelos LLM como jueces.

En la Figura 5.1 se ilustra el flujo operativo de alto nivel del asistente virtual durante una sesión interactiva iniciada mediante el comando `chat`. Tras la llamada inicial desde la CLI (paso 1), se realiza el proceso de indexado de documentos a partir de la base de conocimiento, a través del

componente indexador y el modelo de *embeddings* (pasos 2–7). Una vez completada esta fase, el sistema entra en un bucle de interacción en el que, por cada consulta del usuario, se calcula su representación semántica (pasos 10–12), se recuperan los documentos más relevantes (pasos 13–14) y se genera la respuesta final mediante el modelo de lenguaje (pasos 15–17). Este proceso finaliza cuando el usuario deja de emitir consultas.

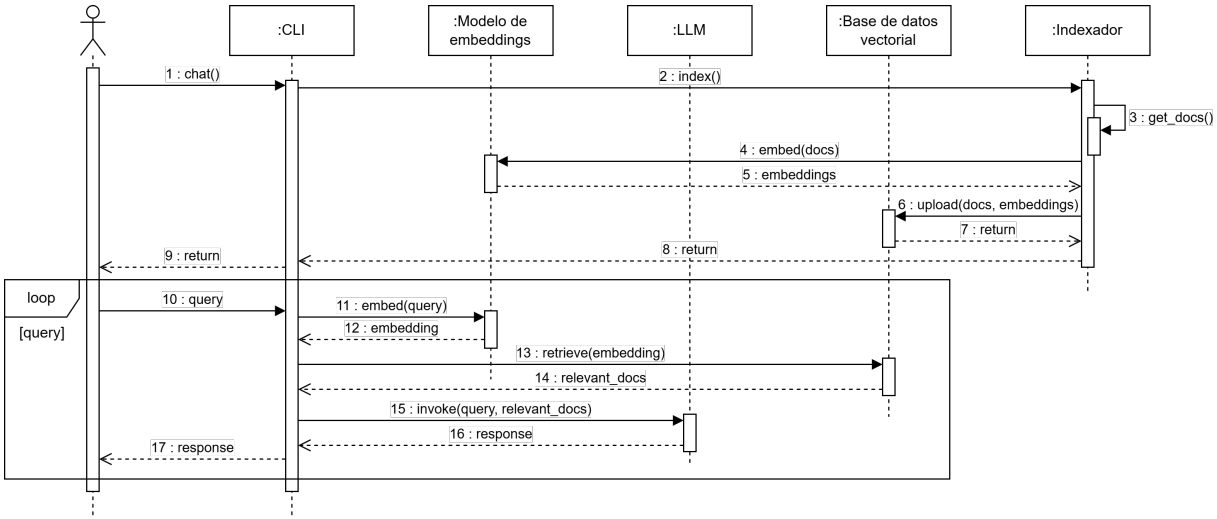

Figura 5.1: Diagrama de secuencia de alto nivel del flujo operativo del asistente virtual.

Por su parte, la Figura 5.2 representa el flujo de evaluación automatizada correspondiente al comando `evaluate`. El proceso comienza con el indexado de los documentos fuente (pasos 2–7), y posteriormente se itera sobre cada ejemplo del conjunto de evaluación. Para cada uno, se repite el ciclo completo de operación del asistente: se genera el *embedding* de la consulta (pasos 9–10), se recupera el contexto relevante (pasos 11–12), y se produce una respuesta (pasos 13–14). Finalmente, esta respuesta es comparada con la referencia esperada mediante un módulo de evaluación, que computa métricas automáticas y validaciones basadas en LLM (paso 15).

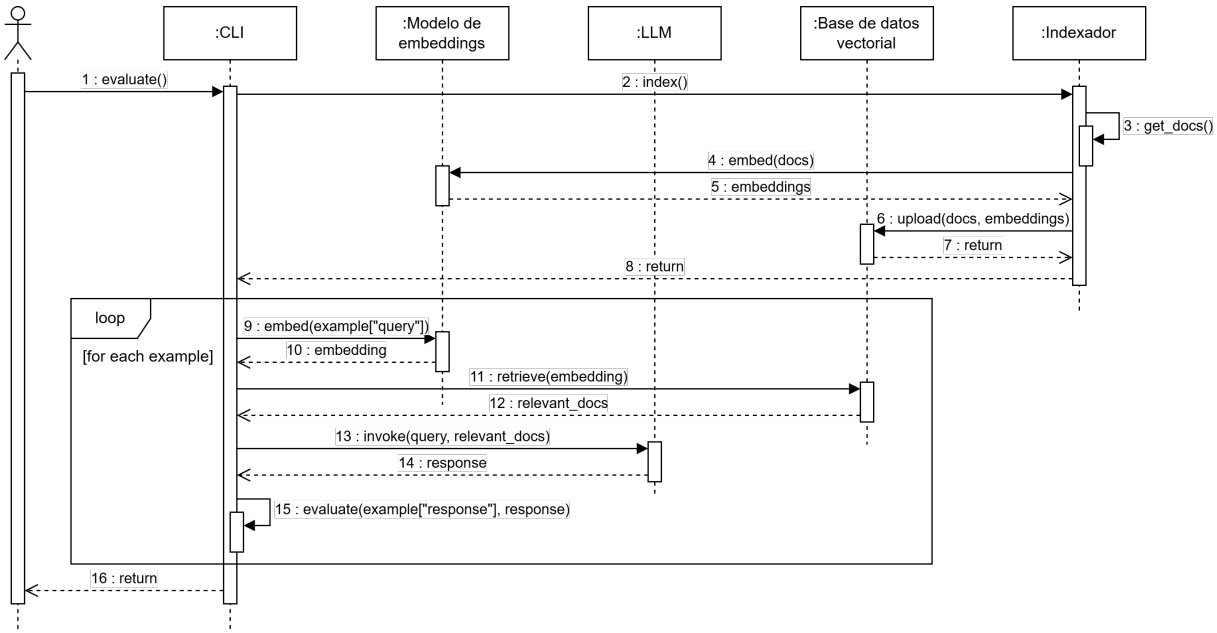

Figura 5.2: Diagrama de secuencia de alto nivel de la evaluación del asistente virtual.

SECCIÓN 5.2

Implementación

La herramienta desarrollada se ha estructurado siguiendo una arquitectura modular y parametrizable, permitiendo su aplicación a distintos contextos sin necesidad de modificar el núcleo del sistema. Su diseño responde a una lógica orientada a proyectos, cada uno de los cuales contiene de forma independiente tanto el conjunto de documentos que constituyen la base de conocimiento como el *prompt* de sistema que condiciona el comportamiento del asistente. La implementación se apoya en el *framework* LangChain, que proporciona los componentes necesarios para construir cadenas de procesamiento orientadas a asistentes virtuales basados en RAG.

En el caso específico considerado, relacionado con la asistencia a usuarios en una aplicación de gestión de flotas, se ha creado una base de conocimiento a partir de dos fuentes heterogéneas proporcionadas por la empresa: un manual técnico en formato PDF y una serie de vídeos tutorizados. El manual fue transformado en texto plano mediante el cargador `PyPDFLoader`, parte del módulo `langchain_community`, y posteriormente sometido a un proceso de depuración para eliminar contenidos confidenciales. Gracias a la buena estructuración formal del documento original, fue posible aplicar una segmentación semánticamente coherente que dio lugar a una colección de fragmentos organizados. Por su parte, las transcripciones de los vídeos se unificaron en un único archivo de texto plano, formando en conjunto con el manual la totalidad de la base de conocimiento utilizada en la experimentación.

Con el fin de permitir una reutilización general de la herramienta, se ha diseñado un flujo de trabajo en el que el usuario únicamente debe proporcionar los ficheros de texto que conforman la base de conocimiento del proyecto, junto con un fichero `system.prompt` que define el rol del asistente e incluye una plantilla de interacción con un marcador específico para insertar el contexto recuperado. El sistema admite múltiples proyectos simultáneos, diferenciados mediante su identificador, y se ha concebido para operar sin requerir intervención sobre el código base, lo que favorece su portabilidad.

Desde el punto de vista funcional, el asistente virtual implementado integra un modelo de lenguaje configurado dinámicamente, un módulo de generación de *embeddings* y una base de datos vectorial para recuperación de contexto. La base de conocimiento es segmentada mediante un procedimiento jerárquico con solapamiento configurable y posteriormente indexada utilizando el sistema `Chroma`. A partir de esta representación vectorial, se construye un recuperador que opera según el tipo de búsqueda definido (por ejemplo, por similitud o MMR), proporcionando al modelo de lenguaje un subconjunto de documentos relevantes para cada entrada del usuario.

Una característica relevante del asistente es la incorporación de memoria a corto plazo, representada por una lista acotada de mensajes que conforman el historial reciente de la conversación. Esta memoria se integra en la plantilla de mensajes del *prompt* a través del mecanismo `MessagesPlaceholder`, permitiendo que el asistente mantenga coherencia conversacional sin incurrir en un coste excesivo.

En cuanto al proceso de evaluación, se ha desarrollado un entorno de pruebas inspirado en el marco metodológico *Retrieval Augmented Generation Assessment* (RAGAS), aunque con modificaciones significativas. En particular, se ha adaptado el sistema para su integración con LangSmith,

una herramienta que permite registrar y analizar interacciones de forma estructurada. A diferencia del enfoque original de RAGAS, que conlleva un coste elevado debido a la complejidad de sus métricas y al uso intensivo de modelos de lenguaje como jueces (*LLM-as-a-judge*), el presente sistema emplea un conjunto reducido y menos sofisticado de evaluaciones automatizadas, lo que permite obtener métricas útiles para la comparación de configuraciones a un coste computacional y económico considerablemente menor.

La implementación está organizada en una jerarquía modular de métricas que abarca tres categorías principales: métricas tradicionales basadas en métodos estadísticos clásicos, métricas basadas en *embeddings* para evaluar la similitud semántica, y métricas que emplean modelos de lenguaje para análisis cualitativos avanzados. Entre estas últimas destacan indicadores como la relevancia y fidelidad de las respuestas y la relevancia contextual, que serán introducidos en el próximo capítulo. Esta estructura jerárquica y modular facilita la incorporación sencilla de nuevas métricas, permitiendo su integración directa en el proceso evaluativo sin necesidad de modificar sustancialmente la arquitectura existente. De este modo, se favorece la extensibilidad y adaptabilidad del sistema a distintos dominios y objetivos de evaluación.

Este enfoque permite evaluar de forma ágil la calidad de las respuestas generadas, atendiendo tanto a su adecuación respecto al contexto como a su coherencia conversacional, sin depender de la intervención humana directa. La arquitectura resultante posibilita, por tanto, el despliegue de asistentes virtuales especializados y su evaluación sistemática en diferentes dominios de aplicación, con una carga operativa moderada.

La documentación completa de la herramienta está disponible en línea[1], mientras que el código fuente se encuentra alojado en un repositorio público de GitHub[2]. Por motivos de protección de datos, la base de conocimiento original no se incluye en el repositorio; en su lugar, se ha añadido una base de conocimiento de ejemplo junto con un *prompt* de sistema ilustrativo, localizados en el directorio `data/`. Para facilitar la reproducibilidad y la instalación, se ha configurado un entorno gestionado mediante Poetry[3], permitiendo una instalación sencilla y controlada de todas las dependencias necesarias para ejecutar la herramienta.

Además, el lector interesado puede encontrar en el Apéndice A una descripción detallada de una versión simplificada de la implementación del asistente virtual aquí propuesto. Aunque reducida en complejidad, dicha versión ilustra con claridad tanto el esquema de parametrización como el flujo operativo del sistema.

[1] https://rubenmartinez795.github.io/ragbot
[2] https://github.com/rubenmartinez795/ragbot
[3] https://python-poetry.org/

CAPÍTULO 6

Evaluación y resultados

"The only way to do great work is to love what you do."

Steve Jobs

En este capítulo se expone la metodología empleada para la evaluación del asistente virtual, se describen las métricas utilizadas y, posteriormente, se presentan los resultados obtenidos en términos del desempeño del sistema ante preguntas relativas al uso de la aplicación de gestión de flotas.

SECCIÓN 6.1

Metodología

La evaluación del asistente se ha estructurado en torno a dos aspectos fundamentales e interdependientes: la calidad de recuperación y la calidad de generación. Ambos componentes están profundamente entrelazados en el marco de los sistemas RAG, dado que el éxito en la generación de respuestas relevantes y coherentes depende en gran medida de la pertinencia del contexto recuperado en relación con la consulta inicial del usuario.

En lo que respecta a la calidad de recuperación, se han considerado diversos parámetros del sistema que afectan a esta dimensión, como el modelo de *embeddings* utilizado, el tamaño de los fragmentos en que se dividen los documentos, el solapamiento entre dichos fragmentos, el tipo de búsqueda empleado, y el número de documentos seleccionados como contexto. A lo largo de este trabajo se emplean los términos «documento» y «fragmento» de forma intercambiable según el nivel de granularidad analizado, aunque se procura mantener una distinción terminológica adecuada conforme a la literatura especializada.

Por otro lado, la calidad de generación se ve influida por una serie de parámetros relativos al modelo de lenguaje utilizado. Entre estos se encuentran la elección del LLM, la temperatura del modelo y los valores de muestreo, tales como top-p y top-k. A estos se suma un elemento crítico para guiar el comportamiento del modelo: el *prompt* de sistema. Este último cumple una función esencial, ya que actúa como contexto instructivo que define el estilo, la estructura y el enfoque con que el modelo debe interpretar y responder las consultas del usuario. La correcta ingeniería de *prompts* resulta, por tanto, fundamental para obtener respuestas útiles, informativas y alineadas con los objetivos del asistente.

El análisis conjunto de recuperación y generación es necesario para comprender en profundidad el rendimiento del sistema. Una recuperación deficiente puede arrastrar al modelo a producir respuestas irrelevantes o incluso incorrectas, por muy competente que sea el generador. En otras palabras, si

los fragmentos recuperados no contienen información relacionada con la pregunta, el modelo de lenguaje no podrá generar una respuesta satisfactoria. Esta interdependencia justifica la adopción de un enfoque de evaluación integral, que contemple simultáneamente la relación entre la consulta, el contexto recuperado y la respuesta producida.

6.1.1. Calidad de recuperación

Para evaluar la calidad de recuperación se analiza el grado de pertinencia de los fragmentos seleccionados como contexto respecto a la consulta formulada. En un escenario ideal, el sistema debería recuperar exclusivamente aquellos fragmentos que contienen información relevante para responder a la pregunta del usuario. Tradicionalmente, esta evaluación puede llevarse a cabo de manera manual, mediante anotadores expertos que juzgan la relevancia de cada fragmento, o bien de forma automática mediante métricas clásicas de recuperación de información. Una de las métricas utilizadas en este área es el *context recall*, que se define como la proporción entre el número de fragmentos relevantes recuperados y el total de fragmentos relevantes disponibles en la base de conocimiento.

Sin embargo, aunque el *context recall* ofrece una medida informativa, presenta ciertas limitaciones. Su valor tiende a aumentar cuando se incrementa el número de fragmentos recuperados, lo cual puede inducir al sistema a incluir información redundante o irrelevante en el contexto. Además, su cálculo requiere una anotación manual extensa de cuáles son los fragmentos relevantes para cada pregunta y para cada configuración posible de segmentación y solapamiento. Esta tarea se vuelve impracticable en sistemas que manejan una base de conocimiento amplia y un conjunto elevado de consultas y configuraciones.

Por ello, en este trabajo se ha adoptado una alternativa más escalable, conocida como relevancia contextual. Esta métrica forma parte de la tríada de evaluación de sistemas RAG, un marco conceptual que vincula tres elementos fundamentales: la consulta, el contexto y la respuesta. La relevancia contextual se centra específicamente en la relación entre la consulta y el contexto recuperado, evaluando cuán bien se ajusta el contenido de los fragmentos al objetivo de la pregunta. Esta evaluación se realiza mediante la estrategia *LLM-as-a-judge*, en la que un modelo de lenguaje, debidamente instruido mediante *prompting*, actúa como evaluador autónomo de la adecuación del contexto.

6.1.2. Calidad de generación

La calidad de la respuesta generada se evalúa a partir de un conjunto de ejemplos que consisten en consultas junto con sus respectivas respuestas de referencia. Esta evaluación busca determinar en qué medida la salida generada por el modelo satisface las expectativas en términos de pertinencia, exhaustividad, claridad y fidelidad al contexto.

Existen varios aspectos que resultan clave para esta evaluación. Uno de ellos es la capacidad del modelo para ignorar el ruido, es decir, su habilidad para discriminar entre información relevante e irrelevante dentro del contexto proporcionado, centrándose únicamente en aquello que resulta útil para la respuesta. Otro aspecto esencial es la capacidad del modelo para abstenerse de responder cuando no posee suficiente información. Esta cualidad, análoga a la prudencia en el razonamiento humano, es fundamental en entornos reales para evitar que el sistema genere afirmaciones infundadas o directamente incorrectas.

Asimismo, se valora la capacidad de síntesis, entendida como la habilidad del modelo para integrar y condensar información dispersa en el contexto de manera coherente y estructurada en una sola respuesta. Por último, también se examina si el modelo es capaz de detectar posibles errores o inconsistencias presentes en los fragmentos recuperados, evitando que dichos defectos se reflejen en la respuesta.

La evaluación se lleva a cabo mediante métricas automáticas como *Bilingual Evaluation Understudy* (BLEU), *Recall-Oriented Understudy for Gisting Evaluation* (ROUGE) y la similitud semántica entre la respuesta generada y la de referencia. Además, se emplean métricas cualitativas fundamentadas en la estrategia *LLM-as-a-judge*, que permiten evaluar la relevancia de la respuesta respecto a la pregunta y su fidelidad respecto al contexto, completando así la tríada evaluativa del sistema.

SECCIÓN 6.2

Métricas

Con el objetivo de proporcionar una evaluación exhaustiva y rigurosa del asistente virtual, se han empleado tanto métricas cuantitativas como cualitativas. Estas métricas han sido integradas en el *framework* de evaluación desarrollado específicamente para este trabajo, parte de `ragbot`, el cual permite aplicar evaluaciones sistemáticas a distintas configuraciones del sistema RAG.

6.2.1. Métricas cuantitativas

Las métricas cuantitativas empleadas en este estudio facilitan la comparación objetiva entre las respuestas generadas por el asistente virtual y las respuestas de referencia elaboradas por expertos. En esta evaluación, se ha implementado un conjunto complementario de métricas que capturan diferentes dimensiones de la calidad textual.

El índice BLEU constituye una métrica fundamental en el análisis. Originalmente desarrollada para evaluar sistemas de traducción automática, BLEU cuantifica la similitud léxica entre la respuesta generada y la referencia a través del cálculo de coincidencias de n-gramas. Su formulación matemática incorpora un factor de penalización por brevedad que previene la asignación de puntuaciones elevadas a respuestas excesivamente cortas. Adicionalmente, BLEU computa un promedio geométrico ponderado de precisiones para diversos tamaños de n-gramas (habitualmente de 1 a 4), lo que permite evaluar tanto la exactitud léxica como la integridad estructural de las frases generadas.

Como métrica complementaria, se implementa ROUGE-L, una variante de ROUGE que se fundamenta en la identificación de la subsecuencia común más larga entre la respuesta generada y la referencia, sin requerir que los elementos sean contiguos. ROUGE-L presenta la ventaja significativa de evaluar la capacidad del modelo para preservar el orden relativo de las palabras y capturar patrones estructurales en las respuestas, incluso cuando existen variaciones léxicas o sintácticas menores. En nuestra evaluación, hemos adoptado específicamente el valor F1 de ROUGE-L, que armoniza la precisión (exactitud de los elementos generados) y la cobertura (exhaustividad de los elementos de referencia capturados) en una única puntuación equilibrada, proporcionando así una valoración integral de la calidad estructural de las respuestas.

Adicionalmente, para transcender las limitaciones inherentes a las comparaciones puramente léxicas, hemos incorporado una medida de similitud semántica (SS) basada en el cálculo de la similitud del coseno entre los vectores de *embeddings* correspondientes a la respuesta generada y la referencia. Como ya hemos visto, estos *embeddings*, derivados de modelos preentrenados de lenguaje, codifican las representaciones semánticas de los textos en espacios vectoriales de alta dimensionalidad, capturando relaciones conceptuales que pueden no manifestarse en coincidencias léxicas directas. Esta métrica resulta especialmente valiosa para evaluar la preservación del significado semántico en casos donde las respuestas presentan reformulaciones o utilizan vocabulario alternativo pero semánticamente equivalente. La evidencia empírica sugiere que esta medida correlaciona significativamente con juicios humanos de equivalencia semántica, complementando así las métricas basadas en *n*-gramas para ofrecer una evaluación multidimensional de la calidad de las respuestas generadas.

6.2.2. Métricas cualitativas

Complementando a las métricas cuantitativas, se han empleado métricas cualitativas que capturan aspectos más subjetivos y contextuales de la generación. Estas métricas, que forman parte de la tríada de evaluación de sistemas RAG, se basan en la estrategia *LLM-as-a-judge*, en la cual se utiliza un modelo de lenguaje debidamente condicionado para que actúe como evaluador.

La primera de estas métricas es la relevancia del contexto, que mide en qué grado los fragmentos recuperados son pertinentes respecto a la consulta. Esta evaluación se realiza en una escala de uno a cinco, donde un valor bajo indica una falta de alineación entre el contenido recuperado y la pregunta, mientras que un valor alto indica una recuperación altamente relevante.

La segunda métrica es la relevancia de la respuesta, que examina en qué medida la salida generada aborda adecuadamente todos los aspectos planteados en la pregunta. Aquí también se utiliza una escala de cinco niveles, y se busca evaluar la capacidad del modelo para extraer del contexto únicamente la información pertinente, sin omisiones ni desviaciones.

Finalmente, se incluye la fidelidad de la respuesta, una métrica crucial que valora la correspondencia entre la respuesta generada y el contenido del contexto. Esta métrica tiene como objetivo detectar posibles alucinaciones del modelo, asegurando que las respuestas estén completamente respaldadas por los fragmentos recuperados. Al igual que las anteriores, se califica en una escala de uno a cinco, siendo cinco una fidelidad total y uno una respuesta no sustentada en absoluto.

SECCIÓN 6.3

Resultados

Con el objetivo de evaluar la eficacia del asistente virtual desarrollado en el presente trabajo, se llevó a cabo una colaboración con la empresa responsable de la plataforma, mediante la cual se elaboró un *dataset* compuesto por 27 preguntas representativas. Estas preguntas fueron seleccionadas por su frecuencia en las consultas reales que los usuarios suelen dirigir al personal de soporte técnico, y respondidas con precisión por expertos en el dominio de la aplicación. De este modo, se obtuvo un conjunto de 27 pares pregunta-respuesta, que constituyen una base fiable para la evaluación del

sistema. El proceso de evaluación se dividió en dos fases principales: la evaluación automática y la evaluación manual.

6.3.1. Evaluación automática

En primer lugar, se realizó una evaluación automática de la calidad de las respuestas generadas por el asistente bajo distintas configuraciones de parámetros. Esta evaluación se centró en medir la métricas presentadas en la sección anterior, tales como BLEU, ROUGE, similitud semántica (SS), relevancia contextual (RC), relevancia de la respuesta (RR) y fidelidad de la respuesta (FR). Las configuraciones evaluadas difieren en aspectos como la temperatura del modelo generativo (t), los parámetros de muestreo top-p (top_p) y top-k (top_k), el tamaño de los fragmentos de contexto recuperados (tf) y su solapamiento (sf), el tipo de búsqueda utilizado (tb), y el número de documentos aportados al modelo como contexto (k). Los resultados obtenidos se resumen en la Tabla 6.1.

Tabla 6.1: Evaluación del asistente virtual con diferentes configuraciones de sus parámetros. Se abrevian las siguientes métricas: similitud semántica (SS), relevancia contextual (RC), relevancia de la respuesta (RR) y fidelidad de la respuesta (FR). Las casillas en blanco indican que no se altera el valor por defecto del parámetro correspondiente, es decir, el de la configuración base.

	t	top_p	top_k	tf	sf	tb	k	BLEU	ROUGE	SS	RC	RR	FR
base	0.0	0.85	40	1000	200	sim	4	21.3	37.3	90.6	3.85	4.44	4.48
		0.0	1					21.3	37.4	90.5	3.85	4.48	4.52
	0.3	0.3	30					21.4	37.3	90.6	3.85	4.44	4.41
(A)	0.6	0.6	60					23.6	38.3	90.4	3.85	4.59	4.59
	0.9	0.9	90					22.2	38.1	90.7	3.85	4.52	4.52
	1.0	1.0	100					20.0	35.7	90.2	3.85	4.33	4.41
				100	0			16.7	27.0	87.9	3.30	3.30	3.59
				500	100			19.1	31.7	89.1	3.81	4.26	4.11
(B)				2000	400			23.2	38.4	90.6	3.78	4.56	4.11
				3000	600			21.9	39.2	90.5	3.70	4.74	4.41
				4000	800			24.2	38.6	89.8	3.93	4.33	3.85
(C)						MMR		22.5	34.6	89.0	3.41	4.19	4.07
							1	18.7	30.2	87.0	3.30	3.81	4.30
							2	23.5	36.8	90.2	3.63	4.33	4.48
(D)							3	21.4	35.6	90.1	3.93	4.22	4.52
							5	23.6	38.2	90.7	3.81	4.74	4.19
							6	22.7	36.6	90.0	3.67	4.59	4.33
(E)	sin utilizar RAG							12.5	11.9	82.5	1.00	4.81	1.00

El análisis de las configuraciones del bloque (A) revela un comportamiento no monótono respecto a la temperatura del modelo, con un punto óptimo en $t = 0.6$ que alcanza los valores máximos de BLEU (23.6) y un ROUGE elevado (38.3). Se observa una clara degradación del rendimiento en los extremos: con generación determinista ($t = 0.0$) y con máxima aleatoriedad ($t = 1.0$), donde se producen descensos significativos en las métricas automáticas. La combinación de valores moderados ($t = 0.6$, $top_p = 0.6$, $top_k = 60$) produce los mejores resultados, demostrando que un punto intermedio en los parámetros de muestreo permite al modelo generar respuestas que combinan precisión factual con

fluidez expresiva. Es destacable que la similitud semántica se mantiene estable (en torno al 90 %) en todas estas configuraciones, sugiriendo que estos parámetros afectan principalmente a la estructura superficial de las respuestas pero preservan su contenido semántico esencial. Además, cabe notar que los resultados de relevancia contextual se mantienen constantes, puesto que estos parámetros no influyen en la calidad de recuperación del sistema.

En el bloque (B), se evidencia un incremento sustancial en el rendimiento al aumentar el tamaño de fragmento desde 100 caracteres (BLEU=16.7, ROUGE=27.0) hasta 2000 caracteres (BLEU=23.2, ROUGE=38.4), identificándose un umbral crítico de información necesaria. La relación entre tamaño de fragmento y calidad no es lineal: para fragmentos de 4000 caracteres, aunque BLEU alcanza su máximo (24.2), la fidelidad de respuesta disminuye (FR=3.85 vs 4.11 con fragmentos de 2000 caracteres). Se observa un fenómeno de saturación informativa en torno a los 3000 caracteres, donde se alcanza el máximo de relevancia de respuesta (RR=4.74) pero con menor fidelidad que configuraciones con fragmentos más pequeños.

La comparación entre búsqueda por similitud estándar y el algoritmo MMR (configuración C) muestra valores inferiores en todas las métricas con MMR (ROUGE desciende de 37.3 a 34.6, y RC de 3.85 a 3.41). Esto evidencia un *trade-off* entre diversidad de información recuperada y precisión de las respuestas. La métrica más afectada es la relevancia contextual, con una reducción del 8.8 %, sugiriendo que MMR recupera fragmentos menos alineados con la consulta específica. Los resultados indican que para dominios especializados como el evaluado, la precisión en la recuperación prevalece sobre la diversidad.

El análisis del bloque (D) muestra que con un solo documento ($k = 1$), todas las métricas son significativamente inferiores a la configuración base. El rendimiento no mejora monótonamente con el aumento de documentos, observándose un patrón oscilante con picos en $k = 2$ y $k = 5$. Diferentes métricas alcanzan sus valores máximos con distintos valores de k: RC con $k = 3$ (3.93), RR con $k = 5$ (4.74), y FR con $k = 3$ (4.52), sugiriendo que el número óptimo depende del aspecto de calidad priorizado. A partir de $k = 5$, se observa una degradación en múltiples métricas, indicando un fenómeno de saturación.

Por último, la configuración (E), sin mecanismo RAG, muestra una caída drástica en BLEU y ROUGE (12.5 y 11.9 respectivamente), evidenciando la dependencia crítica del modelo en el contexto recuperado. La relevancia contextual colapsa al valor mínimo (1.00), mientras que, paradójicamente, la relevancia de respuesta alcanza su máximo (RR=4.81). Esta dualidad sugiere que el mecanismo RAG es responsable de la veracidad factual, mientras que la relevancia percibida puede lograrse incluso con generación no fundamentada.

Del análisis realizado se infiere que una configuración óptima incluiría: temperatura moderada ($t = 0.6$), fragmentos de tamaño considerable (2000–3000 caracteres), búsqueda por similitud estándar y entre 3 y 5 documentos de contexto. Los resultados revelan múltiples *trade-offs* inherentes al diseño de sistemas RAG: entre diversidad y precisión de recuperación, cantidad de contexto y capacidad de integración, o temperatura de generación y fidelidad factual. Se evidencia también una dualidad fundamental entre veracidad factual (dependiente del mecanismo RAG) y relevancia percibida (alcanzable incluso sin RAG).

6.3.2. Evaluación manual

Además de la evaluación automática, se llevó a cabo una evaluación manual del conjunto de 27 ejemplos, realizada por expertos humanos, con el objetivo de verificar la calidad del asistente y analizar la correspondencia entre las métricas automáticas y la percepción humana. Se utilizó la plataforma LangSmith para gestionar las tareas de anotación, con una guía precisa de evaluación. Las respuestas fueron juzgadas según los tres criterios recogidos en el baremo del apéndice B: capacidad del sistema para dar una respuesta, presencia de alucinaciones, y calidad general de la misma. Los resultados obtenidos para la configuración base se presentan en la Tabla 6.2.

Tabla 6.2: Resultados de la evaluación manual del asistente virtual mediante colas de anotación para la configuración base.

	¿Sabe responder?		¿Alucinaciones?		Calidad			
	Sí	No	Sí	No	Excelente	Buena	Normal	Mala
base	81 %	19 %	0 %	100 %	22.2 %	22.2 %	22.2 %	33.4 %

Estos resultados muestran un desempeño generalmente positivo. La ausencia total de alucinaciones pone de manifiesto la efectividad del diseño del *prompt* de sistema, que evita la invención de información ajena al contexto recuperado. El hecho de que el 19 % de las preguntas no obtuvieran respuesta se atribuye a la falta de información relevante en el manual de usuario o en las transcripciones de los vídeos indexados, más que a errores de recuperación o generación.

Al considerar únicamente las respuestas en las que el modelo afirma saber responder, los resultados mejoran notablemente, como se observa en la Tabla 6.3. En este caso, todas las respuestas son válidas, libres de alucinaciones, y con un incremento considerable en la proporción de respuestas excelentes.

Tabla 6.3: Resultados de la evaluación manual del asistente virtual mediante colas de anotación para la configuración base, excluyendo las preguntas a las que el modelo admite no saber la respuesta.

	¿Sabe responder?		¿Alucinaciones?		Calidad			
	Sí	No	Sí	No	Excelente	Buena	Normal	Mala
base	100 %	0 %	0 %	100 %	31.8 %	22.7 %	22.7 %	22.8 %

En conclusión, los resultados de esta sección evidencian que el asistente virtual desarrollado ofrece un rendimiento robusto, con respuestas de alta calidad cuando se dispone de contexto relevante, y con un manejo adecuado de la ignorancia cuando no es posible proporcionar una respuesta fiable. El sistema logra evitar alucinaciones y mantiene una coherencia semántica elevada, lo que lo convierte en una herramienta viable y eficaz para asistir a usuarios en entornos técnicos complejos.

CAPÍTULO 7

Conclusiones y trabajo futuro

"Caminante, son tus huellas
el camino y nada más;
Caminante, no hay camino,
se hace camino al andar".

Antonio Machado

El presente trabajo ha abordado el diseño, implementación y evaluación de un asistente virtual basado en técnicas de RAG aplicado al contexto específico de una aplicación de gestión de flotas de vehículos. Además, se ha realizado un recorrido que abarca desde los fundamentos teóricos de los modelos de lenguaje hasta la implementación práctica de un sistema completo, pasando por la definición de métricas y metodologías de evaluación. Esto ha permitido extraer valiosas conclusiones que se exponen a continuación, junto con las posibles líneas de trabajo futuro que emergen de esta investigación.

SECCIÓN 7.1

Conclusiones

Con el desarrollo de este trabajo se ha demostrado que los sistemas RAG constituyen una aproximación efectiva para dotar a los asistentes virtuales de capacidades de respuesta precisa en dominios específicos. La integración de un mecanismo de recuperación contextual con un modelo generativo permite superar las limitaciones de los modelos preentrenados, particularmente en lo que respecta a la actualización del conocimiento y la fiabilidad de las respuestas.

El análisis de los componentes y parámetros de un sistema RAG evidencia múltiples *trade-offs* que requieren consideración durante su diseño. La calidad de recuperación impacta directamente en la precisión de las respuestas, estableciendo una dependencia crítica entre ambos procesos. En este sentido, el uso de RAG ha demostrado una mejora significativa respecto a configuraciones sin recuperación: se han obtenido incrementos de 8.8 puntos en la métrica BLEU y de 25.4 puntos en ROUGE frente a un modelo GPT sin acceso a fuentes externas, lo que subraya el valor añadido de esta arquitectura en tareas de generación específica. No obstante, incluso sin mecanismos de recuperación, un modelo bien parametrizado puede ofrecer respuestas percibidas como relevantes, lo que reafirma la necesidad de evaluar no solo la coherencia, sino también la fidelidad factual de las respuestas generadas.

La exploración de distintas configuraciones ha revelado patrones útiles para el diseño de este tipo de asistentes: un valor de temperatura próximo a 0.6 ofrece un buen equilibrio entre precisión y naturalidad del lenguaje; fragmentos textuales entre 2000 y 3000 caracteres maximizan la cobertura

sin comprometer la pertinencia; y, en dominios especializados, la búsqueda por similitud tradicional resulta más eficaz que algoritmos como MMR. Asimismo, la capacidad del sistema para abstenerse de responder cuando no dispone de información suficiente ha demostrado ser clave para mantener una percepción positiva sobre su calidad general.

La metodología de evaluación empleada, que combina métricas cuantitativas con valoraciones cualitativas basadas en la estrategia *LLM-as-a-judge*, ha resultado efectiva para captar las múltiples dimensiones de la calidad en este tipo de sistemas. La implementación modular del asistente, cristalizada en la herramienta `ragbot`, ha facilitado la experimentación sistemática y deja sentadas las bases para su extensión y adaptación futura.

Por ende, este trabajo constituye una contribución relevante tanto al conocimiento teórico sobre sistemas RAG como a su aplicación práctica en entornos empresariales, confirmando la viabilidad de desarrollar asistentes virtuales especializados capaces de proporcionar asistencia fiable, coherente y contextualmente informada.

SECCIÓN 7.2

Trabajo futuro

A partir de la investigación llevada a cabo, se identifican tres líneas principales de trabajo futuro: el perfeccionamiento de las técnicas RAG, la evolución del asistente hacia una arquitectura de agente autónomo y la ampliación de las capacidades de evaluación de la herramienta `ragbot`. En relación con las técnicas RAG, resulta prometedor explorar métodos como la expansión y reescritura de consultas, orientados a enriquecer la representación semántica y reducir las discrepancias léxicas entre pregunta y contenido. También se considera una vía alternativa basada en *fine-tuning* de modelos generativos sobre tareas específicas, lo cual permitiría una integración más directa entre comprensión y generación, aunque conlleva un mayor esfuerzo en la curación de datos y el entrenamiento.

En cuanto a la evolución arquitectónica, una dirección especialmente interesante consiste en dotar al asistente de características propias de un agente inteligente. Este podría conectarse directamente con la base de datos del sistema de gestión de flotas, permitiendo ofrecer respuestas actualizadas sobre el estado de vehículos, conductores o métricas operativas. A su vez, la incorporación de razonamiento simbólico y planificación permitiría al sistema descomponer tareas complejas en subtareas más simples, orquestando su resolución eficiente. En paralelo, una integración completa del asistente en la plataforma de gestión plantea retos técnicos vinculados con su personalización y la implementación de mecanismos de retroalimentación continua para el aprendizaje a partir de la experiencia del usuario.

Por último, en lo que respecta a la herramienta `ragbot`, se prevé su evolución mediante la incorporación de nuevas métricas de evaluación que contemplen no solo la calidad de la respuesta, sino también factores como la eficiencia computacional y la coherencia en interacciones prolongadas. Asimismo, ampliar el soporte a modelos alternativos facilitaría comparativas más completas y permitiría adaptar la herramienta a distintos escenarios de uso.

Bibliografía

[1] P. Lewis et al., *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, 2021. arXiv: 2005.11401 [cs.CL]. dirección: https://arxiv.org/abs/2005.11401 (vid. págs. 1, 7).

[2] A. A. Markov, «Essai d'une Recherche Statistique sur le Texte du Roman "Eugene Onegin" Illustrant la Liaison des Epreuve en Chain,» *Izvistia Imperatorskoi Akademii Nauk*, 1913 (vid. pág. 4).

[3] C. Shannon y W. Weaver, *The Mathemathical Theory of Communication*. University of Illinois Press, 1949 (vid. pág. 4).

[4] A. M. Turing, «Computing Machinery and Intelligence,» en *Parsing the Turing Test: Philosophical and Methodological Issues in the Quest for the Thinking Computer*, R. Epstein, G. Roberts y G. Beber, eds. Dordrecht: Springer Netherlands, 2009, págs. 23-65, ISBN: 978-1-4020-6710-5. DOI: 10.1007/978-1-4020-6710-5_3. dirección: https://doi.org/10.1007/978-1-4020-6710-5_3 (vid. pág. 4).

[5] F. Jelinek, «Continuous Speech Recognition by Statistical Methods,» *Proceedings of the IEEE*, vol. 64, n.º 4, págs. 532-556, 1976. DOI: 10.1109/PROC.1976.10159 (vid. pág. 5).

[6] J. Weizenbaum, «Computer Power and Human Reason,» *W. H. Freeman*, 1976 (vid. págs. 5, 27).

[7] I. J. Good, «The Population Frequencies of Species and the Estimation of Population Parameters,» *Biometrika*, vol. 40, n.º 3-4, págs. 237-264, dic. de 1953, ISSN: 0006-3444. DOI: 10.1093/biomet/40.3-4.237. eprint: https://academic.oup.com/biomet/article-pdf/40/3-4/237/492571/40-3-4-237.pdf. dirección: https://doi.org/10.1093/biomet/40.3-4.237 (vid. pág. 5).

[8] I. Witten y T. Bell, «The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression,» *IEEE Transactions on Information Theory*, vol. 37, n.º 4, págs. 1085-1094, 1991. DOI: 10.1109/18.87000 (vid. pág. 5).

[9] S. F. Chen y J. T. Goodman, *An Empirical Study of Smoothing Techniques for Language Modeling*, 1996. arXiv: cmp-lg/9606011 [cmp-lg]. dirección: https://arxiv.org/abs/cmp-lg/9606011 (vid. pág. 5).

[10] N. Morgan y H. Bourlard, «Continuous Speech Recognition Using Multilayer Perceptrons With Hidden Markov Models,» en *International Conference on Acoustics, Speech, and Signal Processing*, 1990, 413-416 vol.1. DOI: 10.1109/ICASSP.1990.115720 (vid. pág. 5).

[11] S. Hochreiter y J. Schmidhuber, «Long Short-Term Memory,» *Neural Computation*, vol. 9, n.º 8, págs. 1735-1780, nov. de 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. dirección: https://doi.org/10.1162/neco.1997.9.8.1735 (vid. pág. 5).

[12] Y. Bengio, P. Lamblin, D. Popovici y H. Larochelle, «Greedy Layer-Wise Training of Deep Networks,» en *Advances in Neural Information Processing Systems*, B. Schölkopf, J. Platt y T. Hoffman, eds., vol. 19, MIT Press, 2006. dirección: https://proceedings.neurips.cc/paper_files/paper/2006/file/5da713a690c067105aeb2fae32403405-Paper.pdf (vid. pág. 5).

[13] R. Collobert y J. Weston, «A Unified Architecture for Natural Language Processing: Deep Neural Networks With Multitask Learning,» en *Proceedings of the 25th International Conference on Machine Learning*, ép. ICML '08, Helsinki, Finland: Association for Computing Machinery, 2008, págs. 160-167, ISBN: 9781605582054. DOI: 10.1145/1390156.1390177. dirección: https://doi.org/10.1145/1390156.1390177 (vid. pág. 5).

[14] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu y P. Kuksa, *Natural Language Processing (almost) from Scratch*, 2011. arXiv: 1103.0398 [cs.LG]. dirección: https://arxiv.org/abs/1103.0398 (vid. pág. 5).

[15] T. Mikolov, K. Chen, G. Corrado y J. Dean, *Efficient Estimation of Word Representations in Vector Space*, 2013. arXiv: 1301.3781 [cs.CL]. dirección: https://arxiv.org/abs/1301.3781 (vid. págs. 6, 25).

[16] J. Pennington, R. Socher y C. Manning, *Glove: Global Vectors for Word Representation*, ene. de 2014. DOI: 10.3115/v1/D14-1162 (vid. pág. 6).

[17] D. Bahdanau, K. Cho e Y. Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*, 2016. arXiv: 1409.0473 [cs.CL]. dirección: https://arxiv.org/abs/1409.0473 (vid. pág. 6).

[18] A. Vaswani et al., «Attention Is All You Need,» en *Advances in Neural Information Processing Systems*, I. Guyon et al., eds., vol. 30, Curran Associates, Inc., 2017. dirección: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf (vid. págs. 6, 15, 18).

[19] J. Devlin, M.-W. Chang, K. Lee y K. Toutanova, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019. arXiv: 1810.04805 [cs.CL]. dirección: https://arxiv.org/abs/1810.04805 (vid. págs. 6, 12).

[20] A. Radford, K. Narasimhan, T. Salimans e I. Sutskever, «Improving Language Understanding by Generative Pre-Training,» *OpenAI Blog*, 2018. dirección: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf (vid. pág. 6).

[21] Y. Zhu et al., *Aligning Books and Movies: Towards Story-like Visual Explanations by Watching Movies and Reading Books*, 2015. arXiv: 1506.06724 [cs.CV]. dirección: https://arxiv.org/abs/1506.06724 (vid. pág. 6).

[22] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei e I. Sutskever, «Language Models Are Unsupervised Multitask Learners,» *OpenAI Blog*, 2019. dirección: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf (vid. pág. 6).

[23] T. B. Brown et al., *Language Models Are Few-Shot Learners*, 2020. arXiv: 2005.14165 [cs.CL]. dirección: https://arxiv.org/abs/2005.14165 (vid. pág. 6).

[24] C. Raffel et al., *Exploring the Limits of Transfer Learning With a Unified Text-to-Text Transformer*, 2023. arXiv: 1910.10683 [cs.LG]. dirección: https://arxiv.org/abs/1910.10683 (vid. págs. 6, 12).

[25] B. Workshop et al., *BLOOM: A 176B-Parameter Open-Access Multilingual Language Model*, 2023. arXiv: 2211.05100 [cs.CL]. dirección: https://arxiv.org/abs/2211.05100 (vid. pág. 6).

[26] A. Chowdhery et al., *PaLM: Scaling Language Modeling With Pathways*, 2022. arXiv: 2204.02311 [cs.CL]. dirección: https://arxiv.org/abs/2204.02311 (vid. pág. 6).

[27] OpenAI et al., *GPT-4 Technical Report*, 2024. arXiv: 2303.08774 [cs.CL]. dirección: https://arxiv.org/abs/2303.08774 (vid. pág. 7).

[28] OpenAI et al., *OpenAI o1 System Card*, 2024. arXiv: 2412.16720 [cs.AI]. dirección: https://arxiv.org/abs/2412.16720 (vid. pág. 7).

[29] DeepSeek-AI et al., *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*, 2025. arXiv: 2501.12948 [cs.CL]. dirección: https://arxiv.org/abs/2501.12948 (vid. pág. 7).

[30] U. Khandelwal, O. Levy, D. Jurafsky, L. Zettlemoyer y M. Lewis, *Generalization Through Memorization: Nearest Neighbor Language Models*, 2020. arXiv: 1911.00172 [cs.CL]. dirección: https://arxiv.org/abs/1911.00172 (vid. pág. 7).

[31] Y. Gao et al., *Retrieval-Augmented Generation for Large Language Models: A Survey*, 2024. arXiv: 2312.10997 [cs.CL]. dirección: https://arxiv.org/abs/2312.10997 (vid. pág. 7).

[32] R. T. de Lima et al., *Know Your RAG: Dataset Taxonomy and Generation Strategies for Evaluating RAG Systems*, 2024. arXiv: 2411.19710 [cs.IR]. dirección: https://arxiv.org/abs/2411.19710 (vid. pág. 7).

[33] N. F. Liu et al., *Lost in the Middle: How Language Models Use Long Contexts*, 2023. arXiv: 2307.03172 [cs.CL]. dirección: https://arxiv.org/abs/2307.03172 (vid. pág. 7).

[34] F. Cuconasu et al., «The Power of Noise: Redefining Retrieval for RAG Systems,» en *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ép. SIGIR 2024, ACM, jul. de 2024, págs. 719-729. DOI: 10.1145/3626772.3657834. dirección: http://dx.doi.org/10.1145/3626772.3657834 (vid. pág. 7).

[35] Y. Liu et al., *RoBERTa: A Robustly Optimized BERT Pretraining Approach*, 2019. arXiv: 1907.11692 [cs.CL]. dirección: https://arxiv.org/abs/1907.11692 (vid. pág. 12).

[36] H. Touvron et al., *LLaMA: Open and Efficient Foundation Language Models*, 2023. arXiv: 2302.13971 [cs.CL]. dirección: https://arxiv.org/abs/2302.13971 (vid. pág. 12).

[37] M. Lewis et al., *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*, 2019. arXiv: 1910.13461 [cs.CL]. dirección: https://arxiv.org/abs/1910.13461 (vid. pág. 12).

[38] OpenAI et al., *GPT-4o System Card*, 2024. arXiv: 2410.21276 [cs.CL]. dirección: https://arxiv.org/abs/2410.21276 (vid. pág. 13).

[39] R. Luo et al., «BioGPT: Generative Pre-Trained Transformer for Biomedical Text Generation and Mining,» *Briefings in Bioinformatics*, vol. 23, n.º 6, sep. de 2022, ISSN: 1477-4054. DOI: 10.1093/bib/bbac409. dirección: http://dx.doi.org/10.1093/bib/bbac409 (vid. pág. 13).

[40] H. Yang, X.-Y. Liu y C. D. Wang, *FinGPT: Open-Source Financial Large Language Models*, 2023. arXiv: 2306.06031 [q-fin.ST]. dirección: https://arxiv.org/abs/2306.06031 (vid. pág. 13).

[41] M. Chen et al., *Evaluating Large Language Models Trained on Code*, 2021. arXiv: 2107.03374 [cs.LG]. dirección: https://arxiv.org/abs/2107.03374 (vid. pág. 13).

[42] C. White et al., «LiveBench: A Challenging, Contamination-Free LLM Benchmark,» en *The Thirteenth International Conference on Learning Representations*, 2025 (vid. pág. 14).

[43] W.-L. Chiang et al., *Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference*, 2024. arXiv: 2403.04132 [cs.AI] (vid. pág. 14).

[44] Y.-A. Wang e Y.-N. Chen, *What Do Position Embeddings Learn? An Empirical Study of Pre-Trained Language Model Positional Encoding*, 2020. arXiv: 2010.04903 [cs.CL]. dirección: https://arxiv.org/abs/2010.04903 (vid. pág. 19).

[45] A. Ruoss et al., *Randomized Positional Encodings Boost Length Generalization of Transformers*, 2023. arXiv: 2305.16843 [cs.LG]. dirección: https://arxiv.org/abs/2305.16843 (vid. pág. 19).

[46] N. Reimers e I. Gurevych, *Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks*, 2019. arXiv: 1908.10084 [cs.CL]. dirección: https://arxiv.org/abs/1908.10084 (vid. pág. 29).

APÉNDICE A

Implementación básica de un *chatbot*

En este capítulo, se presenta una implementación simplificada, pero reproducible de un asistente virtual. Para ello, se irán mostrando extractos del código fuente con la siguiente estructura:

<< example/code.py
```
def func(a: int):
  b = 2
  a *= b
  return a
```

La sección subrayada destaca las nuevas líneas que van siendo introducidas, enmarcadas por fragmentos de código ya existentes para facilitar su ubicación. Adicionalmente, en la esquina superior derecha se especifica el fichero correspondiente en el que se incorporan.

SECCIÓN A.1

Flujo de control

Nuestra versión simplificada del asistente constará de tres ficheros: `cli.py`, `rag.py` y `utils.py`. En un alarde de ingenio, se evita sobrecargar esta sección mostrando los módulos requeridos por cada *script* y otras funciones menos relevantes en la sección A.2. Comencemos presentando la lógica de la herramienta: un bucle de interacciones continuas con el asistente, al que aquí llamamos `rag`, y que el usuario puede finalizar en cualquier momento escribiendo `/bye`.

<< rag/cli.py
```
def chat(rag: Runnable):
  while True:
    query = input(">>> ")
    if query.lower() == "/bye": exit(0)
    response = rag.invoke({"input": query})
    print(response["answer"])
```

Veamos ahora cómo implementar el sistema RAG de la manera más parametrizada posible, facilitando así el proceso de evaluación. Para ello, definimos en `rag.py` una función llamada `setup()` que encapsula la lógica de configuración. Como primer paso, establecemos el nombre del proyecto, que nos permitirá enrutar las consultas a la base de conocimiento correspondiente, asegurándonos de que sea válido.

<< rag/rag.py
```
def setup(
  project_name: str,
) -> Runnable:
  if project_name not in VALID_PROJECTS:
    raise ValueError(f"{project_name} is not a valid project.")
```

Para generar respuestas, necesitamos instanciar un LLM con sus parámetros correspondientes —temperatura, *top-p* y *top-k*—, especificando el proveedor del modelo y su identificador.

```python
def setup(
  project_name: str,
  llm_provider: str,
  llm: str,
  temperature: float,
  top_p: float,
  top_k: int,
) -> Runnable:
  ...
  model = get_model(
    llm_provider, llm, temperature=temperature, top_p=top_p, top_k=top_k)
```
<< rag/rag.py

Ahora podemos avanzar con la fase de indexación. En este paso, la información preprocesada y limpia en texto plano del proyecto se divide en documentos con un tamaño `chunk_size` y un solapamiento `chunk_overlap`. Toda la lógica correspondiente al procesamiento de la base de conocimiento se encapsula en la función `create_docs()`, que se describe en la sección A.2. Asimismo, es fundamental especificar el modelo de *embeddings* que utilizaremos para generar la representación vectorial de los documentos, la cual se almacenará en una base de datos vectorial de ChromaDB. Para ello, utilizamos la función de ayuda `get_embeddings()` de la sección A.2 para obtener el modelo de *embeddings* e instanciamos la base de datos vectorial, indexando los documentos procesados.

```python
def setup(
  ...
  top_k: int,
  emb_provider: str,
  emb_model: str,
  chunk_size: int,
  chunk_overlap: int,
) -> Runnable:
  ...
  docs = create_docs(project_name, chunk_size, chunk_overlap)
  vectorstore = Chroma.from_documents(
    documents=docs,
    embedding=get_embeddings(emb_provider, emb_model))
```
<< rag/rag.py

Una vez completada la indexación, podemos avanzar a la fase de recuperación, donde se instancia el recuperador, definiendo el tipo de búsqueda (`search_type`) y el número de documentos a recuperar (`k_docs`). Una de las ventajas de ChromaDB es que incluye su propio recuperador, el cual se puede instanciar fácilmente mediante el método `as_retriever()`.

```python
def setup(
  ...
  chunk_overlap: int,
  search_type: str,
  k_docs: int,
) -> Runnable:
  ...
  retriever = vectorstore.as_retriever(
    search_type=search_type, search_kwargs={"k": k_docs})
```
<< rag/rag.py

A continuación, debemos crear la plantilla para el *prompt* del modelo, que combina el *prompt* de sistema con la consulta del usuario, permitiéndonos personalizarlo de acuerdo con nuestros intereses.

<< rag/rag.py

```
def setup(...) -> Runnable
  ...
  prompt = ChatPromptTemplate.from_messages(
    [("system", SYSTEM_PROMPT[project_name]), ("human", "{input}")])
```

Con todos los componentes preparados, es momento de unirlos en una única *pipeline* para construir el sistema RAG. En LangChain, el proceso comienza creando una *pipeline* de documentos, que combina el modelo con el *prompt*, y a partir de ahí, una *pipeline* de recuperación, que integra el recuperador junto con la anterior. Así, LangChain se encarga de orquestar toda la lógica subyacente: desde transformar la consulta en su representación vectorial, recuperar los documentos más relevantes e incorporarlos al *prompt*, hasta generar finalmente una respuesta a través del modelo de lenguaje.

<< rag/rag.py

```
def setup(...) -> Runnable
  ...
  qa_chain = create_stuff_documents_chain(llm, prompt)
  rag = create_retrieval_chain(retriever, qa_chain)
  return rag
```

Con todo configurado, el asistente virtual está listo para responder preguntas. Sin embargo, hasta este punto, el asistente no tiene memoria, sino que cada interacción ocurre de manera aislada. Para lograr una conversación más natural, incorporamos un historial de chat en el *prompt*, que se actualiza en cada intercambio con la nueva pregunta del usuario y la respuesta retornada por el modelo, preservando así la continuidad del diálogo.

<< rag/cli.py

```
def chat(rag: Runnable):
  history = []
  while True:
    ...
    response = rag.invoke({"history": history, "input": query})
    history.extend([("human", query), ("ai", response["answer"])])
    print(response["answer"])
```

Además, es necesario ajustar la estructura del `ChatPromptTemplate`, incorporando una nueva variable que contenga el historial de la conversación. Dado que la ventana de contexto del modelo es limitada, también debemos limitar el número máximo de mensajes (`n_messages`) que se incorporan al *prompt* a modo de historial.

<< rag/rag.py

```
prompt = ChatPromptTemplate.from_messages(
  [("system", SYSTEM_PROMPT[project_name]),
   MessagesPlaceholder(variable_name="history", n_messages=HISTORY_SIZE),
   ("human", "{input}")])
```

A continuación, una vez que tenemos la lógica del asistente definida, se construye un programa principal que lance la herramienta. Este interpreta la entrada por terminal, configura el sistema RAG e inicializa el flujo de interacción mediante la función `chat()`, introducida anteriormente.

<< rag/cli.py

```
def main():
  args = parse_args()
  rag = setup(
    project_name=args.project,
    llm_provider=args.llm_provider,
    llm=args.llm,
    temperature=args.temperature,
    top_p=args.top_p,
    top_k=args.top_k,
    emb_provider=args.emb_provider,
    emb_model=args.emb_model,
    chunk_size=args.chunk_size,
    chunk_overlap=args.chunk_overlap,
    search_type=args.search_type,
    k_docs=args.k,
  )
  chat(rag)
```

Finalmente, para lanzar la herramienta y comenzar a interactuar con el asistente virtual, podemos ejecutar el comando `python cli.py --project <project_name> [options]` desde la terminal, especificando las opciones que se consideren oportunas. Además, para poder visualizar las trazas en LangSmith, modificamos el programa principal de modo que se configure la utilización de la plataforma con las credenciales oportunas.

<< rag/cli.py

```
def main():
  args = parse_args()
  use_langchain(args.project)
  rag = setup(...)
  chat(rag)
```

Además de las variables de entorno configuradas en este punto, es necesario definir la variable `LANGCHAIN_API_KEY`, aunque no la incluimos directamente en el código, siguiendo buenas prácticas de seguridad.

<< rag/utils.py

```
def use_langchain(project_name: str):
  os.environ["LANGCHAIN_TRACING_V2"] = "true"
  os.environ["LANGCHAIN_ENDPOINT"] = "https://api.smith.langchain.com"
  os.environ["LANGCHAIN_PROJECT"] = project_name
```

SECCIÓN A.2

Módulos y funciones

Para obtener una versión reproducible de principio a fin de la implementación del asistente virtual, aquí se muestran los módulos requeridos en cada uno de *scripts* (`cli.py`, `rag.py` y `utils.py`) introducidos en la sección A.1. Además, también se presentan las funciones que se indicaron pero no se definieron en dicha sección.

A.2.1. CLI (`cli.py`)

El *script* con la lógica de control del asistente virtual, `cli.py`, necesita la inclusión de los siguientes módulos.

<< rag/cli.py

```python
import argparse
from argparse import ArgumentParser

from langchain_core.runnables import Runnable

from rag import setup
from utils import use_langchain
```

La función `parse_args()` nos permite configurar los parámetros del asistente directamente desde la terminal, estableciendo también valores por defecto para cada uno de ellos. Por ejemplo, el LLM seleccionado es el modelo Gemini 1.5 Flash de Google. Cabe señalar que `project` es un campo obligatorio, ya que sin él el sistema RAG no sabría a qué base de conocimiento referirse.

<< rag/cli.py

```python
def parse_args():
  parser = ArgumentParser(
    prog="python cli.py",
    description="A RAG-powered chatbot.",
    usage="python cli.py [options]",
    formatter_class=argparse.MetavarTypeHelpFormatter,
  )
  parser.add_argument("--project", type=str, required=True)
  parser.add_argument("--llm-provider", type=str, default="google")
  parser.add_argument("--llm", type=str, default="gemini-1.5-flash")
  parser.add_argument("--temperature", type=float, default=0.0)
  parser.add_argument("--top-p", type=float, default=0.85)
  parser.add_argument("--top-k", type=int, default=40)
  parser.add_argument("--emb-provider", type=str, default="google")
  parser.add_argument(
    "--emb-model", type=str, default="models/embedding-001"
  )
  parser.add_argument("--chunk-size", type=int, default=3000)
  parser.add_argument("--chunk-overlap", type=int, default=600)
  parser.add_argument("--search-type", type=str, default="similarity")
  parser.add_argument("--k", type=int, default=4)

  return parser.parse_args()
```

A.2.2. RAG (`rag.py`)

El script `rag.py` contiene la lógica del sistema RAG explicada en la sección A.1. Los módulos requeridos por este son los siguientes:

<< rag/rag.py

```python
import os
import re
import shutil

from langchain.chains.combine_documents import (
  create_stuff_documents_chain
)
from langchain.chains.retrieval import create_retrieval_chain
from langchain_community.vectorstores import Chroma
from langchain_core.documents import Document
from langchain_core.prompts import (
  ChatPromptTemplate, MessagesPlaceholder
)
from langchain_core.runnables import Runnable
from langchain_text_splitters import RecursiveCharacterTextSplitter

from utils import get_embeddings, get_model
```

Asimismo, en la sección A.1, se hacía uso de una función denominada `create_docs()`, que dividía la base de conocimiento en fragmentos con un solapamiento determinado entre ellos. Estos fragmentos son los documentos que posteriormente constituyen la base de datos vectorial. A continuación, se describe una versión simplificada de esta función.

En primer lugar, se instancia el separador de texto `RecursiveCharacterTextSplitter`.

<< rag/rag.py

```python
def create_docs(
  project_name: str, chunk_size: int, chunk_overlap: int
) -> list[Document]:
  # Declare splitter
  text_splitter = RecursiveCharacterTextSplitter(
    separators=["\n\n", "\n", " ", ""],
    chunk_size=chunk_size,
    chunk_overlap=chunk_overlap,
  )
```

Si el proyecto es el que queremos, en este caso `project_name="myproject"`, procedemos a crear los documentos. En caso contrario, lanzamos una excepción indicando que no hay datos para el proyecto especificado.

<< rag/rag.py

```python
def create_docs(...) -> list[Document]:
  ...
  if project_name == "myproject":
    # Do something
  else:
    raise ValueError(f"No data for {project_name}")
```

Dentro de la rama correspondiente a este proyecto en particular, escribimos toda la lógica de lectura de la base de conocimiento y separación en varios documentos. Primero, leemos el manual y las transcripciones de los vídeos de la base de conocimiento en texto plano.

<< rag/rag.py

```
def create_docs(...) -> list[Document]:
  ...
  if project_name == "myproject":
    with open("data/man_kb.txt", "r", encoding="utf-8") as f:
      man_text = f.read()
    with open("data/transcripts_kb.txt", "r", encoding="utf-8") as f:
      transcripts_text = f.read()
  else:
    raise ValueError(f"No data for {project_name}")
```

Finalmente, el contenido se fragmenta en documentos de menor tamaño para facilitar su procesamiento. En el caso del manual de uso de la aplicación, esta división se realiza mediante expresiones regulares, procurando que el texto correspondiente a una misma sección permanezca agrupado de manera coherente. Por otro lado, las transcripciones de los vídeos se segmentan utilizando el separador `RecursiveCharacterTextSplitter`, presentado previamente.

<< rag/rag.py

```
def create_docs(...) -> list[Document]:
  ...
  if project_name == "myproject":
    ...
    man_splits = re.split(MAN_SPLIT_PATTERN, man_text)
    transcript_splits = text_splitter.split_text(transcript_text)

    docs = [Document(page_content=chunk) for chunk in man_splits]
    docs.extend([Document(page_content=chunk) for chunk in transcript_splits])
  else:
    raise ValueError(f"No data for {project_name}")
  return docs
```

A.2.3. UTILS (`utils.py`)

El *script* `utils.py` reúne varias funciones de utilidad, entre ellas dos que actúan como una capa de abstracción para la instanciación de modelos de lenguaje y de *embeddings*. Los módulos necesarios son:

<< rag/utils.py

```
import os

import google.generativeai as genai
from langchain_core.embeddings import Embeddings
from langchain_core.language_models import LLM, BaseChatModel
from langchain_google_genai import (
  ChatGoogleGenerativeAI, GoogleGenerativeAIEmbeddings
)
from langchain_huggingface import HuggingFaceEndpoint
from langchain_ollama import ChatOllama
```

La función `get_model()` permite instanciar un LLM de manera flexible, abstrayendo tanto el proveedor del modelo como sus parámetros. Facilita el uso de modelos de Google, Ollama y HuggingFace, haciendo posible definir distintas configuraciones de evaluación de forma sencilla a través de ficheros de configuración.

<< rag/utils.py

```python
def get_model(
    provider: str,
    model_id: str,
    temperature: float,
    top_p: float = 0.85,
    top_k: int = 40,
) -> BaseChatModel | LLM:
  if provider == "google":
    # Configure Google AI Studio API key
    genai.configure(api_key=os.environ["GOOGLE_API_KEY"])
    llm = ChatGoogleGenerativeAI(
      model=model_id, temperature=temperature, top_p=top_p, top_k=top_k
    )
  elif provider == "ollama":
    llm = ChatOllama(model=model_id, temperature=temperature)
  elif provider == "hf":
    llm = HuggingFaceEndpoint(
      repo_id=model_id, temperature=temperature)
  else:
    raise ValueError(f"Unknown provider: {provider}")
  return llm
```

La función `get_embeddings()` es similar a `get_model()`, pero para la instanciación de modelos de *embeddings*. En este caso, solo se soportan los modelos de Google.

<< rag/utils.py

```python
def get_embeddings(provider: str, model_id: str) -> Embeddings:
  if provider == "google":
    embeddings = GoogleGenerativeAIEmbeddings(model=model_id)
  else:
    raise ValueError(f"Unknown provider: {provider}")
  return embeddings
```

APÉNDICE B

Baremo de evaluación manual

Con el fin de realizar una evaluación manual rigurosa de las respuestas generadas por el asistente virtual, se ha definido un baremo estructurado en tres criterios fundamentales: la capacidad del sistema para emitir una respuesta, la presencia de alucinaciones, y la calidad global de la respuesta proporcionada. Esta evaluación ha sido llevada a cabo por expertos en el dominio específico de la aplicación, siguiendo las directrices que se detallan a continuación:

1. **Capacidad para ofrecer una respuesta:** se valora si el asistente intenta responder a la pregunta planteada, con independencia de la corrección o relevancia del contenido.
   - **Sí:** el asistente emite una respuesta informativa, distinta a una negativa explícita.
   - **No:** el asistente declara explícitamente que no puede responder (por ejemplo, mediante expresiones del tipo «Lo siento, no sé responder a la pregunta»).
2. **Presencia de alucinaciones:** se determina si la respuesta contiene afirmaciones incorrectas, fabricadas o inconsistentes con respecto a la base de conocimiento disponible.
   - **Sí:** la respuesta incluye información que no tiene respaldo en la documentación de referencia, es incoherente o manifiestamente errónea.
   - **No:** la respuesta está fundamentada en hechos verificados que constan en el manual o en los vídeos tutoriales de la aplicación.
3. **Calidad general de la respuesta:** se evalúa el grado de utilidad y precisión de la respuesta ofrecida, según el siguiente esquema ordinal:
   - **Excelente:** la respuesta es completa, precisa y resuelve plenamente la consulta del usuario.
   - **Buena:** la respuesta es correcta, aunque presenta omisiones menores o aspectos que podrían desarrollarse más.
   - **Normal:** la respuesta es parcialmente útil, pero su redacción o nivel de detalle puede dificultar su comprensión, especialmente para usuarios inexpertos.
   - **Mala:** la respuesta no aporta información relevante o resulta confusa, imprecisa o inútil para resolver la consulta planteada.