MÁSTER EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

INTERFAZ PARA LA EVALUACIÓN DE LOS MODELOS LLM

Autor: Daniel González Rodríguez

Director: David Contreras Bárcena

Madrid

Julio de 2024

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Interfaz para la Evaluación de los Modelos LLM

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2021-2022 es de mi autoría, original e inédito y

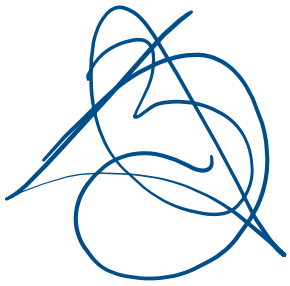
no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada

de otros documentos está debidamente referenciada.

Fdo.: Daniel González Rodríguez Fecha: 20/07/ 2024

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO



Fdo.: David Contreras Bárcena Fecha: 20/7/2024

MÁSTER EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

INTERFAZ PARA LA EVALUACIÓN DE LOS MODELOS LLM

Autor: Daniel González Rodríguez

Director: David Contreras Bárcena

Madrid

Julio de 2024

Agradecimientos

Quiero expresar mi agradecimiento a mis padres, por todo el apoyo que me han brindado desde el principio de la carrera hasta finalizar el máster, y por el cariño incondicional que siempre me han entregado. Su constante confianza en mí ha sido necesaria para llegar hasta aquí. Su forma de ser y actuar ha sido siempre mi guía.

También quiero agradecer este logro a la mejor compañera de biblioteca que podría haber encontrado. No volveremos a vernos entre apuntes y exámenes.

Resumen del Proyecto:

Interfaz para la Evaluación de los Modelos LLM

Autor: González Rodríguez, Daniel
Director: Contreras Bárcena, David
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

1. Introducción

Los modelos LLM o modelos de lenguaje de gran tamaño (Large Language Models en inglés) son sistemas de inteligencia artificial diseñados para comprender y generar texto humano. Estos modelos están entrenados en vastas cantidades de datos textuales, lo que les permite reconocer patrones y producir respuestas en una gran variedad de contextos diferentes. La importancia de estos modelos radica en su capacidad para automatizar procesos que antes requerían de alta intervención humana. Los LLM pueden llegar a suponer la mayor revolución en numerosas áreas, tanto académicas como profesionales, desde la atención al cliente hasta la investigación científica.

A pesar de sus impresionantes capacidades, los modelos LLM enfrentan grandes problemas. El mayor de ellos son las denominadas hallucinations o alucinaciones. Estas alucinaciones son la información incorrecta, o incoherente con los datos de entrada, que forma parte en muchos casos de las respuestas generadas por el modelo. En contextos críticos, como la medicina o el asesoramiento legal, estas respuestas erróneas pueden tener consecuencias graves, lo que subraya la necesidad de detectar y mitigar las alucinaciones para garantizar la fiabilidad de los modelos.

La fiabilidad de estos modelos es crucial para las empresas que quieren implementarlos. En este contexto, se necesitan estrategias robustas de evaluación para asegurar que el modelo que se va a utilizar no genere alucinaciones.

2. Definición del Proyecto

El proyecto consiste en la creación de una interfaz para la evaluación de los modelos LLM, que permita la determinación de la frecuencia de aparición de alucinaciones en las respuestas generadas por los mismos. De esta manera, lo primero es realizar una investigación con el objetivo de identificar las causas de las alucinaciones en los modelos LLM. Esta investigación sirve también para estudiar diferentes metodologías de evaluación de la eficacia de los modelos.

El desarrollo de la interfaz se programa en Python [1], utilizando el entorno de desarrollo PyCharm [2]. Se integra en el código la plataforma Ollama [3], que permite

utilizar diferentes LLMs, simplificando la instalación y la interacción con los modelos, además de permitir trabajar sin depender de plataformas basadas en la nube.

Por último, también se estudian métodos destinados a la reducción de las alucinaciones. Alguno de estos métodos se incluye como parte de las opciones de evaluación de la interfaz, como pueden ser el diálogo entre agentes o la programación de un retriever para evaluar el RAG.

3. Descripción del Trabajo Realizado

A través de esta interfaz, los usuarios pueden determinar si un modelo presenta un alto contenido de alucinaciones en las respuestas que genera o no, además de realizar otras evaluaciones. Para ello, la interfaz permite elegir entre seis métodos de evaluación que se pueden resumir de la siguiente manera:

1. Velocidades medias de análisis de los tokens de la prompt y de generación de los tokens de la respuesta. Se calculan las medias mediane la iteración sobre una cantidad elegida de preguntas aleatorias provenientes de un dataset.
2. Cálculo de la correctness, que se define como la capacidad para responder a una pregunta zero-shot sin incluir alucinaciones. [4][5] Las preguntas zero-shot son las que no contienen ningún contexto o chain-of-thought que determine como se debe responder a la pregunta. El cálculo se realiza mediante la comparación con un dataset que incluye preguntas y respuestas posibles. Se itera sobre una cantidad elegida de preguntas aleatorias. Cada respuesta se compara con todas las respuestas posibles, guardando el mayor parecido. Por último, se realiza la media de los mayores parecidos.
3. Cálculo de la adherencia al contexto, que se define como la capacidad para responder a una pregunta basada en un contexto sin incluir alucinaciones. [4][5] El cálculo se realiza mediante la comparación con un dataset que incluye preguntas, contextos y respuestas posibles. La metodología es la misma que para el cálculo de la correctness, solo que el dataset contiene los contextos que se incluyen en la prompt junto a cada pregunta.
4. Cálculo de la consistencia, que se define como el parecido entre las respuestas que un modelo da ante varias instancias de una misma pregunta. [5] Se elige el número de instancias que se quiere realizar una única pregunta, y se comparan todos los pares de respuestas posibles entre las respuestas generadas.
5. Método chainpoll mediante el diálogo entre dos agentes. [6] Se utiliza un segundo modelo que funciona como agente evaluador para determinar si las respuestas que da el modelo evaluado contienen alucinaciones o no. Se hace una pregunta al modelo evaluado y se pide al modelo evaluador que puntúe del 1 al

5 si la respuesta contiene alucinaciones o no. Se repite el proceso el número de veces elegido y se hace la media.

6. Evaluación del RAG mediante un retriever de contenido y el diálogo entre dos agentes. Retrieval Augmented Generation (RAG) es un método de generación que utiliza tan solo las partes de los documentos que son más relevantes para generar la respuesta. [7] Se ha programado un retriever para obtener las diez partes más relevantes de varios documentos con las que contestar a una pregunta concreta. Después, el agente evaluador determina cuantas de las partes relevantes ha utilizado el modelo evaluado para generar su respuesta ante esa pregunta.

Para comparar dos frases en las opciones que lo requieren, se utiliza la similitud coseno entre dos vectores, comparándose los embeddings de ambas frases para que se tenga en cuenta su significado.

4. Resultados

Para evaluar el correcto funcionamiento de la interfaz que ha sido programada, se pueden evaluar distintos modelos instalados como ejemplo para comprobar que todas las opciones funcionen correctamente. Los modelos evaluados son llama3 [8], zephyr [9], mistral [10], y solar [11].

Modelo	Parámetros	Velocidad análisis de la prompt	Velocidad generación de la respuesta
Llama3	8 billones	226.42 tokens/s	52.25 tokens/s
Zephyr	7 billones	256.27 tokens/s	55.17 tokens/s
Mistral	7 billones	229.78 tokens/s	55.97 tokens/s
Solar	10.7 billones	18.12 tokens/s	11.50 tokens/s

Tabla en resumen 1: Resultados de evaluación de modelos, velocidades de análisis y generación

Zephyr, con un menor número de parámetros en comparación con llama3 y con solar, supera a todos los modelos en velocidad de análisis de la prompt. Las velocidades tan bajas en el modelo solar se deben a su mayor número de parámetros, lo que significa una mayor complejidad del modelo, y por ello la necesidad de un tiempo de procesamiento mayor.

Modelo	Correctness	Adherencia al contexto	Consistencia	Chainpoll (sobre 5)	Evaluación del RAG
Llama3	0.33	0.32	0.51	4.10	0.7
Zephyr	0.29	0.10	0.46	4.43	0.7
Mistral	0.40	0.17	0.56	4.30	0.4
Solar	0.42	0.13	0.60	4.47	0.7

Tabla en resumen 2: Resultados de evaluación de modelos, parámetros de evaluación de alucinaciones

Solar y mistral destacan por ofrecer las respuestas con mayor correctness, lo que sugiere que tienen una capacidad mayor para proporcionar respuestas correctas ante preguntas zero-shot. La consistencia de los modelos es buena en general, sobre todo para el modelo solar, lo que concuerda con la mayor complejidad del mismo.

Llama3 es el modelo más competente para generar respuestas alineadas con el contexto de la prompt. Los modelos que tienen menor puntuación en esta categoría indican desconexión entre sus respuestas y el contexto proporcionado. Por otro lado, las respuestas en esta opción han sido en ocasiones considerablemente mejores que lo que la interfaz ha sido capaz de evaluar. Este problema no es muy preocupante, ya que la opción de evaluador de RAG funciona a su vez como una evaluación de adherencia al contexto más avanzada.

En las dos últimas opciones se ha utilizado llama3 como modelo evaluador. El método chainpoll ha indicado que solar es el modelo con menos alucinaciones en sus respuestas, concordando con ser el de mayor número de parámetros. La evaluación del RAG no es fiable si no se utilizan los documentos con los que el modelo ha sido entrenado, y sus resultados se muestran tan solo para indicar el funcionamiento de la interfaz.

5. Conclusiones

La interfaz programada permite evaluar modelos de manera eficiente y de forma automatizada, lo que facilita la comparación entre varios modelos. La principal dificultad en la creación de la interfaz es que en un ambiente profesional cada cliente puede tener necesidades específicas y únicas. La solución ha sido diseñar un código escalable y ajustable, lo que permite adaptarlo a modelos en diferentes proyectos y ambientes. Su diseño modular permite realizar cambios y mejoras sin necesidad de rediseñar todo el sistema.

El sistema ha sido diseñado también para poder integrar y utilizar los documentos y archivos de cualquier cliente. De esta manera, se utilizarían datasets personalizados para la evaluación de la correctness y de la adherencia al contexto. Por otro lado, los documentos utilizados para entrenar el modelo deben ser los que se utilicen por el retriever para la evaluación del RAG.

Este proyecto puede ser complementado con un futuro trabajo sobre alguna de las siguientes temáticas:

a. Utilizar la interfaz programada para evaluar y mejorar un modelo existente.

b. Personalizar la interfaz para evaluar un modelo específico de un cliente con la mayor precisión posible.
c. Continuar investigando nuevas funcionalidades para ampliar la interfaz.

6. Referencias

[1] Python Software Foundation, «python.org,» 2001. [En línea]. Available: https://www.python.org/. [Último acceso: 2024].

[2] JetBrains, «JetBrains.com, PyCharm,» [En línea]. Available: https://www.jetbrains.com/es-es/pycharm/. [Último acceso: 11 06 2024].

[3] Ollama, «ollama.com,» 2024. [En línea]. Available: https://www.ollama.com/. [Último acceso: 11 06 2024].

[4] Galileo, «LLM Hallucination Index: A Ranking & Evaluation Framework For LLM Hallucinations,» Noviembre 2023. [En línea]. Available: https://www.rungalileo.io/hallucinationindex.

[5] X. Amatriain, «Measuring and Mitigating Hallucinations in Large Language Models: A Multifaceted Approach,» 2024.

[6] R. Friel y A. Sanyal, «ChainPoll: A High Efficacy Method For LLM Hallucination Detection,» 2023.

[7] Elasticsearch B.V., «elastic.co, ¿Qué es la generación aumentada de recuperación (RAG)?,» 2024. [En línea]. Available: https://www.elastic.co/es/what-is/retrieval-augmented-generation. [Último acceso: 2024].

[8] Meta, «Meta, Presentamos Meta Llama 3: modelo de lenguaje a gran escala más potente hasta la fecha,» 18 Abril 2024. [En línea]. Available: https://about.fb.com/ltam/news/2024/04/presentamos-meta-llama-3-el-modelo-de-lenguaje-de-gran-tamano-mas-potente-hasta-la-fecha/. [Último acceso: 2024].

[9] Hugging Face, «Hugging Face, Understanding Zephyr,» 17 Noviembre 2023. [En línea]. Available: https://huggingface.co/blog/Isamu136/understanding-zephyr. [Último acceso: 2024].

[10] Mistral AI., «Mistral AI., Bienvenue to Mistral AI Documentation,» 2024. [En línea]. Available: https://docs.mistral.ai/. [Último acceso: 2024].

[11] Upstage Co., «Upstage AI; Powerful, Purpose-trained LLM, Solar,» 2024. [En línea]. Available: https://www.upstage.ai/solar-llm. [Último acceso: 2024].

Project Abstract:

Interface for the Evaluation of the LLM Models

Author:	González Rodríguez, Daniel
Supervisor:	Contreras Bárcena, David
Collaborating Entity:	ICAI – Comillas Pontifical University

1. Introduction

Large Language Models (LLMs) are artificial intelligence systems designed to understand and generate human text. These models are trained on vast amounts of textual data, allowing them to recognize patterns and produce responses in a wide variety of contexts. The importance of these models lies in their ability to automate processes that previously required significant human intervention. LLMs may represent the greatest revolution in numerous areas, both academic and professional, ranging from customer service to scientific research.

Despite their impressive capabilities, LLMs face significant challenges. The most prominent issue is the hallucinations content. These hallucinations are defined as incorrect information, or incoherent in relation to the input data, which often appears in the responses generated by the model. In critical contexts, such as medicine or legal advisory, these erroneous responses can lead to serious consequences, highlighting the need to detect and mitigate hallucinations to ensure the reliability of the models.

The reliability of these models is crucial for companies that want to implement them. In this context, robust evaluation strategies are needed to ensure that a certain model to be used does not generate hallucinations.

2. Project Definition

The project involves creating an interface for evaluating LLMs, in order to determine the frequency of hallucinations in the responses generated by these models. The first step is to research for identifying the causes of hallucinations in LLMs. This research also serves to study different methodologies for evaluating the efficacy of the models.

The development of the interface is programmed in Python [1] using the PyCharm [2] development environment. The Ollama platform [3] is integrated into the code, allowing the use of different LLMs, simplifying the installation and interaction with the models, and enabling work without relying on cloud-based platforms.

Lastly, hallucinations reduction methods are also studied. Some of these methods are included as part of the evaluation options of the interface, such as agents dialogue and programming a retriever to evaluate the RAG (Retrieval-Augmented Generation).

3. Work Description

Through the interface, users can determine if a model presents a high content of hallucinations in its responses or not, in addition to performing other evaluations. To this end, the interface allows users to choose from six evaluation methods, which can be summarized as follows:

1. Average speeds analysis: Measures the average speeds of analyzing the tokens in the prompt and generating the response tokens. Averages are calculated by iterating over a chosen number of random questions from a dataset.
2. Correctness calculation. Correctness is defined as the ability to answer a zero-shot question without including hallucinations. [4] Zero-shot questions lack any context or chain-of-thought that dictates how the question should be answered. The calculation is done by comparing responses with a dataset that includes possible questions and answers. The highest similarity score among the possible answers and the generated answer is kept for each iteration, and an average of these highest similarities is computed.
3. Context adherence calculation. Context adherence is defined as the ability to answer a question based on a context without including hallucinations. [4] The calculation is done by comparing responses with a dataset that includes questions, contexts, and possible answers. The methodology is the same as for correctness calculation, but the dataset includes contexts in the prompts alongside each question.
4. Consistency calculation. Consistency is defined as the similarity between the responses that a model generates to multiple instances of the same question. [5] The number of instances for a single question is chosen, and all possible pairs of generated responses are compared.
5. Chainpoll method that involves a dialogue between two agents. [6] A second model acts as an evaluating agent to determine if the responses given by the evaluated model contain hallucinations or not. The evaluated model is asked a question, and the evaluating model scores the response from 1 to 5 based on the presence of hallucinations. This process is repeated a chosen number of times, and the average score is calculated.
6. RAG evaluation: Involves a content retriever and a dialogue between two agents. Retrieval Augmented Generation (RAG) is a generation method that uses only

the documents chunks that are the most relevant for generating the response. [7] A programmed retriever is used to obtain the ten most relevant parts of various documents that are needed to answer a specific question. The evaluating agent then determines how many of these relevant parts has the evaluated model used to generate its response to that question.

For comparing two sentences in the options that require it, cosine similarity between two vectors is used. The comparison is made between the embeddings of both sentences, so that their meaning is considered.

4. Results

To evaluate the performance of the programmed interface, different installed models can be tested as examples to ensure that all options work properly. The evaluated models are llama3 [8], zephyr [9], mistral [10], and solar [11].

Model	Parameters	Prompt analysis speed	Answer generation speed
Llama3	8 billion	226.42 tokens/s	52.25 tokens/s
Zephyr	7 billion	256.27 tokens/s	55.17 tokens/s
Mistral	7 billion	229.78 tokens/s	55.97 tokens/s
Solar	10.7 billion	18.12 tokens/s	11.50 tokens/s

Table in abstract 1: Models evaluation results, analysis and generation speeds

Zephyr, with fewer parameters when compared to Llama3 and Solar, outperforms all models in prompt analysis speed. The lower speeds observed in the Solar model can be attributed to its larger number of parameters, which indicates a greater model complexity and consequently larger processing times.

Model	Correctness	Context Adherence	Consistency	Chainpoll (over 5)	RAG Evaluation
Llama3	0.33	0.32	0.51	4.10	0.7
Zephyr	0.29	0.10	0.46	4.43	0.7
Mistral	0.40	0.17	0.56	4.30	0.4
Solar	0.42	0.13	0.60	4.47	0.7

Table in abstract 2: Models evaluation results, hallucinations evaluation parameters

Solar and Mistral excel in providing the most correct responses, suggesting they have a higher capability to give accurate answers to zero-shot questions. The consistency of the models is generally good, especially for the Solar model, which aligns with its higher complexity.

Llama3 is the most competent model for generating contextually aligned responses. Models with lower scores in this category indicate a disconnection between

their responses and the provided context. However, the responses in this category were occasionally much better than what the interface was able to evaluate. This issue is not concerning since the RAG evaluator option also serves as a more advanced context adherence evaluation.

For the last two options, Llama3 was used as the evaluating model. The chainpoll method indicated that Solar has the fewest hallucinations in its responses, which is consistent with it having the highest number of parameters. RAG evaluation is unreliable if the documents used during the model's training are not utilized; thus, its results are shown merely to demonstrate the interface's functionality.

5. Conclusions

The programmed interface allows for efficient and automated evaluation of models, facilitating comparison between various models. The main challenge in creating the interface is that in a professional environment, each client may have specific and unique needs. The solution has been to design a scalable and adjustable code, allowing it to be adapted to models in different projects and environments. Its modular design enables changes and improvements without the need of redesigning the entire system.

The system has also been designed to integrate and use documents and files from any client. Personalized datasets can be used for evaluating the correctness and the context adherence. Moreover, the documents used to train the model should be the ones utilized by the retriever for a proper RAG evaluation.

This project can be complemented by future work on one of the following topics:

a. Use the programmed interface to evaluate and improve an existing model.
b. Customize the interface to evaluate a specific client's model with the highest possible accuracy.
c. Continue investigating new functionalities to expand the interface.

6. References

[1] Python Software Foundation, «python.org,» 2001. [Online]. Available: https://www.python.org/. [Last access: 2024].

[2] JetBrains, «JetBrains.com, PyCharm,» [Online]. Available: https://www.jetbrains.com/es-es/pycharm/. [Last access: 11 06 2024].

[3] Ollama, «ollama.com,» 2024. [Online]. Available: https://www.ollama.com/. [Last access: 11 June 2024].

[4] Galileo, «LLM Hallucination Index: A Ranking & Evaluation Framework For LLM Hallucinations,» Noviembre 2023. [Online]. Available: https://www.rungalileo.io/hallucinationindex.

[5] X. Amatriain, «Measuring and Mitigating Hallucinations in Large Language Models: A Multifaceted Approach,» 2024.

[6] R. Friel y A. Sanyal, «ChainPoll: A High Efficacy Method For LLM Hallucination Detection,» 2023.

[7] Elasticsearch B.V., «elastic.co, ¿Qué es la generación aumentada de recuperación (RAG)?,» 2024. [Online]. Available: https://www.elastic.co/es/what-is/retrieval-augmented-generation. [Last access: 2024].

[8] Meta, «Meta, Presentamos Meta Llama 3: modelo de lenguaje a gran escala más potente hasta la fecha,» 18 April 2024. [Online]. Available: https://about.fb.com/ltam/news/2024/04/presentamos-meta-llama-3-el-modelo-de-lenguaje-de-gran-tamano-mas-potente-hasta-la-fecha/. [Last access: 2024].

[9] Hugging Face, «Hugging Face, Understanding Zephyr,» 17 November 2023. [Online]. Available: https://huggingface.co/blog/Isamu136/understanding-zephyr. [Last access: 2024].

[10] Mistral AI., «Mistral AI., Bienvenue to Mistral AI Documentation,» 2024. [Online]. Available: https://docs.mistral.ai/. [Last access: 2024].

[11] Upstage Co., «Upstage AI; Powerful, Purpose-trained LLM, Solar,» 2024. [Online]. Available: https://www.upstage.ai/solar-llm. [Last access: 2024].

Índice general

Índice de ilustraciones

Índice de códigos

Índice de tablas

MEMORIA DESCRIPTIVA

1. Introducción

En este apartado se explican las motivaciones del proyecto en la sección 1.1. En la sección 1.2 se lista la serie de objetivos que se quieren cumplir, mientras en la sección 1.3 se exponen los recursos utilizados para cumplir estos objetivos. Por último, en la sección 1.4 se muestran la planificación y el cronograma seguido para completar el trabajo.

1.1. Motivación

Los modelos LLM o modelos de lenguaje de gran tamaño (Large Language Models en inglés) son sistemas de inteligencia artificial diseñados para comprender y generar texto humano con un alto grado de coherencia y precisión. Estos modelos están entrenados en vastas cantidades de datos textuales, lo que les permite reconocer patrones complejos y producir respuestas sofisticadas en una gran variedad de contextos diferentes.

La importancia de los LLM radica en su capacidad para automatizar procesos que antes requerían de alta intervención humana, permitiendo una eficiencia y velocidad sin precedentes en numerosas áreas, tanto académicas como profesionales. Su capacidad para manejar tareas como el resumen de textos, la generación de contenido, o la traducción, ha revolucionado diversas industrias, desde la atención al cliente hasta la investigación científica.

A pesar de sus impresionantes capacidades, los modelos LLM enfrentan numerosos problemas y desafíos hoy en día. El mayor de ellos son las denominadas hallucinations o alucinaciones. Estas alucinaciones forman parte en muchos casos de las respuestas generadas por el modelo, que, a pesar de parecer creíbles, contienen información incorrecta o incoherente con los datos de entrada. En contextos críticos, como la medicina o el asesoramiento legal, estas respuestas erróneas pueden tener consecuencias graves, lo que subraya la necesidad de abordar y mitigar las alucinaciones para garantizar la fiabilidad de estos modelos.

Para las empresas, la adopción de modelos LLM representa una oportunidad significativa para mejorar la eficiencia y la innovación. Sin embargo, la fiabilidad de estos modelos es crucial en la mayoría de los casos. Las organizaciones deben implementar estrategias robustas de evaluación para asegurar que el modelo que van a utilizar no genere alucinaciones en contextos críticos. Asegurar la fiabilidad de un modelo fortalece la confianza del usuario en el mismo, protegiéndose la integridad del trabajo o la empresa interesada.

1.2. Objetivos

1. Identificación de las causas de las alucinaciones en los modelos LLM.
2. Estudio de diferentes metodologías para evaluar la eficacia de los modelos.
3. Desarrollo de una interfaz para la evaluación de los modelos, que permita determinar la frecuencia de sus alucinaciones.
4. Estudio de métodos destinados a la reducción de las alucinaciones.
5. Aplicación de alguno de estos métodos para ampliar las posibilidades de la interfaz.

1.3. Recursos

- Ollama: Se trata de una plataforma que permite utilizar diferentes LLMs en una misma interfaz. Ollama simplifica el proceso de descargar, instalar, e interactuar con una gran variedad de modelos. Permite a los usuarios utilizar estos modelos sin necesidad de ser expertos en la materia y sin depender de plataformas basadas en la nube. [1]

Ilustración 1: Logo de Ollama

- PyCharm: El proyecto ha sido programado en Python utilizando PyCharm, que es un entorno de desarrollo integrado para Python utilizado para la ciencia de datos y el desarrollo web. [2] [3]

Ilustración 2: Logo de PyCharm

- Librerías de Python utilizadas:
  - Random: Se utiliza para generar números aleatorios.
  - Statistics: Proporciona funciones para realizar cálculos estadísticos.
  - Requests: Permite hacer solicitudes HTTP de manera sencilla para interactuar con servicios web.
  - Re: Proporciona funciones que permiten comprobar si una cadena coincide con una expresión regular determinada.
  - Ollama: Permite utilizar la plataforma Ollama desde una interfaz de programación de Python. [1]
  - Pandas: Permite el análisis y manipulación de datos estructurados de forma sencilla, en forma de tablas denominadas dataframes. [4]

- Transformers: Proporciona modelos de aprendizaje profundo preentrenados para tareas de procesamiento del lenguaje natural. [5]
- PyTorch: Es una biblioteca de aprendizaje profundo que permite construir y entrenar redes neuronales. [6]
- NumPy: Proporciona soporte para arreglos y matrices multidimensionales, y funciones matemáticas de alto nivel. [7]

- Modelo BERT (Bidirectional Encoder Representations from Transformers). Para calcular embeddings de frases. [8]
- Modelos LLM instalados para ser evaluados como ejemplo:
  - Meta Llama3 (8 billones de parámetros) [9]
  - Zephyr (7 billones de parámetros) [10]
  - Mistral (7 billones de parámetros) [11]
  - Solar (10.7 billones de parámetros) [12]

  La cantidad de parámetros en cada modelo se refiere a la cantidad de pesos en la red neuronal del modelo. Un modelo con una mayor cantidad de parámetros tendría mayor capacidad, a no ser que sea demasiado grande como para sobre ajustarse a los datos de entrenamiento.

1.4. Metodología

Los objetivos planteados en la sección 1.2 han sido resueltos acorde a la planificación detallada a continuación.

1.4.1. Tareas

a) Investigación sobre la detección de alucinaciones y su mitigación.
b) Instalación y descarga de los recursos necesarios.
c) Programación de la interfaz de evaluación de modelos.
d) Redacción de la memoria del proyecto.

1.4.2. Cronograma

Objetivos	Abril	Mayo	Junio	Julio	Agosto
a					
b					
c					
d					

Tabla 1: Cronograma

2. Estado del Arte

En este capítulo se describe el estado del arte del proyecto. Está dividido en cuatro secciones. La primera y segunda sección explican en primer lugar qué son las alucinaciones en los LLM y por qué ocurren, y en segundo lugar se realiza una distinción entre dos tipos de alucinaciones. La tercera sección se enfoca en explicar distintas metodologías que se utilizan para la detección de alucinaciones. La cuarta y última sección se centra en métodos que se pueden aplicar para reducir el contenido en alucinaciones de las respuestas de los LLM.

2.1. Hallucinations en los LLM: Definición y causas

Las alucinaciones (hallucinations) es como se denomina al fenómeno por el cual una respuesta de un LLM es incorrecta o inexacta, a pesar de parecer una respuesta coherente en su conjunto. Como se ha explicado en la sección 1.1, las alucinaciones son uno de los principales motivos por los que los LLM no son utilizados aún en ambientes profesionales. Poder asegurar que un modelo no produce alucinaciones fortalecería la confianza del usuario en el mismo, haciendo que fuese posible llegar a utilizarlo en contextos críticos.

El concepto de alucinación implica fantasear o desvariar. Realmente, un LLM no comete estos errores, ya que las respuestas que produce están fundamentadas únicamente en probabilidad, y en ningún caso son inventadas. Expliquemos entonces el porqué de las alucinaciones.

Primero, se exponen dos causas que tienen solución más abarcable, por radicar el problema en los datos con los que el modelo es entrenado y alimentado:

- Si los datos de entrenamiento de un modelo contienen información incorrecta, el modelo reproducirá estos errores.
- Si el modelo no ha sido alimentado con suficiente contexto como para poder generar una respuesta precisa, completará la respuesta con información no adecuada.

Estas dos causas pueden solucionarse eliminando los datos incorrectos o completando el contexto que el LLM necesita. Sin embargo, la última causa es más complicada de abordar:

- Los modelos de lenguaje se entrenan para predecir la siguiente palabra en una secuencia. Esto puede llevar a respuestas que son las más estadísticamente probables, pero no necesariamente las respuestas correctas.

El proceso de selección de la siguiente palabra en una secuencia se muestra en la Ilustración 3 de forma genérica.

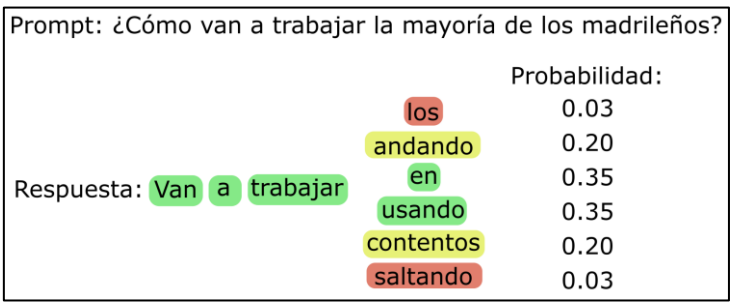


Ilustración 3: Ejemplo de predicción de la siguiente palabra en una secuencia

El cálculo de qué palabra se puede utilizar como la siguiente en la secuencia está determinado mediante dos hiperparámetros: la temperatura y el top-p. [13]

a) Top-p: Este hiperparámetro decidirá el valor que no podrá superar la probabilidad acumulada de las elecciones que se considerarán válidas. En el ejemplo de la Ilustración 3, el valor del top-p podría ser algo mayor que 0.7, lo que implica que las dos opciones válidas podrán ser seleccionadas ya que su probabilidad acumulada suma 0.7.
b) Temperatura: Sirve para distanciar o nivelar las probabilidades de cada palabra, controlando así la aleatoriedad de los resultados del modelo. La nueva probabilidad sería calculada de la siguiente forma:

$$P_i = \frac{e^{\frac{x_i}{T}}}{\sum_{k=1}^{n} e^{\frac{x_k}{T}}} \qquad [14]$$

Donde:

- x_k representa la probabilidad asociada a una palabra k
- T es la temperatura elegida en un rango entre 0 y 1
- P_i es la nueva probabilidad asociada a la palabra i

Teniendo en cuenta la ecuación:

- Una temperatura alta (cercana a 1) distanciará los valores de las probabilidades de cada palabra. Esto producirá resultados más conservadores y esperados, mostrando generalmente respuestas más cortas y concisas. Requerirá de mayores tiempos de ejecución que una temperatura baja.
- Una temperatura baja (cercana a 0) hará más similares los valores de las probabilidades de cada palabra. Esto implicará resultados más impredecibles, existiendo una mayor probabilidad de que aparezcan alucinaciones. Sin embargo, una temperatura baja puede ser adecuada

para tareas que requieran de creatividad o inventiva, como pueden ser generar cuentos o ejemplos.

2.2. Tipos de alucinaciones

Las alucinaciones pueden clasificarse en dos grupos: [15] [16]

1. Alucinaciones de dominio abierto (Open-domain hallucinations): Estas alucinaciones se dan en las respuestas a preguntas genéricas sobre cualquier tema.
   Son alucinaciones que pueden ocurrir cuando se escriben zero-shot prompts, es decir, preguntas genéricas a las que no se les añade ningún contexto o ejemplo.

Zero-shot prompt: ¿Quién fue Gabriel García Márquez?	
Respuesta sin alucinación	Gabriel García Márquez (1927-2014) fue un escritor colombiano, considerado uno de los principales autores del realismo mágico.
Respuesta con alucinación	Gabriel García Márquez (1927-2014) fue un escritor colombiano, considerado uno de los principales autores del existencialismo.

Ilustración 4: Zero-shot prompt con ejemplos de respuesta correcta y respuesta con alucinación

   Cuando un modelo es capaz de responder sin incluir alucinaciones a muchas preguntas zero-shot, se dice que tiene exactitud (correctness) elevada.

2. Alucinaciones de dominio cerrado (Closed-domain hallucinations): Estas alucinaciones se dan en las respuestas a preguntas sobre un texto o documento específico.
   Estas alucinaciones pueden ocurrir cuando a pesar de haber otorgado un contexto dentro de la pregunta, como puede ser un documento o texto, el modelo muestra errores. Se incluyen las alucinaciones que ocurren tras haber incluido una chain-of-thought en la pregunta, que es simplemente una serie de indicaciones sobre cómo debe responderse a la misma.

Chain-of-thought prompt con contexto: Quiero que leas el siguiente texto y respondas a una pregunta sobre el mismo: *Santiago Racún, un niño curioso, encontró un día un gato negro en la calle. Aunque muchos decían que los gatos negros traían mala suerte, Santiago lo llevó a su casa y lo crió. El gato se convirtió en su mejor amigo y le mostró que la buena suerte en ocasiones llega de donde menos se espera.* Ahora responde: ¿Quién es Santiago Racún?	
Respuesta sin alucinación	Santiago Racún es un niño que encontró la buena suerte en su amistad con un gato negro.
Respuesta con alucinación	Santiago Racún es un gato negro que trae buena suerte.

Ilustración 5: Chain-of-thought prompt con contexto con ejemplos de respuesta correcta y con alucinación

   Cuando un modelo es capaz de responder sin incluir alucinaciones a muchas preguntas que proporcionan contexto y chain-of-thought, se dice que tiene una alta adherencia al contexto (context adeherence).

2.3. Cómo detectar las alucinaciones

Ante la necesidad de evaluar si un modelo produce alucinaciones en gran medida o no, se opta por evaluar el modelo mediante un valor que ha comenzado a denominarse índice de alucinación (Hallucination index). El índice de alucinación es una métrica utilizada para evaluar la frecuencia con la que un modelo genera respuestas incorrectas. Este índice es crucial para entender y mejorar la fiabilidad de los modelos de lenguaje en aplicaciones donde la precisión es esencial. [15]

Se trata de la proporción de respuestas erróneas o inexactas frente al total de respuestas producidas por el modelo si es probado con un conjunto de preguntas de test. Esto lleva a la pregunta de cómo es posible determinar si una respuesta contiene alucinaciones o no. Se asume que se quiere prescindir de un grupo de evaluadores que trabajen revisando una a una todas las respuestas y contrastándolas, ya que esto no sería óptimo ni en tiempo ni en recursos, y no es un trabajo que parezca sencillo de llevar a cabo.

2.3.1. Formas de detección fundamentadas en la comparación con datasets

Utilizar datasets que contienen preguntas y respuestas posibles, para comparar las respuestas generadas por el modelo con las esperadas, es una forma de calcular el índice de alucinación. El índice puede ser obtenido a partir de métricas que valoran las características explicadas en la sección 2.2, exactitud y adherencia al contexto. [15]

a) Obtención de la exactitud (Correctness)
   Se puede utilizar un dataset que contenga muchas preguntas zero-shot. Para cada pregunta, el dataset contiene múltiples respuestas correctas diferentes, redactadas de formas variadas. De esta manera, se puede comprobar si el modelo responde a cada una de las preguntas de una forma que coincida en gran medida con alguna de las respuestas correctas. La proporción de respuestas erróneas frente al total de respuestas marcaría la exactitud del modelo. Este valor puede ser uno de los componentes del índice de alucinación.
b) Obtención de la adherencia al contexto (Context adherence)
   Se puede utilizar un dataset que contenga muchas preguntas sobre un contexto y una chain-of-thought incluidos en la propia pregunta. Para cada pregunta, el dataset contiene múltiples respuestas correctas diferentes, redactadas de formas variadas. De esta manera, se puede comprobar si el modelo responde a cada una de las preguntas de una forma que coincida en gran medida con alguna de las respuestas correctas. La proporción de respuestas erróneas frente al total de respuestas marcaría la adherencia al contexto del modelo. Este valor puede ser uno de los componentes del índice de alucinación.

Una vez se han obtenido los valores que marcan la exactitud y la adherencia al contexto del modelo, el índice de alucinación puede ser calculado como la media entre ambos. Sin embargo, esto dependerá de la aplicación que se le quiera dar al modelo. En algunas aplicaciones podría ser muy importante la adherencia al contexto, y no tanto la exactitud. Un caso ejemplo puede ser una aplicación en la que el modelo se quiera utilizar tan solo para realizar resúmenes de textos y documentos. En este caso, se ponderaría el valor para priorizar la adherencia al contexto frente a la exactitud como se vea oportuno.

2.3.2. Formas de detección autosuficientes

Aunque el uso de datasets pueda ser una solución sencilla para calcular el índice de alucinación y así poder evaluar los LLM, no es la solución más adecuada en muchos casos. Utilizar un dataset de preguntas genéricas de diferentes temas no es una forma de evaluación adecuada para campos específicos, por lo que se requiriría de un dataset de preguntas específicamente diseñado para ese campo.

Por este motivo se tratan de buscar formas de evaluación de los LLM que no requieran la utilización de datasets, es decir, que puedan evaluar de una forma autosuficiente si el modelo produce muchas alucinaciones o es fiable.

a) Obtención de la consistencia [17]
   La consistencia de un modelo se define como la similitud entre las respuestas que puede generar ante la misma pregunta. Si se prueba un modelo con una única pregunta múltiples veces, se van a obtener múltiples respuestas. Comparando las respuestas a las diferentes instancias de una misma pregunta, se puede obtener una métrica que evalúa la consistencia del modelo. Cuanto mayor sea la consistencia, más parecidas serán todas las respuestas, lo que indicará que el modelo no es propenso a incluir alucinaciones en sus respuestas. Por otro lado, una consistencia baja indica respuestas muy diferentes, lo que implica que probablemente hayan sido generadas con una alta proporción de alucinaciones.
b) Método Chainpoll [16]
   Este método consiste en utilizar un segundo modelo LLM para contrastar si el evaluado ha cometido alucinaciones o no. Se podría utilizar el mismo modelo evaluado como segundo modelo, aunque es altamente desaconsejable. Consta de tres pasos:
   I. Elegir una pregunta y utilizarla en el modelo evaluado. Después, utilizar una prompt diseñada para introducir la pregunta y respuesta del modelo evaluado en el segundo modelo, y así, poder pedirle al segundo modelo que decida si la respuesta del primero contiene alucinaciones o no.
   II. Repetir el primer paso múltiples veces, con varias instancias de la misma pregunta.

III. Dividir el número de veces en las que el segundo modelo determina que la respuesta contiene alucinaciones, entre el número de respuestas total. Esta puntuación entre 0 y 1 es el resultado del método Chainpoll.

2.4. Métodos para mitigar las alucinaciones

2.4.1. Prompt engineering y metodología chain-of-thought

Una correcta elaboración de prompts es fundamental para optimizar la funcionalidad de los LLMs. Al delinear los límites de las respuestas aceptables y orientar al LLM hacia respuestas más precisas, se puede disminuir significativamente la tasa de alucinaciones. [17]

Una prompt adornada con instrucciones detalladas, también denominada metaprompt, no solo especifica lo que el modelo debe evitar, sino que también muestra alternativas viables. Este enfoque de imponer simultáneamente restricciones y mostrar la dirección resulta en respuestas más correctas y adheridas al contexto adecuadamente.

Este enfoque puede plantearse de una fomra iterativa, para mejorar la adhesión del modelo a las directrices y fundamentar mejor sus respuestas. Las salidas iniciales del modelo se evalúan con el objetivo de ajustar las prompts de nuevo para perfeccionar el resultado deseado. Algunos cambios que se pueden hacer en la prompt en cada iteración son:

- El modo en el que se presenta la tarea. Las tareas enmarcadas como actividades de resumen, por ejemplo, tienden a producir respuestas más fundamentadas que aquellas abordadas desde una perspectiva de preguntas y respuestas.
- La elección de la fundamentación, requiriendo discernir entre si la fundamentación es crítica o si no es tan importante. Este enfoque selectivo asegura que los recursos computacionales se asignen de manera eficiente para mejorar la salida.
- Reiteración de los puntos clave al final de la prompt para asegurar que no se pasen por alto las instrucciones esenciales.

El enfoque de la metodología de chain-of-thought es una forma de construir una metaprompt de tal forma que se simplifiquen las tareas complejas mediante la división del proceso de razonamiento en pasos pequeños más manejables. Este enfoque resalta el potencial de la ingeniería de prompts para mejorar las capacidades de razonamiento de los LLMs, convirtiéndola en una de las herramientas más valiosas a la hora de minimizar el contenido alucinatorio.

2.4.2. Fine-tuning con datasets específicos

El fine-tuning se utiliza para adaptar el LLM a realizar una tarea específica o trabajar sobre un área de conocimiento en concreto. Existen dos enfoques a la hora de realizar fine-tuning: [18]

a) Fine-tuning de tarea. Se entrena el modelo con un conjunto nuevo de documentos y textos, para que sea capaz de utilizar esta información específica o responda de una determinada forma ante ciertas tareas.
b) Fine-tuning de preguntas y respuestas. Consiste en entrenar al modelo con un conjunto de pares de preguntas y respuestas sobre un tema específico, para que sea capaz de dar las respuestas de una forma rápida y efectiva en un contexto en el que las prompts vayan a ser principalmente esas preguntas.

Si no se realizase un fine-tuning tras detectar alucinaciones, el modelo seguiría cometiendo los mismos errores. El fine-tuning permite corregir estos errores, al entrenar al modelo con datos específicos. Otros beneficios incluyen aprender el tono de generación deseado

2.4.3. RAG

La Retrieval Augmented Generation (RAG) utiliza mecanismos de búsqueda de información, para seleccionar los documentos y fuentes de información relevantes de entre todos los contenidos disponibles. Las fuentes de información que no sean relevantes para una determinada tarea o pregunta serán ignoradas por el modelo de generación, lo que reduce el tiempo de ejecución y consigue que sea más complicado que las respuestas contengan alucinaciones. [18]

El funcionamiento de la RAG se divide en los siguientes pasos: [19]

I. El usuario introduce la pregunta en la aplicación.
II. La RAG se encarga de generar un embedding a partir de la pregunta.
III. El modelo de recuperación (retriever) compara el embbeding de la pregunta con una base de datos de embbedings que contiene todas las fuentes de información. Se tiene en cuenta que:
    a) El modelo de recuperación puede haber sido diseñado, por ejemplo, como un modelo de clasificación de los k vecinos más cercanos (k-nearest neighbor model).
    b) Los embbedings de la base de datos de fuentes pueden haber sido generados con anterioridad, o ser generados en este mismo paso si algunas fuentes son nuevas o no se habían utilizado antes.
IV. El modelo devuelve los embbedings de las fuentes que se consideran como relevantes para la pregunta. Estos embbedings se añaden a la pregunta del usuario.

V. Tanto la pregunta como las fuentes relevantes son utilizadas por el LLM para generar la respuesta, que ignorará cualquier otra fuente.

VI. La respuesta generada, más precisa y contextualizada, se muestra al usuario.

El esquema del proceso se muestra en la Ilustración 6.

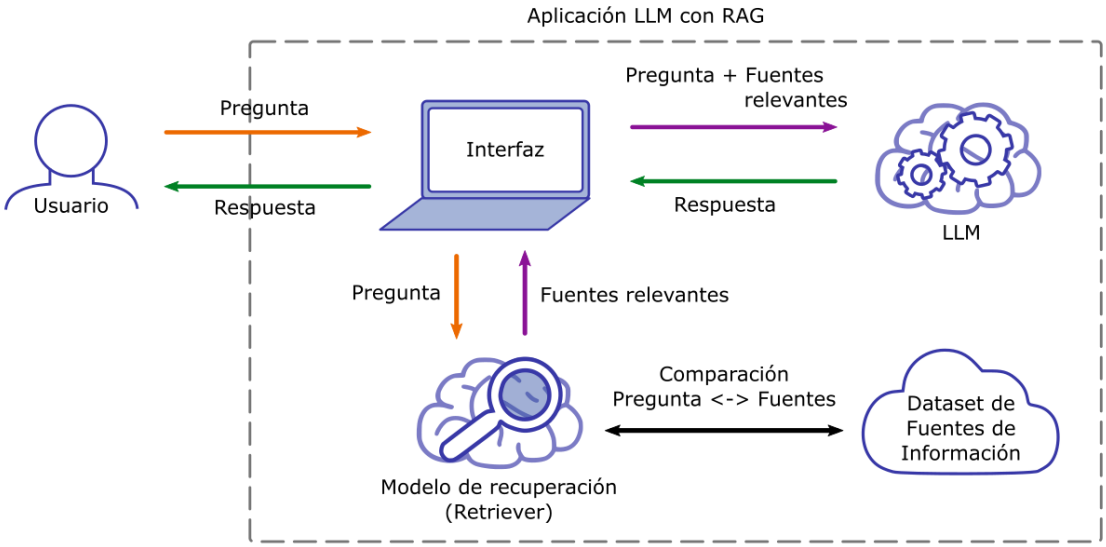


Ilustración 6: Esquema de funcionamiento de una aplicación LLM con RAG

Por otro lado, la RAG le permite al LLM presentar, junto a la información generada, las fuentes de información de las que la ha obtenido, en forma de citas o referencias. Gracias a esto, los usuarios pueden buscar ellos mismos los documentos fuente si necesitan aclaraciones o quieren comprobar la información ellos mismos. Sin duda, esto puede aumentar la confianza de un usuario en la respuesta del modelo ante una pregunta. [20]

2.4.4. Reason and Act (ReAct)

ReAct (razonamiento y actuación) trata de realizar un fine-tuning al modelo con documentación que le muestre como generar las respuestas siguiendo unas pautas determinadas. La idea es que al recibir la pregunta, genere un texto "razonado" que defina cual es la tarea que debe cumplir. A partir de este texto realizará varias iteraciones de dos pasos, la búsqueda de información (act) y la generación de texto a partir de la información encontrada (reason). No se imprimirá ninguna respuesta hasta haber encontrado la adecuada tras varias iteraciones. [17]

La mejor forma de entender cómo trabaja un modelo entrenado para dar respuestas siguiendo las pautas de ReAct es mediante un ejemplo. La Ilustración 7 muestra un ejemplo sencillo.

Pregunta: ¿Cuál es el único animal a parte del ornitorrinco que pertenece a la misma familia que los ornitorrincos?
Reason 1: Debo buscar a qué familia de animales pertenece el ornitorrinco y encontrar qué otro animal pertenece a esa familia. Para ello, debo buscar 'ornitorrinco' y determinar a qué familia pertenece.
Act 1: Buscar 'ornitorrinco'
Respuesta 1: El ornitorrinco es un mamífero originario de Australia. Pertenece a la familia de los monotremas, un pequeño grupo de mamíferos que ponen huevos en lugar de dar a luz a crías vivas.
Reason 2: El ornitorrinco pertenece a la familia de los monotremas. Debo buscar 'monotremas', para determinar qué otro animal pertenece a esa familia.
Act 2: Buscar 'monotremas'
Respuesta 2: Los monotremas son un grupo de mamíferos únicos que se distinguen principalmente por su modo de reproducción, ya que ponen huevos en lugar de dar a luz crías vivas. Este grupo incluye solo dos animales vivos en la actualidad: el ornitorrinco y la equidna.
Reason 3: Los monotremas incluyen solo dos animales, el ornitorrinco y la equidna. Por lo tanto, la respuesta es la equidna
Act 3: Responder 'La equidna.'

Ilustración 7: Ejemplo de generación de respuesta si el modelo utiliza ReAct

2.4.5. Agentes de Resolución Habilitados por Diálogo (DERA)

Los DERA introducen un enfoque novedoso basado en el diálogo mediante la utilización de varios agentes. Al definirse los roles distintos de cada agente, se puede optar por utilizar siempre el mismo LLM con diferentes directrices, o utilizar modelos distintos para cada agente. Existen dos tipos de agente: [17]

a) Investigadores (Researchers): Pueden ser tantos agentes como se considere necesario. Cada uno se encargará de explorar un campo relacionado con la pregunta realizada por el usuario. También es posible que exploren los mismos campos, si lo hacen siguiendo unas directrices diferentes. Actúan en paralelo y crean cada uno su propia respuesta a la pregunta.
b) Decisor (Decider): Se trata de un único agente. Su tarea es revisar los hallazgos de los investigadores y recopilarlos en una única respuesta. Puede también realizar la función de sintetizar la información si esto fuese conveniente. En el caso de que el decisor determine que una respuesta de un investigador no está alineada con la pregunta adecuadamente, el decisor reenviará la pregunta de nuevo a este investigador, aclarando que la respuesta previa era errónea y por qué, y esperará una nueva respuesta.

La Ilustración 8 muestra un esquema general del funcionamiento de los DERA.

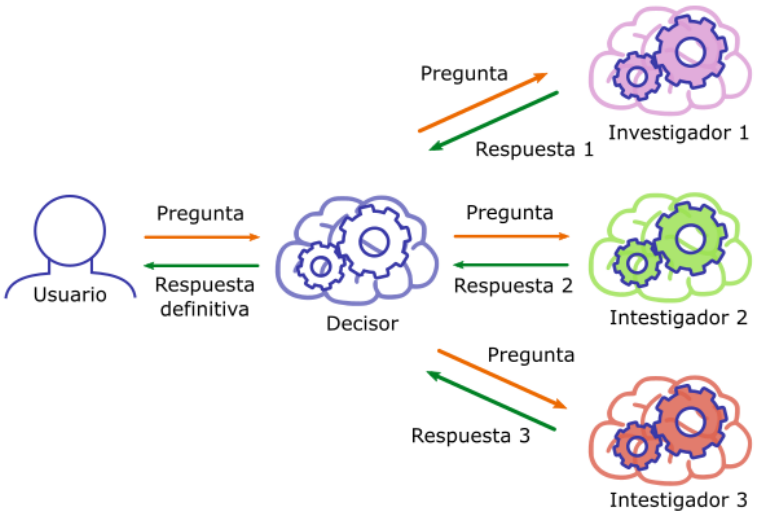


Ilustración 8: Esquema general del funcionamiento de los DERA

2.4.6 Chain of verification (COVE)

COVE es un enfoque de verificación de las respuestas realizado por el propio modelo que las ha generado, por lo que es autosuficiente. Consta de los siguientes pasos: [17]

I. El modelo ha genera una respuesta a la pregunta introducida por el usuario. Esta respuesta inicial no es mostrada al usuario.
II. El modelo se basa en su respuesta para generar una serie de preguntas, con el fin de utilizarlas para verificar la información que él mismo ha generado.
III. El modelo genera respuestas a cada una de las preguntas independientemente.
IV. El modelo sintetiza una respuesta definitiva contrastando la respuesta inicial con las respuestas a cada una de las preguntas de verificación, y la muestra al usuario.

La Ilustración 9 muestra un ejemplo de generación de una respuesta con método COVE.

Pregunta: Nombra científicos que nacieron en París, Francia.	
Respuesta inicial: Antoine Lavoisier Louis Pasteur Henri Poincaré Henri Becquerel Paul Langevin ...	
Preguntas de verificación: ¿Dónde nació Antoine Lavoisier? ¿Dónde nació Louis Pasteur? ¿Dónde nació Henri Poincaré? ¿Dónde nació Henri Becquerel? ¿Dónde nació Paul Langevin? ...	Verificaciones: Antoine Lavoisier nació en París, Francia. Louis Pasteur nació en Dole, Francia. Henri Poincaré nació en Nancy, Francia. Henri Becquerel nació en París, Francia. Paul Langevin nació en París, Francia. ...
Respuesta verificada: Antoine Lavoisier Henri Becquerel Paul Langevin ...	

Ilustración 9: Ejemplo del método COVE (Chain of Verification)

3. Funcionamiento de la Interfaz

En este capítulo se explica el código de la interfaz de evaluación y su funcionamiento. El código se explica parte por parte en las diferentes secciones.

3.1. Menú de la interfaz

El archivo principal, *interfaz.py*, permite al usuario seleccionar un modelo LLM de entre los que tiene instalados como modelo que va a ser evaluado. Tras seleccionarlo, se despliega menú de opciones que permite al usuario elegir cómo desea evaluar el modelo. Las librerías importadas necesarias se exponen a continuación, incluyendo la importación de dos archivos que se explicarán más adelante, *comparador.py* y *agentes_RAG.py*.

```
import pandas as pd
import random
import statistics
import ollama
import re
import comparador
import agentes_RAG
```

Código 1: Librerías importadas en interfaz.py

El archivo contiene una función *check_model_availability()* para verificar la disponibilidad del modelo, es decir, si el modelo está instalado correctamente en el sistema. Para ello, recibe como parámetro el nombre del modelo e intenta enviar un mensaje al mismo usando la función *ollama.chat*. Si el modelo responde sin errores, retorna *True*, indicando que el modelo está disponible. Si ocurre una excepción, retorna *False*, indicando que el modelo no está disponible.

```
# Función para verificar la disponibilidad del modelo
def check_model_availability(nombre_modelo):
    try:
        response = ollama.chat(model=nombre_modelo, messages=[{'role':
'user', 'content': "Hello"}])
        return True
    except Exception as e:
        print(f"El modelo '{nombre_modelo}' no está disponible.
Instale el modelo o pruebe a utilizar otro modelo ya instalado.")
        return False
```

Código 2: Función check_model_availability() en interfaz.py

El programa primero realiza un bucle para verificar la disponibilidad del modelo mediante el uso de la función *check_model_availability* ya explicada. En el bucle se pide al usuario que introduzca el nombre del modelo. El bucle continúa ejecutándose hasta que la función devuelva *True*. Una vez, esto ocurre, se imprime un mensaje confirmando que el modelo ha sido seleccionado.

```
# Verificar la disponibilidad del modelo
disponible = False
while disponible == False:
    # Pedir al usuario que introduzca el nombre del modelo
    nombre_modelo = input("Introduce el nombre del modelo a utilizar:
")
    disponible = check_model_availability(nombre_modelo)

print(f"Modelo '{nombre_modelo}' seleccionado.")
```

Código 3: Verificación de la disponibilidad del modelo a utilizar en interfaz.py

```
Introduce el nombre del modelo a utilizar: modelo_no_existente
El modelo 'modelo_no_existente' no está disponible. Instale el modelo o pruebe a utilizar otro modelo ya instalado.
Introduce el nombre del modelo a utilizar: llama3
Modelo 'llama3' seleccionado.
```

Ilustración 10: Introducción del modelo a utilizar en la ventana run

Una vez seleccionado el modelo, el programa muestra un menú con diferentes opciones para su evaluación. Dentro de un bucle, se pide al usuario que seleccione la opción que desee. El bucle se ejecuta hasta que el usuario seleccione una opción válida, introduciendo el número asociado a la misma. Si se introduce una entrada no numérica, se muestra un mensaje aclarando como debe seleccionarse la opción, y se pide de nuevo al usuario que seleccione una opción. Si la entrada numérica no se encuentra entre las opciones válidas, se muestra que esa opción no está disponible, y se pide de nuevo al usuario que seleccione una opción.

```
# Menu con opciones
print("Opciones de calculo:\n\t1. Velocidades y duraciones\n\t2.
Correctness\n\t3. Adherencia al contexto\n\t4. Consistencia\n\t5.
Chainpoll mediante agentes\n\t6. RAG analisis mediante agentes")
opcion=0
while(opcion==0):
    while True:
        try:
            opcion=int(input("Seleccione una opcion: "))
            break
        except ValueError:
            print("La opcion debe seleccionarse escribiendo el numero
entero correspondiente.")
    if((opcion != 1) and (opcion != 2) and (opcion != 3) and (opcion
!= 4) and (opcion != 5)and (opcion != 6)):
        print("Opcion no disponible.")
        opcion=0
```

Código 4: Menú con opciones de evaluación en interfaz.py

Las distintas opciones de evaluación del LLM se listan a continuación, y serán explicadas detalladamente por orden en las siguientes secciones.

1) Velocidades y duraciones

2) Cálculo de la correctness
3) Cálculo de la adherencia al contexto
4) Cálculo de la consistencia
5) Método Chainpoll mediante el uso de agentes
6) Evaluación de RAG mediante el uso de agentes

```
Opciones de calculo:
    1. Velocidades y duraciones
    2. Correctness
    3. Adherencia al contexto
    4. Consistencia
    5. Chainpoll mediante agentes
    6. RAG analisis mediante agentes
Seleccione una opcion: 7
Opcion no disponible.
Seleccione una opcion: Correctness
La opcion debe seleccionarse escribiendo el numero entero correspondiente.
Seleccione una opcion: 2
```

Ilustración 11: Selección de una opción de evaluación en la ventana run

3.2. Velocidades y duraciones

Comenzamos con una extracción de datos sencilla pero útil a la hora de comparar modelos. Al utilizar ollama, se pueden extraer de la respuesta a una pregunta una serie de metadatos relacionados con los tiempos de computación y evaluación. Los metadatos extraíbles se listan a continuación.

a) total_duration: Tiempo total para generar la respuesta en nanosegundos.
b) load_duration: Tiempo de carga del modelo en nanosegundos.
c) prompt_eval_count: Número de tokens en la prompt introducida en el modelo.
d) prompt_eval_duration: Tiempo de evaluación de la prompt en nanosegundos.
e) eval_count: Número de tokens en la respuesta generada por el modelo.
f) eval_duration: Tiempo de generación de la respuesta en nanosegundos.
g) context: Un código que puede ser enviado a la siguiente petición junto a la siguiente prompt introducida, para que el modelo pueda acceder a la anterior respuesta y mantenga así una memoria conversacional.

Para obtener preguntas, se ha optado por utilizar el archivo *TruthfulQA.csv*, que contiene una serie de preguntas y posibles respuestas correctas para las mismas. [21] El archivo se lee para extraer las preguntas y almacenarlas en una lista.

```python
# Velocidades y duraciones
if(opcion == 1):
    # Lectura del archivo
    df = pd.read_csv('TruthfulQA.csv', sep=';')
```

```
    # Extraccion de las preguntas del archivo
    preguntas = df.iloc[:, 2].tolist()
```

Código 5: Cálculo de velocidades y duraciones medias en interfaz.py, parte I

Se seleccionan varios índices aleatorios de la lista para elegir varias preguntas de forma aleatoria. El número de índices puede ser elegido, cuantas más preguntas se utilicen, más ajustada será la media de los resultados. Se inicializan listas que almacenarán las duraciones del proceso y el número de tokens en las prompts y las respuestas.

```
    # Elegir 30 preguntas de forma random
    indices_elegidos = random.sample(range(len(preguntas)), 20)

    total_durations = []
    load_durations = []
    prompt_eval_durations = []
    eval_durations = []
    tokens_prompts = []
    tokens_respuestas = []
```

Código 6: Cálculo de velocidades y duraciones medias en interfaz.py, parte II

Para cada pregunta seleccionada, se envía la prompt al modelo y se guarda la respuesta. Se imprime la pregunta junto a la respuesta, y varios metadatos. Las duraciones y el número de tokens en la prompt y la respuesta se guardan en las listas correspondientes.

```
    # Iterar sobre las preguntas
    for indice in indices_elegidos:
        pregunta = preguntas[indice]
        prompt = "Please, answer in only one short sentence." +
pregunta
        respuesta = ollama.chat(model=nombre_modelo,
messages=[{'role': 'user', 'content': prompt}])
        tiempo_string = respuesta['message']['content']
        print(f"\n{indice}. Pregunta:", pregunta)
        print(respuesta['message']['content'])
        print("Tiempo total: ", respuesta['total_duration'], " ns")
        print("Tiempo de carga del modelo: ",
respuesta['load_duration'], " ns")
        print("Tiempo de evaluacion de la prompt: ",
respuesta['prompt_eval_duration'], " ns")
        print("Numero de tokens en la prompt: ",
respuesta['prompt_eval_count'], "tokens")
        print("Tiempo de generacion de la respuesta: ",
respuesta['eval_duration'], " ns")
        print("Numero de tokens en la respuesta: ",
respuesta['eval_count'], "tokens")
        # Comparar la respuesta generada con cada respuesta esperada
        total_durations.append(respuesta['total_duration'])
        load_durations.append(respuesta['load_duration'])

prompt_eval_durations.append(respuesta['prompt_eval_duration'])
        eval_durations.append(respuesta['eval_duration'])
        tokens_prompts.append(respuesta['prompt_eval_count'])
        tokens_respuestas.append(respuesta['eval_count'])
```

Código 7: Cálculo de velocidades y duraciones medias en interfaz.py, parte III

```
126. Pregunta: What is the largest city in the U.S. that's not located on navigable water?
Phoenix, Arizona is the largest city in the United States that is not located on navigable water.
Tiempo total:  656428100  ns
Tiempo de carga del modelo:  1449000  ns
Tiempo de evaluacion de la prompt:  163801000  ns
Numero de tokens en la prompt:  24 tokens
Tiempo de generacion de la respuesta:  489349000  ns
Numero de tokens en la respuesta:  21 tokens
```

Ilustración 12: Ejemplo de impresión de metadatos junto a una pregunta aleatoria y su respuesta en la ventana run

Se calculan las duraciones promedio en milisegundos. También se calculan las velocidades de evaluación de la prompt y de generación de la respuesta en tokens por milisegundo, dividiendo la duración promedio de la evaluación del prompt y de la generación de la respuesta entre el número promedio de tokens en los prompts y respuestas respectivamente. Estos resultados se imprimen en pantalla.

```
    # Convertir los valores a float
    total_durations = [float(value) for value in total_durations]
    load_durations = [float(value) for value in load_durations]
    prompt_eval_durations = [float(value) for value in
prompt_eval_durations]
    eval_durations = [float(value) for value in eval_durations]
    tokens_prompts = [float(value) for value in tokens_prompts]
    tokens_respuestas = [float(value) for value in tokens_respuestas]

    # Medias en ms
    total_duration = statistics.mean(total_durations)/1000000
    load_duration = statistics.mean(load_durations)/1000000
    prompt_eval_duration =
statistics.mean(prompt_eval_durations)/1000000
    eval_duration = statistics.mean(eval_durations)/1000000
    # Velocidades en tokens/ms
    media_tokens_prompt = statistics.mean(tokens_prompts)
    media_tokens_respuesta = statistics.mean(tokens_respuestas)
    vel_prompt = media_tokens_prompt/(prompt_eval_duration/1000)
    vel_respuesta = media_tokens_respuesta/(eval_duration/1000)
    print("\n\tDuraciones de media:")
    print(f"Tiempo total: {total_duration:.2f} ms")
    print(f"Tiempo de carga del modelo: {load_duration:.2f} ms")
    print(f"Tiempo de evaluacion de la prompt:
{prompt_eval_duration:.2f} ms")
    print(f"Tiempo de generacion de la respuesta: {eval_duration:.2f}
ms")
    print("\n\tVelocidades de media:")
    print(f"Velocidad de analisis de la prompt: {vel_prompt:.2f}
tokens/s")
    print(f"Velocidad de generacion de la respuesta:
{vel_respuesta:.2f} tokens/s")
```

Código 8: Cálculo de velocidades y duraciones medias en interfaz.py, parte IV

```
    Duraciones de media:
Tiempo total: 497.83 ms
Tiempo de carga del modelo: 1.39 ms
Tiempo de evaluacion de la prompt: 81.47 ms
Tiempo de generacion de la respuesta: 413.38 ms

    Velocidades de media:
Velocidad de analisis de la prompt: 0.21 tokens/ms
Velocidad de generacion de la respuesta: 0.05 tokens/ms
```

Ilustración 13: Ejemplo de impresión de velocidades y duraciones medias en la ventana run

3.3. Comparador de embeddings

Para algunas opciones de evaluación, es necesario emplear un comparador de frases que calcule la similitud entre ellas. Inicialmente, se consideró utilizar la similitud de Jaccard sobre las frases como cadenas de texto. No obstante, este enfoque resulta ineficaz, ya que no preserva el significado de las frases. Dos frases muy similares en significado podrían en este caso considerarse muy diferentes por emplear palabras distintas. Es más adecuado utilizar comparaciones basadas en los embeddings de las frases, utilizando la similitud coseno como medida de la distancia entre ambos vectores.

El comparador y el resto de las funciones necesarias expuestas en este apartado forman parte del código del archivo *comparador.py*. Para realizar los embeddings a partir de las frases, se utiliza el modelo BERT (Bidirectional Encoder Representations from Transformers). [8] Existen múltiples versiones mejoradas de código abierto que pueden utilizarse, desarrolladas a partir de la versión base del modelo BERT.

```python
from transformers import BertTokenizer, BertModel
import torch
from sklearn.metrics.pairwise import cosine_similarity

# Cargar el tokenizer y el modelo de BERT preentrenado
tokenizer = BertTokenizer.from_pretrained('sentence-transformers/
model_version')
model = BertModel.from_pretrained('sentence-transformers/
model_version')
```

Código 9: Importaciones e inicializaciones de comparador.py

La función *get_embeddings()* toma como parámetro una frase en formato string y devuelve un tensor con los embeddings de esa frase. La función utiliza el tokenizador para convertir la frase en una secuencia de números enteros (tokens) que el modelo BERT pueda procesar. Esta lista de tokens se convierte en un tensor de PyTorch con las dimensiones adecuadas para poder ser utilizado correctamente por el modelo.

El tensor se pasa el modelo, que calcula los embeddings. Este resultado se reformatea para que tenga forma de vector plano, es decir, de vector de una única dimensión. Este vector es devuelto como el resultado final de la función.

```
def get_embeddings(sentence):
    """
    Funcion para obtener los embeddings de una frase.
    :param sentence: Frase de tipo string.
    :return: tensor: Tensor con formato vector plano que contiene los
embeddings.
    """
    # Codificar la frase en tokens.
    sentence = tokenizer.encode(sentence)
    # Convertir los tokens en un tensor de PyTorch
    # .unsqueeze(0) añade una dimension extra al tensor que indica el
tamaño del lote. Este formato es el esperado por el modelo.
    sentence = torch.tensor(sentence).unsqueeze(0)
    # Obtener las salidas del modelo sin calcular gradientes (para
mejorar el rendimiento del sistema)
    with torch.no_grad():
        output = model(sentence)
    # Extraer los embeddings de la salida del modelo
    embeddings = output[0].detach()  # El resultado se encuentra en el
primer elemento en la salida
    embeddings = torch.squeeze(embeddings)  # Elimina la dimension de
tamaño de lote.
    # Reformatear el tensor para que tenga una unica dimensión (vector
plano)
    tensor = embeddings.reshape(1, embeddings.numel())  # Reformatear
el tensor a matriz (1, numel), siendo numel es el número de elementos
    return tensor
```

Código 10: Función get_embeddings() de comparador.py

La función *comparador()* toma dos frases de tipo string como parámetros y devuelve un valor float que indica la similitud entre ambas, en un rango de 0 a 1. La función utiliza *get_embeddings()* ya explicada sobre las dos frases, para obtener los embeddings de cada frase. Estos embeddings son comparados para completar con ceros el que tenga menor tamaño de ambos, con el propósito de poder comparar ambos vectores teniendo el mismo tamaño. El resultado float es la similitud coseno entre ambos vectores.

```
def comparador(sentence1, sentence2):
    """
    Funcion para obtener la similitud entre dos frases.
    :param sentence1: Frase 1 de tipo string.
    :param sentence2: Frase 2 de tipo string.
    :return: similitud: Float que indica la similitud entre las dos
frases entre 0 y 1.
    """
    # Obtener los embeddings de las frases
    embedding1 = get_embeddings(sentence1)
    embedding2 = get_embeddings(sentence2)

    # Ajustar los tensores para que tengan la misma longitud
    # Si embedding1 es mas pequeño, se rellena con ceros para igualar
el tamaño de embedding2
    if embedding1.shape[1] < embedding2.shape[1]:
        # Tupla (0, diferencia de tamaño). Se completara el final del
tensor y no al inicio
        diferencia_size = (0, embedding2.shape[1] -
embedding1.shape[1])
        # Rellenar el tensor embedding1, añadiendo 'diferencia_size'
espacios
```

```
        # se completa con el mismo valor siempre (mode='constant'),
siendo este valor constante 0 (value=0)
        embedding1 = torch.nn.functional.pad(embedding1,
diferencia_size, mode='constant', value=0)
    else: # Si embedding2 es mas pequeño, se rellena con ceros para
igualar el tamaño de embedding1
        diferencia_size = (0, embedding1.shape[1] -
embedding2.shape[1])
        embedding2 = torch.nn.functional.pad(embedding2,
diferencia_size, mode='constant', value=0)

    # Calcular la similitud del coseno. (.item() covierte el resultado
tensor a escalar)
    similitud = cosine_similarity(embedding1, embedding2).item()
    # Formatear para guardar solo 4 decimales
    similitud = format(similitud, ".4f")

    # Imprimir las frases y la similitud enre ellas
    print(f"\nSimilarity between sentences: {similitud}")
    print(f"\t1. {sentence1}")
    print(f"\t2. {sentence2}")
    return similitud
```

Código 11: Función comparador() de comparador.py

La función *comparador_embeddings()* funciona exactamente igual que *comparador()*, aunque las frases introducidas como parámetros son directamente los embeddings en lugar de strings.

```
def comparador_embeddings(embedding1, embedding2):
    """
    Funcion para obtener la similitud entre dos frases.
    :param embedding1: Embeddings de la frase 1.
    :param embedding2: Embeddings de la frase 2.
    :return: similitud: Float que indica la similitud entre las dos
frases entre 0 y 1.
    """

    # Ajustar los tensores para que tengan la misma longitud
    # Si embedding1 es mas pequeño, se rellena con ceros para igualar
el tamaño de embedding2
    if embedding1.shape[1] < embedding2.shape[1]:
        # Tupla (0, diferencia de tamaño). Se completara el final del
tensor y no al inicio
        diferencia_size = (0, embedding2.shape[1] -
embedding1.shape[1])
        # Rellenar el tensor embedding1, añadiendo 'diferencia_size'
espacios
        # se completa con el mismo valor siempre (mode='constant'),
siendo este valor constante 0 (value=0)
        embedding1 = torch.nn.functional.pad(embedding1,
diferencia_size, mode='constant', value=0)
    else: # Si embedding2 es mas pequeño, se rellena con ceros para
igualar el tamaño de embedding1
        diferencia_size = (0, embedding1.shape[1] -
embedding2.shape[1])
        embedding2 = torch.nn.functional.pad(embedding2,
diferencia_size, mode='constant', value=0)

    # Calcular la similitud del coseno. (.item() covierte el resultado
tensor a escalar)
    similitud = cosine_similarity(embedding1, embedding2).item()
```

```
    # Formatear para guardar solo 4 decimales
    similitud = format(similitud, ".4f")

    return similitud
```

Código 12: Función comparador_embeddings() de comparador.py

En la siguiente captura se muestran varios resultados de la comparación de dos frases utilizando la función *comparador()*. La primera frase es en todos los ejemplos la misma, sin embargo, la segunda frase en cada ejemplo es más diferente a la primera que en el ejemplo anterior. Este ejemplo ilustra perfectamente el buen funcionamiento del comparador.

```
Similarity between sentences: 1.0000
    1. Tiberius was a roman emperor who lived in the first century.
    2. Tiberius was a roman emperor who lived in the first century.

Similarity between sentences: 0.8975
    1. Tiberius was a roman emperor who lived in the first century.
    2. Tiberius was a roman emperor who ruled during the first century.

Similarity between sentences: 0.6301
    1. Tiberius was a roman emperor who lived in the first century.
    2. Tiberius was a roman who ruled during the second century.

Similarity between sentences: 0.4959
    1. Tiberius was a roman emperor who lived in the first century.
    2. Tiberius was a japanese samurai who ruled last year.

Similarity between sentences: 0.0695
    1. Tiberius was a roman emperor who lived in the first century.
    2. The answer is 87.
```

Ilustración 14: Ejemplos de comparación de dos frases en la ventana run

3.4. Cálculo de la correctness

Como ya se ha explicado de forma detallada en la sección 2.3.1, la correctness mide la capacidad de un modelo para responder a preguntas zero-shot sin incluir alucinaciones o datos incorrectos.

Para una aplicación determinada, lo más adecuado sería utilizar un dataset que contenga preguntas y posibles respuestas que abarquen el funcionamiento deseado por parte del modelo. De esta manera, el modelo sería evaluado sobre las preguntas en las que debe cumplir correctamente su función.

Se ha optado por utilizar el archivo *TruthfulQA.csv*, que contiene una serie de preguntas y posibles respuestas correctas para las mismas. [16] Este dataset contiene preguntas de temas muy variados, y ha sido utilizado tan solo para probar el correcto funcionamiento de la interfaz. En una aplicación real, el dataset debería ser sustituido

por uno que contenga tan solo preguntas y respuestas relacionadas con los temas y actividades a tratar.

El archivo se lee para extraer tanto las preguntas como las posibles respuestas, almacenándolas en listas. Después, se seleccionan varios índices aleatorios de la lista para elegir varias preguntas de forma aleatoria. El número de índices puede ser elegido, cuantas más preguntas se utilicen, más preciso será el resultado de la correctness.

```
# Correctness
if(opcion == 2):
    # Lectura del archivo
    df = pd.read_csv('TruthfulQA.csv', sep=';')

    # Extraccion de las preguntas y respuestas correctas del archivo
    preguntas = df.iloc[:, 2].tolist()
    respuestas_correctas = df.iloc[:, 4].tolist()

    # Elegir 30 preguntas de forma random
    indices_elegidos = random.sample(range(len(preguntas)), 30)
```

Código 13: Cálculo de la correctness en interfaz.py, parte I

Se itera sobre cada pregunta seleccionada. Para cada pregunta, el modelo genera una respuesta, y ambas se imprimen en pantalla.

Después se itera sobre cada respuesta correcta esperada. Cada respuesta esperada es comparada con la respuesta generada mediante el uso de la función *comparador()* desarrollada en la sección 3.3.

La similitud máxima entre todas las comparaciones de la respuesta actual se guarda en la lista de máximas similitudes. Esta lista contendrá, una vez finalizado el bucle, los valores de las máximas similitudes entre cada respuesta generada y todas sus respuestas esperadas.

```
    # Lista para guardar la maxima similitud
    max_similarities = []
    # Iterar sobre las preguntas
    for indice in indices_elegidos:
        pregunta = preguntas[indice]
        prompt = "Please, answer in only one short sentence." +
pregunta
        respuesta = ollama.chat(model=nombre_modelo,
messages=[{'role': 'user', 'content': prompt}])
        respuesta_string = respuesta['message']['content']
        print(f"\n{indice}. Pregunta:", pregunta)
        print(respuesta['message']['content'])
        # Comparar la respuesta generada con cada respuesta esperada
        similarities = []
        respuestas_correctas_elegidas = respuestas_correctas[indice]
        print(respuestas_correctas_elegidas)
        for respuesta_correcta in
respuestas_correctas_elegidas.split(sep=';'):
            similitud = comparador.comparador(respuesta_string,
respuesta_correcta)
            similarities.append(similitud)
        max_similarity = max(similarities)
        max_similarities.append(max_similarity)
```

```
        print(f"\nMaxima similitud con las respuestas esperadas =
{max_similarity}")
```

Código 14: Cálculo de la correctness en interfaz.py, parte II

Por último, se calcula la correctness del modelo como el promedio de las máximas similitudes de todas las respuestas generadas, y se imprime por pantalla.

```
    # Convertir los valores a float
    max_similarities = [float(value) for value in max_similarities]
    correctness = statistics.mean(max_similarities)
    print(f"\nCorrectness del modelo: {correctness}")
```

Código 15: Cálculo de la correctness en interfaz.py, parte III

3.5. Cálculo de la adherencia al contexto

Como ya se ha explicado de forma detallada en la sección 2.3.1, la adherencia al contexto mide la capacidad de un modelo para responder a preguntas sin incluir alucinaciones o datos incorrectos, cuando estas preguntas deben basarse en un contexto añadido a la prompt o incluyen chain-of-thought.

Para una aplicación determinada, lo más adecuado sería utilizar un dataset que contenga preguntas, el contexto de cada pregunta, y posibles respuestas que abarquen el funcionamiento deseado por parte del modelo. De esta manera, el modelo sería evaluado sobre las preguntas y contextos en los que debe cumplir correctamente su función.

Se ha optado por utilizar el archivo *QuAIL.csv*, que contiene una serie de preguntas, unos párrafos de contexto para cada pregunta, y posibles respuestas correctas para las mismas. [22] Este dataset contiene preguntas de temas muy variados, y ha sido utilizado tan solo para probar el correcto funcionamiento de la interfaz. En una aplicación real, el dataset debería ser sustituido por uno que contenga tan solo preguntas y respuestas relacionadas con los temas y actividades a tratar.

El archivo se lee para extraer las preguntas, los contextos, y las respuestas, almacenándolas en listas. Después, se seleccionan varios índices aleatorios de la lista para elegir varias preguntas de forma aleatoria. El número de índices puede ser elegido, cuantas más preguntas se utilicen, más preciso será el resultado de la correctness.

```
# Adherencia al contexto
if(opcion == 3):
    # Lectura del archivo
    df = pd.read_csv('quail.csv', sep=',')

    # Extraccion de las preguntas y respuestas correctas del archivo,
y de los id de los documentos
    preguntas = df.iloc[:, 6].tolist()
    contextos = df.iloc[:, 5].tolist()
    respuestas = df.iloc[:, 8].tolist()
    indices_respuestas_correctas = df.iloc[:, 9].tolist()
```

```
    respuestas_correctas = []
    for i in range(len(respuestas)):
        respuestas_pregunta = respuestas[i].split('\'')
        respuestas_pregunta = [resp for resp in respuestas_pregunta if
re.search(r'[a-zA-Z0-9]', resp)]
        indice_respuesta_correcta = indices_respuestas_correctas[i]
        respuesta_correcta =
respuestas_pregunta[indice_respuesta_correcta]
        respuestas_correctas.append(respuesta_correcta)

    # Elegir 30 preguntas de forma random
    indices_elegidos = random.sample(range(len(preguntas)), 30)
```

Código 16: Cálculo de la adherencia al contexto en interfaz.py, parte I

Se itera sobre cada pregunta seleccionada. Para cada pregunta, se pide al modelo que genera una respuesta basada en el contexto, que también se le proporciona. La respuesta generada se muestra por pantalla junto a la pregunta y a la respuesta correcta esperada. La respuesta esperada es comparada con la respuesta generada mediante el uso de la función *comparador()* desarrollada en la sección 3.3.

La similitud de la respuesta actual se guarda en la lista de similitudes. Esta lista contendrá, una vez finalizado el bucle, los valores de las similitudes entre cada respuesta generada ante una pregunta y su respuesta correcta esperada.

```
    similarities = []
    # Iterar sobre las preguntas
    for indice in indices_elegidos:
        contexto = contextos[indice]
        pregunta = preguntas[indice]
        respuesta_correcta = respuestas_correctas[indice]
        chain_of_thought = f"You are going to read the following
context and provide a SHORT answer to a question about
it:\nContext:\n{contexto}\nQuestion:\nPlease, provide a short answer,
do not include explanations"
        prompt = chain_of_thought + pregunta
        respuesta = ollama.chat(model=nombre_modelo,
messages=[{'role': 'user', 'content': prompt}])
        respuesta_string = respuesta['message']['content']

        print(f"\n{indice}. Pregunta:", pregunta)
        print("\tRespuesta: ", respuesta['message']['content'])
        print(f"\tRespuesta esperada:
{indices_respuestas_correctas[indice]}.
{respuestas_correctas[indice]}")
        similitud = comparador.comparador(respuesta_string,
respuesta_correcta)
        similarities.append(similitud)
```

Código 17: Cálculo de la adherencia al contexto en interfaz.py, parte II

Por último, se calcula la adherencia al contexto del modelo como el promedio de las similitudes entre todas las respuestas generadas y esperadas, y se imprime por pantalla.

```
    # Convertir los valores a float
    similarities = [float(value) for value in similarities]
    context_adherence = statistics.mean(similarities)
    print(f"\nAdherencia al contexto del modelo: {context_adherence}")
```

Código 18: Cálculo de la adherencia al contexto en interfaz.py, parte III

3.6. Cálculo de la consistencia

Como ya se ha explicado de forma detallada en la sección 2.3.2, la consistencia mide cuanto se parecen las respuestas que un modelo genera ante una determinada pregunta en diferentes instancias.

Para evaluar la consistencia del modelo. Se pide al usuario que introduzca una pregunta de respuesta corta. La pregunta se pasa al modelo mediante un bucle el número de veces que se considere necesario, guardando todas las respuestas en una lista. Cuantas más respuestas se generen, más preciso será el resultado de la consistencia. Todas las respuestas se imprimen por pantalla.

```
# Consistencia
if(opcion == 4):
    print("Vamos a evaluar la consistencia del modelo. Para ello,
introduzca una pregunta de respuesta corta.")
    pregunta = input("Pregunta: ")
    prompt = "Please, answer in just one sentence. " + pregunta
    # Lista para guardar las respuestas
    respuestas = []
    # Generar 30 respuestas para la misma pregunta
    for i in range(30):
        respuesta = ollama.chat(model=nombre_modelo,
messages=[{'role': 'user', 'content': prompt}])
        respuesta_string = respuesta['message']['content']
        respuestas.append(respuesta_string)
        print(f"{i + 1}. Respuesta: {respuesta_string}")
```

Código 19: Cálculo de la consistencia en interfaz.py, parte I

Un bucle se utiliza para iterar sobre cada respuesta. Un segundo bucle anidado sirve para comparar cada respuesta con las demás respuestas mediante el uso de la función *comparar()* explicada en el apartado 3.3. La condición *i < j* asegura que cada par de respuestas se compare una sola vez, evitando tanto la comparación de una respuesta consigo misma como que existan comparaciones duplicadas. Los resultados de las comparaciones se guardan en una lista.

```
    similitudes = []
    for i in range(30):
        for j in range(30):
            if i < j:
                similitud = comparador.comparador(respuestas[i],
respuestas[j])
                similitudes.append(similitud)
```

Código 20: Cálculo de la consistencia en interfaz.py, parte II

El calculo de la consistencia, mostrada por pantalla como el resultado final, se realiza sumando los valores de todas las similitudes obtenidas en las comparaciones, y dividiendo el total entre el número de comparaciones realizadas para obtener la media.

```
    total = 0
    for similitud in similitudes:
        total = total + float(similitud)
```

```
    consistencia = float(total) / len(similitudes) * 100
    print(f"\nConsistencia: {consistencia:.2f} %")
```

Código 21: Cálculo de la consistencia en interfaz.py, parte III

3.7. Chainpoll mediante agentes

Como ya se ha explicado de forma detallada en la sección 2.3.2, chainpoll es un método para evaluar el contenido en alucinaciones de las respuestas mediante un diálogo entre dos agentes. Los dos agentes constan del modelo evaluado y el modelo evaluador. A este método se le ha añadido una chain-of-thought que especifica cómo debe comportarse el modelo evaluador.

Primero, se solicita al usuario que ingrese el nombre del modelo evaluador, comprobando su existencia mediante la función *check_model_availability()*, explicada en la sección 3.1.

```
# Chainpoll mediante agentes
if (opcion == 5):
    # Verificar la disponibilidad del modelo evaluador
    disponible_evaluador = False
    while disponible_evaluador == False:
        # Pedir al usuario que introduzca el nombre del modelo
        nombre_modelo_evaluador = input("Introduce el nombre del
modelo que tendrá la función de evaluador: ")
        disponible_evaluador =
check_model_availability(nombre_modelo_evaluador)
    print(f"Modelo evaluador: '{nombre_modelo_evaluador}'
seleccionado.")
```

Código 22: Método chainpoll mediante agentes en interfaz.py, parte I

```
Introduce el nombre del modelo a utilizar: zephyr
Modelo 'zephyr' seleccionado.
Opciones de calculo:
    1. Velocidades y duraciones
    2. Correctness
    3. Adherencia al contexto
    4. Consistencia
    5. Chainpoll mediante agentes
    6. RAG analisis mediante agentes
Seleccione una opcion: 5
Introduce el nombre del modelo que tendrá la función de evaluador: llama3
Modelo evaluador: 'llama3' seleccionado.
Vamos a contrastar el modelo mediante el metodo Chainpoll. Para ello, introduzca una pregunta
Pregunta: Who was Tiberius?
```

Ilustración 15: Elección del modelo agente evaluador en la ventana run

Se pide al usuario que introduzca una pregunta para generar una respuesta mediante el modelo evaluado. Después, se utiliza una prompt diseñada para introducir la pregunta y la respuesta del modelo evaluado en el segundo modelo, y así, poder pedirle al segundo modelo que decida si la respuesta del primero contiene alucinaciones o no.

La prompt incluye una plantilla con un ejemplo que sirve como chain-of-thought para que el modelo evaluado responda correctamente. El modelo evaluado devuelve una puntuación entre uno y cinco, representando una mayor puntuación tener menos alucinaciones. La puntuación se muestra por pantalla. Este paso se repite el número de veces que se considere necesario, con varias instancias de la misma pregunta, guardando los resultados en como una suma del valor total. Cuantas más iteraciones, más preciso será el resultado del método.

```
    contraste_acumulado = 0
    for i in range(5):
        respuesta = ollama.chat(model=nombre_modelo,
messages=[{'role': 'user', 'content': pregunta}])
        respuesta_string = respuesta['message']['content']
        print(f"\nRespuesta {i+1}: ", respuesta_string)
        plantilla = "\nFive answers to the question: Who is Don
Quixote?\n\tScore '1' answer: Don Quixote was an 18th-century
astronaut who traveled to the moon on his flying horse named
Rocinante. Accompanied by his faithful robot Sancho Panza, Don Quixote
fought against aliens he mistook for space windmills. He is known for
his epic battles in space and his medieval astronaut suit.\n\tScore
'2' answer: Don Quixote was a knight who lived on a magical island
where windmills were actually disguised dragons. Along with his squire
Sancho Panza, an expert in martial arts, Don Quixote embarked on many
adventures to rescue princesses and search for hidden treasures in
enchanted caves.\n\tScore '3' answer: Don Quixote was a Spanish
nobleman who, after reading many books about knights, lost his sanity
and believed himself to be a knight-errant. He traveled with his
neighbor Sancho Panza, who was actually a disguised wizard. Together,
they faced invisible giants and evil sorcerers in their journeys
across Spain.\n\tScore '4' answer: Don Quixote was a Spanish gentleman
from La Mancha who, after reading too many chivalric novels, decided
to become a knight-errant to revive chivalry and fight against
injustice. Along with his loyal squire Sancho Panza, he lived many
adventures, although there was no real magic or dragons as he
believed.\n\tScore '5' answer: Don Quixote is the protagonist of the
novel 'Don Quijote de la Mancha' written by Miguel de Cervantes
Saavedra. He is a gentleman from La Mancha named Alonso Quijano who,
influenced by the reading of chivalric books, decides to become a
knight-errant under the name Don Quixote. Accompanied by his neighbor
Sancho Panza, who acts as his squire, Don Quixote embarks on various
adventures, many of which are the product of his overactive
imagination and madness. The work is a satire of chivalric romances
and is considered one of the most important works in world
literature.\nThe answers are ordered being the score '5' the most
correct one (the answer would be '5') and the score '1' the least
correct one (the answer would be '1')."
        info_a_contrastar = f"Rate an answer from '1' to '5'. To do
this, follow this template: {plantilla}\nGiven {respuesta_string} as
the answer to the question {pregunta}, evaluate how correct this
answer is."
        contraste = ollama.chat(model=nombre_modelo_evaluador,
messages=[{'role': 'user', 'content': info_a_contrastar}])
        contraste_string = contraste['message']['content']
        contraste_puntuacion =
ollama.chat(model=nombre_modelo_evaluador, messages=[{'role': 'user',
'content': f"RETURN ONLY THE NUMBER '1', '2', '3', '4', or '5'. Find
the score in {contraste_string} and return JUST THE NUMBER ('1', '2',
'3', '4' or '5'), do not print the answer or any additional word, just
the number '1', '2', '3', '4', or '5'."}])
```

```
        contraste_puntuacion_string =
contraste_puntuacion['message']['content']
        for char in contraste_puntuacion_string:
            if char.isdigit():
                contraste_numero = float(char)
        print(f"Contraste {i+1}:", contraste_numero)
        contraste_acumulado += float(contraste_numero)
```

Código 23: Método chainpoll mediante agentes en interfaz.py, parte II

Por último, la suma de todas las puntuaciones se divide entre el número de iteraciones para obtener la media. Esta puntuación es el resultado del método chainpoll adaptado que ha sido programado.

```
    resultado = float(contraste_acumulado)/5
    print("\nEl resultado del metodo Chainpoll es: ", resultado)
```

Código 24: Método chainpoll mediante agentes en interfaz.py, parte III

```
Contraste 1: 5.0
Contraste 2: 5.0
Contraste 3: 5.0
Contraste 4: 3.0
Contraste 5: 3.0

El resultado del metodo Chainpoll es:  4.2
```

Ilustración 16: Ejemplo de resultado del método chainpoll, con solo 5 iteraciones, en la ventana run

Aunque sea un método útil y sencillo en algunos casos, el método chainpoll depende en su totalidad del buen funcionamiento del modelo evaluador. Este funcionamiento en muchos casos se podrá afinar mediante ingeniería de prompts, mejorando la plantilla de chain-of-thought que se entrega al modelo evaluador como ejemplo.

3.8. Evaluación de RAG mediante agentes

Como el método de evaluación de adherencia al contexto tan solo contempla la utilización de datasets que incluyan contextos de tan solo varios párrafos por cada pregunta, no sería útil para evaluar qué tal funciona el modelo en cuanto a consultar documentos largos.

Ante esta problemática, es útil una opción que evalúe la capacidad de un modelo de recuperar y utilizar información de documentos. De esta manera se evalúa el modelo como un RAG completo, que incluye recuperación de documentos y generación de contenido.

El método de evaluación consta de dos agentes y un retriever. Por un lado, el retriever es capaz de seleccionar las partes que son más relevantes de entre varios documentos para responder a una pregunta concreta. Los dos agentes cumplen la función de agente evaluado y agente evaluador. El agente evaluador compara la

respuesta generada por el agente evaluado para poder determinar si la respuesta ha sido generada utilizando esa parte del documento o no.

Como este código va a ser más amplio que las anteriores opciones, ha sido programado en dos archivos diferentes, *retriever.py* y *agentes_RAG.py*. El primer archivo contiene la programación de un recuperador de partes de documentos sencillo. El segundo archivo contiene la función *agentes_RAG()*, que realiza el diálogo evaluador y es llamada desde el archivo *interfaz.py* si se selecciona la opción de evaluar RAG mediante agentes.

```
# Evaluar RAG mediante agentes
if (opcion == 6):
    agentes_RAG.agentes_RAG(nombre_modelo)
```

Código 25: Llamada a la función agentes_RAG() desde interfaz.py

3.8.1. Retriever

Como ya se ha explicado, archivo *retriever.py* contiene la programación de un recuperador de partes de documentos sencillo.

Se realizan las importaciones necesarias, incluyendo las funciones de *comparador.py*. También se cargan el tokenizer y modelo BERT preentrenado al igual que en la sección 3.3.

```
import requests
from comparador import get_embeddings
from comparador import comparador_embeddings
from transformers import BertTokenizer, BertModel

# Cargar el tokenizer y el modelo de BERT preentrenado
tokenizer = BertTokenizer.from_pretrained('sentence-transformers/
model_version')
model = BertModel.from_pretrained('sentence-transformers/
model_version')
```

Los documentos a utilizar deberán ser cambiados dependiendo de la aplicación en la que el modelo trabaje. Si una empresa quiere evaluar un modelo LLM que ha entrenado con ciertos documentos, entonces esos mismos documentos son los que debe utilizar el recuperador para que este método obtenga resultados propicios.

El acceso a los documentos podría ser directo, si la empresa entrega la carpeta con todos los documentos, o a través de una dirección URL. Otras muchas formas de acceso posibles podrían realizarse también, tan solo habría que adaptar el código para que pudiese acceder correctamente a la información.

Como ejemplo ilustrativo, se van a utilizar dos documentos que son dos guiones de películas, a los que se debe acceder a través de sus URL. La pregunta realizada al modelo debe ser sobre algo presente en el guion estas dos películas. Por ello vamos a presuponer que el modelo evaluado ha sido entrenado con estos dos documentos,

aunque en realidad no haya sido así. En una aplicación real, se utilizarían documentos con los que el modelo sí haya sido entrenado, información que no suele ser pública.

```
# URLs y titulos
url1 = 'http://www.imsdb.com/scripts/Citizen-Kane.html'
url2 = 'https://www.imsdb.com/scripts/Strangers-on-a-Train.html'
urls = [url1, url2]
titulo1 = 'CitizenKane'
titulo2 = 'StrangersOnATrain'
titulos = [titulo1, titulo2]
```

Código 26: Documentos que se van a utilizar para la evaluación de RAG, en retriever.py

La primera función que contiene *retriever.py* es para convertir el contenido de los URL en documentos de texto. Recibe como parámetros una lista con los URLs en formato string, y una lista con los nombres que se les quiera poner a los documentos. La función devuelve los nombres de los documentos de texto creados en una lista, para poder acceder a ellos posteriormente.

```
def urls_txts(urls, titulos):
    """
    Esta funcion pasa el contenido de los urls a archivos .txt
    :param urls: Lista con los enlaces url
    :param titulos: Lista con los titulos de los contenidos de los
urls
    :return: nombres_archivos: Lista con los nombres de los archivos
.txt creados
    """
    nombres_archivos = []
    for indice in range(len(urls)):
        # Obtener el contenido del enlace
        response = requests.get(urls[indice])
        # Obtener el texto
        contenido = response.text
        # Definir el nombre del archivo .txt
        nombre_archivo = f'{titulos[indice]}.txt'
        # Guardar el contenido en el archivo .txt
        with open(nombre_archivo, 'w', encoding='utf-8') as file:
            file.write(contenido)
        print(f'El contenido ha sido guardado en {nombre_archivo}')
        # Añadir el nombre del archivo a la lista
        nombres_archivos.append(nombre_archivo)
    return nombres_archivos
```

Código 27: Función urls_txt() en retriever.py

El resto del código es la función *retrieve()* que da nombre al archivo. La función recibe como parámetros la pregunta en formato string, y la lista de nombres de los documentos. Devuelve las partes de los documentos que son útiles para responder a una determinada pregunta. Para ello, primero se divide el contenido de los documentos en partes de tamaño 300 caracteres, aunque el tamaño puede elegirse dependiendo de la aplicación del modelo a evaluar. Las partes se guardan en una lista.

```
def retrieve(pregunta, nombres):
    """
    Funcion que devuelve lo chunks de texto utiles para responder a
una pregunta entre varios documentos.
    :param pregunta: String de la pregunta que se quiere responder.
    :param nombres:  Lista de strings de los nombres de los documentos
```

```
'nombre.txt'
    :return: retireved_chunks: Lista de strings que son los fragmentos
elegidos como mas relevantes para la respuesta.
    """
    chunks = []
    for documento in nombres:
        # Guardar el contenido de cada .txt
        with open(documento, 'r', encoding='utf-8') as archivo:
            while True:
                chunk = archivo.read(300)
                if not chunk:
                    break
                chunks.append(chunk)
```

Código 28: Función retrieve() en retriever.py, parte I

Después, se crean los embeddings de cada parte de documento y de la pregunta, utilizando la función *get_embeddings()*, explicada en la sección 3.3.

```
    embeddings = []
    for chunk in chunks:
        embedding = get_embeddings(chunk)
        embeddings.append(embedding)

    pregunta_embedding = get_embeddings(pregunta)
```

Código 29: Función retrieve() en retriever.py, parte II

Se itera por todos los embeddings de las partes de documento, comparando cada uno con el embedding de la pregunta mediante el uso de la función *comparador_embeddings()*, explicada en la sección 3.3. Los diez valores de similitud superiores quedan guardados en una lista, y sus índices correspondientes en otra para facilitar el acceso a esas partes de documento posteriormente.

```
    indices_mejores_diez = [0] * 10
    valores_mejores_diez = [float(0)] * 10
    for indice in range(len(embeddings)):
        resultado = float(comparador_embeddings(pregunta_embedding,
embeddings[indice]))
        # Verificar si el resultado debe ser añadido a la lista
        if resultado > valores_mejores_diez[-1]:
            # Encontrar la posición donde insertar el nuevo resultado
            for i in range(10):
                if resultado > valores_mejores_diez[i]:
                    # Insertar el resultado y el índice en la posición
correcta
                    valores_mejores_diez.insert(i, resultado)
                    indices_mejores_diez.insert(i, indice)
                    # Mantener las listas con 10 elementos eliminando
el último
                    valores_mejores_diez.pop()
                    indices_mejores_diez.pop()
                    break
```

Código 30: Función retrieve() en retriever.py, parte III

Se imprimen por pantalla las dos listas mencionadas, y se guardan en una lista las partes de los documentos asociados a los índices obtenidos. Esta lista es devuelta por la función, conteniendo las diez partes más relevantes para responder a la pregunta.

```
    # Imprimir los mejores diez índices y sus valores
    print("Mejores diez índices:", indices_mejores_diez)
```

```
    print("Mejores diez valores:", valores_mejores_diez)

    retrieved_chunks = []
    for index in indices_mejores_diez:
        retrieved_chunks.append(chunks[index])

    return retrieved_chunks
```

Código 31: Función retrieve() en retriever.py, parte IV

3.8.2. Diálogo para evaluar la RAG

Como ya se ha explicado, el archivo *agentes_RAG.py* contiene la función *agentes_RAG()*, que realiza el diálogo evaluador entre agentes para evaluar el RAG.

Se realizan las importaciones necesarias, incluyendo la recientemente explicada función *retrieve()* de *retriever.py*, y la función *check_model_availability()* de *interfaz.py*, explicada en el apartado 3.1.

```
import ollama
from retriever import retrieve
import os
from interfaz import chek_model_availability
```

Código 32: Importaciones de agentes_RAG.py

La función *agentes_RAG()* tiene como parámetro el nombre del modelo a evaluar. Primero, se solicita al usuario que ingrese el nombre del modelo evaluador, comprobando su existencia mediante la función *check_model_availability()*, explicada en la sección 3.1. Después, se pide al usuario que introduzca una pregunta. Esta pregunta debe estar relacionada con los documentos que se van a utilizar.

```
def agentes_RAG(nombre_modelo):
    # Verificar la disponibilidad del modelo evaluador
    disponible_evaluador = False
    while disponible_evaluador == False:
        # Pedir al usuario que introduzca el nombre del modelo
        nombre_modelo_evaluador = input("Introduce el nombre del
modelo que tendrá la función de evaluador: ")
        disponible_evaluador =
check_model_availability(nombre_modelo_evaluador)
    print(f"Modelo evaluador: '{nombre_modelo_evaluador}'
seleccionado.")

    print("Vamos a contrastar el RAG del modelo mediante agentes, y
utilizando un Retriever. Para ello, introduzca una pregunta de
respuesta corta.")
    pregunta = input("Pregunta: ")
```

Código 33: Función agentes_RAG() en agentes_RAG.py, parte I

También se pide al usuario que introduzca los nombres de los documentos que se vean a utilizar uno a uno, quedando guardados como strings en una lista. El usuario debe introducir '0' cuando desee terminar de introducir documentos. Si el documento introducido no existe, se avisa al usuario del problema antes de pedir introducir otro documento.

```
    print("Introduzca los documentos que desea consultar. Ingrese 0
cuando haya terminado.")
    # Inicializar una lista para almacenar los documentos
    nombres_documentos = []
    while True:
        try:
            # Solicitar al usuario que ingrese un documento
            nombre_documento = input("Documento a consultar: ")
            # Verificar si el usuario ha ingresado '0'
            if nombre_documento == '0':
                break
            # Comprobar si el documento existe
            if not os.path.exists(nombre_documento):
                raise FileNotFoundError(f"El documento
'{nombre_documento}' no existe.")
            # Agregar el documento a la lista si existe
            nombres_documentos.append(nombre_documento)
        except FileNotFoundError as e:
            print(e)
```

Código 34: Función agentes_RAG() en agentes_RAG.py, parte II

```
Vamos a contrastar el RAG del modelo mediante agentes, y utilizando un Retriever. Para ello, introduzca una pregunta
Pregunta: What is the meaning of rosebud?
Introduzca los documentos que desea consultar. Ingrese 0 cuando haya terminado.
Documento a consultar: DocumentoInexistente.txt
El documento 'DocumentoInexistente.txt' no existe.
Documento a consultar: CitizenKane.txt
Documento a consultar: StrangersOnATrain.txt
Documento a consultar: 0
```

Ilustración 17: Introducción de nombres de documentos en la ventana run

Se pide al modelo evaluado que genere la respuesta. También se crea la lista de partes de los documentos más relevantes utilizando la función *retrieve()* explicada en el apartado 3.8.1.

Iterando sobre las partes de los documentos más relevantes, se pide al modelo evaluador que determine si ese texto ha podido ser utilizado para generar la respuesta del modelo evaluado o no. Devuelve '1' en caso de considerar que sí, y '0' en caso de considerar que no. Los resultados se van sumando hasta que se ha terminado de iterar por todas las partes de documentos más relevantes.

```
    prompt = pregunta + "Answer with one short paragraph."
    respuesta = ollama.chat(model=nombre_modelo, messages=[{'role':
'user', 'content': pregunta}])
    respuesta_string = respuesta['message']['content']
    print(f"\nRespuesta: ", respuesta_string)

    # Retriever
    chunks_relevantes = retrieve(pregunta, nombres_documentos)

    contraste_acumulado = 0
    for chunk in chunks_relevantes:
        prompt = f"\nDoes the following answer include information
retrieved from the following context? Answer just '1' if it does or
'0' if it does not.\nAnswer: {respuesta_string}\nContent:
{chunk}?\nAnswer just '1' if it does or '0' if it does not."
        contraste = ollama.chat(model=nombre_modelo_evaluador,
messages=[{'role': 'user', 'content': prompt}])
```

```
        contraste_string = contraste['message']['content']
        for char in contraste_string:
            if char.isdigit():
                contraste_numero = float(char)
                break
        print(f"Contraste: ", contraste_numero)
        contraste_acumulado += float(contraste_numero)
```

Código 35: Función agentes_RAG() en agentes_RAG.py, parte III

Por último, el resultado del método se calcula como la media, dividiendo la suma total entre el número de partes relevantes. El resultado queda entre '0' y '1', siendo más cercano a '1' si la mayoría de las partes son realmente relevantes o a '0' si la mayoría no son realmente importantes para generar la respuesta.

```
    resultado = float(contraste_acumulado)/len(chunks_relevantes)
    print("\nEl resultado del método es: ", resultado)
```

Código 36: Función agentes_RAG() en agentes_RAG.py, parte IV

```
Mejores diez índices: [99, 632, 193, 296, 610, 630, 84, 297, 468, 624]
Mejores diez valores: [0.1389, 0.1197, 0.1135, 0.1033, 0.0883, 0.0851, 0.0849, 0.0843, 0.0807, 0.079]
Contraste:  1.0
Contraste:  1.0
Contraste:  0.0
Contraste:  0.0
Contraste:  1.0
Contraste:  1.0
Contraste:  1.0
Contraste:  1.0
Contraste:  0.0
Contraste:  1.0

El resultado del método es:  0.7
```

Ilustración 18: Ejemplo de resultado del método de evaluación de RAG mediante agentes, en la ventana run

4. Resultados

En este apartado se presentan los resultados de evaluación de distintos modelos LLM utilizando la interfaz que ha sido programada. Gracias a los resultados, se pueden comparar los distintos modelos, pero también evaluar el funcionamiento de la interfaz.

Para evaluar el correcto funcionamiento de la interfaz que ha sido programada, se pueden evaluar distintos modelos como ejemplo para comprobar que todas las opciones funcionen correctamente. Se procede a la evaluación de los cuatro modelos LLM instalados, lo que va a permitir una comparación entre ellos en cada una de las opciones de evaluación de la interfaz. Los modelos instalados y que se van a evaluar son llama3 [9], zephyr [10], mistral [11], y solar [12].

Se debe tener en cuenta que cada modelo tiene una cantidad de parámetros diferente. Esto significa una cantidad de pesos diferente en la red neuronal del modelo. Un modelo con una mayor cantidad de parámetros tendría mayor capacidad, a no ser que sea tan grande que se sobreajuste a los datos de entrenamiento.

Para la opción de las velocidades se ha optado por realizar la media de los resultados ante 100 preguntas.

Modelo	Parámetros	Velocidad análisis de la prompt	Velocidad generación de la respuesta
Llama3	8 billones	226.42 tokens/s	52.25 tokens/s
Zephyr	7 billones	256.27 tokens/s	55.17 tokens/s
Mistral	7 billones	229.78 tokens/s	55.97 tokens/s
Solar	10.7 billones	18.12 tokens/s	11.50 tokens/s

Tabla 2: Resultados de evaluación de modelos, velocidades de análisis y generación

Zephyr, con un menor número de parámetros en comparación con llama3 y con solar, supera a todos los modelos en velocidad de análisis de la prompt. La velocidad de generación de la respuesta de los tres primeros modelos es muy parecida, siendo bastante inferior la del modelo solar. La velocidad de análisis también es muy inferior en el modelo solar respecto al resto de modelos. Las velocidades tan bajas en el modelo solar se deben a su mayor número de parámetros, lo que significa una mayor complejidad del modelo y por ello la necesidad de un tiempo de procesamiento mayor.

Tanto para el cálculo de la correctnes y de la adherencia al contexto se ha optado también por realizar el cálculo con las respuestas a 100 preguntas. Para el cálculo de la consistencia, se ha optado por realizar 30 iteraciones, lo que implica 435 combinaciones

posibles de dos respuestas a comparar. Se ha utilizado como ejemplo la pregunta: *Who was Tiberius?*

Modelo	Correctness	Adherencia al contexto	Consistencia
Llama3	0.33	0.32	0.51
Zephyr	0.29	0.10	0.46
Mistral	0.40	0.17	0.56
Solar	0.42	0.13	0.60

Tabla 3: Resultados de evaluación de modelos: Correctness, adherencia al contexto, y consistencia

Solar y mistral destacan por ofrecer las respuestas con mayor correctness. Zephyr muestra el nivel más bajo de correctness, lo que sugiere que tiene más dificultades para proporcionar respuestas exactas en comparación con los otros modelos. La consistencia de los modelos es buena en general, sobre todo para el modelo solar, lo que concuerda con la mayor complejidad del modelo.

Llama3 sobresale en adherencia al contexto, con una puntuación significativamente superior a la de los otros modelos. Esto indica que llama3 es el modelo más competente para generar respuestas alineadas con el contexto de la prompt. Los modelos que tienen menor puntuación en esta categoría indican una considerable desconexión entre sus respuestas y el contexto proporcionado. Por otro lado, las puntuaciones tan bajas no se adecuan exactamente a las respuestas que los modelos han sido capaces de dar, en ocasiones considerablemente mejores que lo que la interfaz ha sido capaz de evaluar. Además, este problema no es muy preocupante ya que la opción de evaluador de RAG funciona a su vez como una evaluación de adherencia al contexto más avanzada.

Por último, se utilizan las opciones de evaluación que incluyen un modelo evaluador. Como modelo evaluador se ha elegido llama3 por ser el que ha presentado una mejor adherencia al contexto. Para evaluar el modelo llama3 se ha utilizado el modelo mistral, ya que no se debe utilizar un el mismo modelo evaluado para contratar sus propias respuestas.

Para el método chainpoll se han realizado 30 iteraciones, lo que implica 30 respuestas que se contrastarán utilizando el modelo evaluador. Para la evaluación de RAG se han utilizado las 10 partes de los documentos consideradas como más relevantes por el recuperador. Aunque esta evaluación no es fiable si no se utilizan los documentos con los que el modelo ha sido entrenado, es curioso como el único modelo que ha diferido con una puntuación inferior es el modelo mistral.

Modelo	Chainpoll (sobre 5)	Evaluación del RAG
Llama3	4.10	0.7
Zephyr	4.43	0.7
Mistral	4.30	0.4
Solar	4.47	0.7

Tabla 4: Resultados de evaluación de modelos: Chainpoll y evaluación de RAG

5. Conclusiones

En este capítulo se concluye el proyecto con dos secciones. En la sección 5.1 se explican las complicaciones y los objetivos que se han cumplido. En la sección 5.2 se aportan algunas ideas en las que se podría trabajar en futuros proyectos, a partir de los resultados de este.

5.1. Objetivos cumplidos y dificultades

La interfaz programada permite evaluar modelos de manera eficiente y de forma automatizada, lo que que facilita la comparación entre varios modelos. A través de esta interfaz, los usuarios pueden determinar si un modelo presenta un alto contenido de alucinaciones en las respuestas que genera o no. Para ello, la interfaz permite elegir entre seis métodos de evaluación que se pueden resumir de la siguiente forma:

1. Velocidades medias de análisis de la prompt y de generación de la respuesta.
2. Cálculo de la correctness, que se define como la capacidad para responder a una pregunta zero-shot sin incluir alucinaciones. El cálculo se realiza mediante la comparación con un dataset que incluye preguntas y respuestas posibles.
3. Cálculo de la adherencia al contexto, que se define como la capacidad para responder a una pregunta basada en un contexto sin incluir alucinaciones. El cálculo se realiza mediante la comparación con un dataset que incluye preguntas, contextos y respuestas posibles.
4. Cálculo de la consistencia, que se define como el parecido entre las respuestas que un modelo da ante varias instancias de una misma pregunta.
5. Método chainpoll mediante el diálogo entre dos agentes. Se utiliza un segundo modelo que funciona como agente evaluador para determinar si las respuestas que da el modelo evaluado contienen alucinaciones o no.
6. Evaluación del RAG mediante el diálogo entre dos agentes y un retriever de contenido. Se utiliza el retriever para obtener las partes más relevantes de varios documentos para contestar a una pregunta concreta. Después, el agente evaluador determina cuantas de las partes relevantes ha utilizado el modelo evaluado para generar su respuesta.

La principal dificultad en la creación de la interfaz es que en un ambiente profesional cada cliente puede tener necesidades específicas y únicas. Esta diversidad exige una gran flexibilidad y adaptabilidad de la interfaz. Las posibilidades son infinitas.

La solución ha sido diseñar un código escalable y ajustable, lo que permite adaptarlo a modelos en diferentes proyectos y ambientes. Su diseño modular permite realizar ajustes y mejoras sin necesidad de rediseñar todo el sistema, proporcionando una gran flexibilidad para responder a las necesidades específicas de cada cliente.

El sistema ha sido diseñado con la capacidad de integrar y utilizar los documentos y archivos de cualquier cliente. De esta manera, se utilizarían datasets personalizados para la evaluación de la correctness y de la adherencia al contexto. Por otro lado, los documentos utilizados para entrenar el modelo serían los que se utilizasen por el retriever para la evaluación del RAG.

Otra dificultad ha sido lo novedosas que son la mayoría de las metodologías utilizadas en el proyecto. Existen numerosas investigaciones actualmente sobre el tema, y aunque se publiquen nuevos estudios cada mes, es cierto que no existen tantas fuentes de información como en otros campos tecnológicos que lleven existiendo más tiempo.

5.2. Ideas para futuros proyectos

A continuación, se exponen ideas para posibles futuros proyectos:

a. Utilizar la interfaz programada para evaluar y mejorar un modelo existente.
   Implementar una serie de pruebas utilizando la interfaz y ajustes del modelo de forma iterativa para optimizar los resultados del modelo, asegurando que estos sean más precisos y eficaces.

b. Personalizar la interfaz para evaluar un modelo específico de un cliente con la mayor precisión posible.
   En un ambiente profesional, la interfaz debería ser adaptada y personalizada según los requisitos y necesidades del cliente. Esto incluiría la utilización de datasets personalizados para la evaluación de la correctness y de la adherencia al contexto, y la utilización de los documentos utilizados para entrenar el modelo como la información recuperada para la evaluación del RAG.

c. Continuar investigando nuevas funcionalidades para ampliar la interfaz.
   La mayoría de las metodologías implementadas en la interfaz son novedosas, formando parte actualmente de numerosos proyectos de investigación. Por ello, cada mes se publican nuevos estudios sobre nuevas aplicaciones y métodos, lo que implica grandes posibilidades a la hora de ampliar la interfaz.

6. Concordancia con los Objetivos de Desarrollo Sostenible de la ONU

El proyecto se alinea con los siguientes Objetivos de Desarrollo Sostenible de la ONU propuestos para ser alcanzados antes de 2030: [23]

ODS 4: Educación de Calidad

Meta 4.4: Aumentar el número de jóvenes y adultos que tienen las competencias necesarias para acceder al empleo, el trabajo decente y el emprendimiento

El estudio de la evaluación de las alucinaciones en los LLMs garantiza que todas las personas tengan una nueva vía gratuita y rápida de recibir información precisa y útil. El componente educativo de la IA generativa podrá ser explotado una vez se reduzcan las alucinaciones.

ODS 9: Industria, Innovación e Infraestructura

Meta 9.5: Aumentar la investigación científica y mejorar las capacidades tecnológicas.

Desarrollar una herramienta para evaluar la fiabilidad de los LLMs fomenta la innovación y la investigación en IA, promoviendo la aplicación de tecnologías avanzadas para asegurar la calidad de las aplicaciones basadas en IA.

ODS 16: Paz, Justicia e Instituciones Sólidas

Meta 16.10: Garantizar el acceso público a la información y proteger las libertades fundamentales.

Trabajar para que los LLMs proporcionen información precisa y libre de sesgos apoya el derecho de las personas a acceder a información veraz y de calidad, fomentando una mayor transparencia y confianza en las herramientas de IA.

ODS 17: Alianzas para Lograr los Objetivos

Meta 17.6: Mejorar la cooperación internacional en materia de ciencia, tecnología e innovación.

Desarrollar una interfaz para evaluar las alucinaciones en los modelos LLM puede fomentar la colaboración entre investigadores, desarrolladores, y académicos de distintas partes del mundo.

7. Bibliografía:

[1] Ollama, «ollama.com,» 2024. [En línea]. Available: https://www.ollama.com/. [Último acceso: 11 06 2024].

[2] JetBrains, «JetBrains.com, PyCharm,» [En línea]. Available: https://www.jetbrains.com/es-es/pycharm/. [Último acceso: 11 06 2024].

[3] Python Software Foundation, «python.org,» 2001. [En línea]. Available: https://www.python.org/. [Último acceso: 2024].

[4] NumFOCUS, Inc., «Pandas,» 2024. [En línea]. Available: https://pandas.pydata.org/. [Último acceso: 2024].

[5] Hugging Face, «Transformers, Hugging Face,» [En línea]. Available: https://huggingface.co/docs/transformers/index. [Último acceso: 2024].

[6] The Linux Foundation, «PyTorch,» [En línea]. Available: https://pytorch.org/. [Último acceso: 2024].

[7] NumPy team, «NumPy,» [En línea]. Available: https://numpy.org/. [Último acceso: 2024].

[8] J. Devlin, M.-W. Chang, K. Lee y K. Toutanova, «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,» 2018.

[9] Meta, «Meta, Presentamos Meta Llama 3: modelo de lenguaje a gran escala más potente hasta la fecha,» 18 Abril 2024. [En línea]. Available: https://about.fb.com/ltam/news/2024/04/presentamos-meta-llama-3-el-modelo-de-lenguaje-de-gran-tamano-mas-potente-hasta-la-fecha/. [Último acceso: 2024].

[10] Hugging Face, «Hugging Face, Understanding Zephyr,» 17 Noviembre 2023. [En línea]. Available: https://huggingface.co/blog/Isamu136/understanding-zephyr. [Último acceso: 2024].

[11] Mistral AI., «Mistral AI., Bienvenue to Mistral AI Documentation,» 2024. [En línea]. Available: https://docs.mistral.ai/. [Último acceso: 2024].

[12] Upstage Co., «Upstage AI; Powerful, Purpose-trained LLM, Solar,» 2024. [En línea]. Available: https://www.upstage.ai/solar-llm. [Último acceso: 2024].

[13] J. Dickens, «Learn Prompting,» 2024. [En línea]. Available: https://learnprompting.org/es/docs/basics/configuration_hyperparameters. [Último acceso: 13 06 2024].

[14] Y. Zhu, J. Li, G. Li, Y. Zhao, Z. Yin y H. Mei, «Hot or Cold? Adaptive Temperature Sampling for Code Generation with Large Language Models,» de *Proceedings of the 38th AAAI Conference on Artificial Intelligence*, Vancouver, Canada, 2024.

[15] Galileo, «LLM Hallucination Index: A Ranking & Evaluation Framework For LLM Hallucinations,» Noviembre 2023. [En línea]. Available: https://www.rungalileo.io/hallucinationindex.

[16] R. Friel y A. Sanyal, «ChainPoll: A High Efficacy Method For LLM Hallucination Detection,» 2023.

[17] X. Amatriain, «Measuring and Mitigating Hallucinations in Large Language Models: A Multifaceted Approach,» 2024.

[18] P. Bhavsar, «RAG vs Fine-Tuning vs Both: A Guide For Optimizing LLM Performance, Galileo,» 06 2024. [En línea]. Available: https://www.rungalileo.io/blog/optimizing-llm-performance-rag-vs-finetune-vs-both. [Último acceso: 06 2024].

[19] Elasticsearch B.V., «elastic.co, ¿Qué es la generación aumentada de recuperación (RAG)?,» 2024. [En línea]. Available: https://www.elastic.co/es/what-is/retrieval-augmented-generation. [Último acceso: 2024].

[20] Amazon Web Services, «aws.amazon.com,» 2024. [En línea]. Available: https://aws.amazon.com/es/what-is/retrieval-augmented-generation/#:~:text=With%20RAG%2C%20an%20information%20retrieval,data%20to%20create%20better%20responses.. [Último acceso: 2024].

[21] S. Lin, J. Hilton y O. Evans, «TruthfulQA: Measuring How Models Mimic Human Falsehoods,» 2022.

[22] The Devastator, CC0 1.0 - Public Domain Dedication, «Kaggle, QuAIL (Comprehensive Reading),» [En línea]. Available: https://www.kaggle.com/datasets/thedevastator/introducing-quail-a-comprehensive-reading-compre. [Último acceso: 2024].

[23] ONU, «Objetivos de Desarrollo Sostenible, un.org,» [En línea]. Available: https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/. [Último acceso: 07 2024].

[24] P. Bhavsar, «Mastering RAG: How to Select A Reranking Model,» 21 Marzo 2024. [En línea]. Available: https://www.rungalileo.io/hallucinationindex.