UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

"Sistema para la Extracción de Entidades Médicas a partir de Audio Clínico mediante Procesamiento del Lenguaje Natural"

TRABAJO FIN DE GRADO

JUNIO - 2025

AUTOR: Víctor Navarro Fenoll

DIRECTORES: Fco. Javier Gimeno Blanes

Raúl García Jiménez

RESUMEN

En este Trabajo de Fin de Grado, se ha desarrollado un sistema capaz de extraer la información clínica relevante a partir de transcripciones de consultas médico-paciente. La documentación precisa y eficiente es fundamental para la práctica médica, pero los métodos tradicionales consumen un tiempo valioso. El objetivo principal del proyecto es facilitar y optimizar este proceso, permitiendo a los médicos dedicar mayor tiempo a la atención directa al paciente. Para lograr nuestro cometido, se llevaron a cabo un importante número de pruebas y experimentos con diversas herramientas de transcripción y LLMs disponibles.

Entre las herramientas consideradas para realizar la transcripción, consideramos algunas opciones como Google Speech-to-Text, Whisper o Amazon Transcribe, decantándonos por **Faster Whisper** para la transcripción debido a su gran rendimiento y eficiencia. En cuanto a los LLMs exploramos variedad de opciones y arquitecturas, como BERT, Mistral o Llama, resultando **Llama 3.1-8B** el más adecuado para ser adaptado a nuestro proyecto mediante las técnicas QLoRA y *fine-tuning*. Los resultados obtenidos son prometedores, y sugieren un potencial significativo para mejorar la eficiencia y calidad de la atención médica.

ABSTRACT

In this Thesis, we have developed a system to extract relevant clinical information from transcripts of medical appointments. Accurate and efficient documentation is fundamental to medical practice, but traditional methods consume valuable time. The main objective of the project is to facilitate and optimize this process, allowing doctors to dedicate more time to direct patient care. As part of the research in the development phase, a significant number of tests and experiments were made with diverse avaliable transcription tools and LLMs.

Among all the transcription tools we considered, we evaluated options such as Google Speech-to-Text, Amazon Transcribe or Whisper, ultimately opting for **Faster Whisper** due to its superior performance and efficency. We also explored a variety of LLM's architectures and models, including BERT, Mistral or Llama. Resulting **Llama 3.1-B i**n the most suitable to adapt to our project throught QLoRA and fine-tuning techiques. Results were promising, and suggest significant potential for improving the efficency and quality of medical cara.

INDICE DE FIGURAS

INDICE DE TABLAS

LISTADO DE ACRÓNIMOS

PLN: Procesado del Lenguaje Natural

NLU: Comprensión del Lenguaje Natural (*Natural Language Understanding)*

NLG: Generación de Leguaje Natural (*Natural Laguage Generation)*

LLM: Modelos de Lenguaje Grande (*Large Language Model)*

STT: Texto a Discurso (*Speech-to-Text)*

ASR: Reconocimiento Automático del Habla (*Automatic Speech Recognition)*

WER: Tasa de Error de Palabra (*Word Error Ratio)*

RMS*:* Normalización de Valor Cuadrático Medio *(Root Mean Square Normalization)*

NER: Reconocimiento de entidades nombradas (*Named Entity Recognition*

1 INTRODUCCIÓN

En este primer capítulo, exploraremos los elementos fundamentales de este Trabajo de Fin de Grado. En él, delinearemos la **justificación del proyecto** y la **contextualización del problema** que se aborda, estableceremos los **objetivos** a conseguir y definiremos el **alcance** de las tareas a realizar. Finalmente, presentaremos la **estructura de la memoria** para guiar al lector a través de los subsiguientes contenidos.

1.1 Justificación del Proyecto y Contextualización del Problema

La documentación en las consultas médicas es esencial en la práctica clínica. Un registro preciso y completo de la información es fundamental para garantizar el mejor seguimiento al paciente, la comunicación efectiva entre los distintos profesionales de la salud y la toma de decisiones clínicas. Además, la precisión y calidad de los datos también resultan esenciales para la investigación.

Generalmente, los métodos tradicionales de documentación se basan en la toma de notas manuales durante la consulta, siendo el propio especialista el que rellena todos los datos en papel o en su ordenador. Esto consume una cantidad significativa de tiempo del médico, lo cual reduce el tiempo disponible para la atención directa al paciente. Además, las notas escritas pueden resultar ilegibles, incompletas o propensas a errores; y la interpretación de la información puede variar entre los profesionales.

Grabar y transcribir la consulta completa puede ayudar a la hora de analizar el contenido de la consulta de forma más detallada, pero la búsqueda y análisis de la información en grandes volúmenes de texto no estructurado puede ser lenta y difícil. En este contexto, las tecnologías de procesamiento del lenguaje natural (PLN) y la inteligencia artificial (IA), nos brindan un gran potencial para abordar estos problemas, además de ser un campo en constante evolución y que cuenta con avances recientes en optimización e implementación de modelos.

Este proyecto se centra precisamente en ello; en el desarrollo de un sistema automatizado para la extracción de información clínica relevante a partir de estas transcripciones de consulta médico-paciente. Al automatizar este proceso, buscamos minimizar la carga administrativa de los médicos, permitiéndoles dedicar más tiempo a la atención directa y personalizada al paciente.

1.2 Objetivos del Proyecto

En esta sección, detallaremos los objetivos que pretendemos alcanzar con el desarrollo del presente Trabajo de Fin de Grado. Estos objetivos han guiado el diseño, implementación y evaluación del sistema, y permitirán medir el éxito en función de las metas que seamos capaces de cumplir.

1.2.1 Objetivo General

- Desarrollar e implementar un sistema automatizado para la extracción de información clínica relevante a través de transcripciones grabadas en consulta, con el fin de optimizar la documentación y facilitar la labor de los médicos. En la figura 1 podemos observar el diagrama de flujo de forma simplificada.

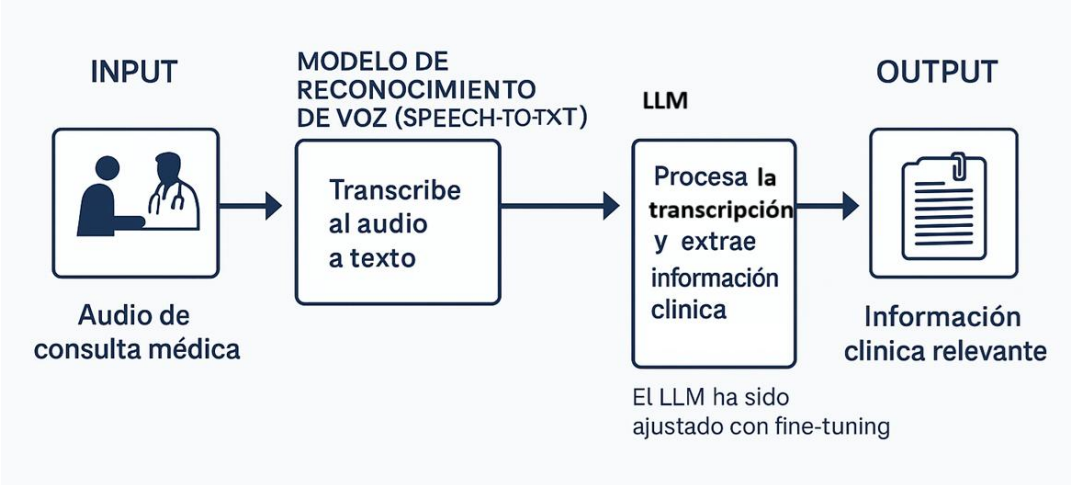

Figura 1. Esquema simplificado de extracción de la información clínica desde el audio de consulta

1.2.2 Objetivos Específicos

- Diseñar e implementar un código capaz de realizar la transcripción de audio de la consulta a texto mediante la tecnología speech-to-text

- Entrenar y adaptar un modelo de lenguaje mediante las técnicas de QLoRA y *fine-tuning* para la identificación, extracción y clasificación de la información clínica.

- Desarrollar un *prompt* efectivo que permita la extracción clínica de forma estructurada.

- Implementar un proceso que estructure la información extraída en formato JSON.

- Evaluar el rendimiento del sistema desarrollado. Tanto en términos de precisión de la transcripción como en la información clínica extraída.

1.3 Alcance del Proyecto

El proyecto contempla el desarrollo de un prototipo funcional del sistema y la posterior realización de pruebas para evaluar el rendimiento, así como la elaboración de una documentación técnica detallada.

Por otro lado, delimitamos el proyecto a la extracción de información, excluyendo los siguientes aspectos:

- No llevaremos a cabo la integración directa con los sistemas de historia clínica electrónica (HCE) existentes
- No entraremos a desarrollar una interfaz de usuario completa, ni la implementación de funcionalidades como la edición o corrección manual de las transcripciones.
- No extraeremos información clínica más allá de síntomas, enfermedades y tratamientos, ni realizaremos pruebas del sistema en un entorno clínico real con pacientes reales.
- Tampoco desarrollaremos ningún tipo de aplicación móvil o web para el acceso al sistema.

Estas exclusiones se deben, en su mayoría, a limitaciones de tiempo, recursos y la complejidad intrínseca de algunas de estas tareas.

1.4 Estructura de la Memoria

Estructuramos la memoria en los siguientes capítulos:

Capítulo 1. Introducción: Donde presentamos una visión general del proyecto; incluyendo la justificación de su realización y la contextualización del problema, los objetivos que pretendemos alcanzar, hasta dónde alcanza el trabajo y la estructura completa de la memoria.

Capítulo 2. Análisis del Estado del Arte: En este capítulo realizamos una revisión de las tecnologías y herramientas existentes que han sido relevantes para el desarrollo del proyecto. Incluimos en este punto los tipos de PLN, las distintas herramientas para la transcripción de audio y la historia y evolución de los LLM.

Capítulo 3. Herramientas: En el capítulo de herramientas valoramos los distintos entornos de ejecución, herramientas STT y LLMs actuales; así como cuáles se adaptan mejor a nuestro proyecto en específico. Estudiamos más en profundidad el funcionamiento específico de Faster Whisper y Llama 3.

Capítulo 4. Diseño e implementación: Describimos el diseño del sistema que hemos creado y detallamos el proceso de implementación de las herramientas, incluyendo los códigos desarrollados y la configuración de parámetros.

Capítulo 5. Conclusiones y Futuras Líneas de Trabajo: Exponemos las conclusiones del proyecto y proponemos posibles líneas de trabajo para futuras investigaciones.

Bibliografía: Se listan las referencias bibliográficas utilizadas a lo largo de la elaboración de la memoria.

Anexo: En el que añadimos algunos códigos que no tengan relación directa con el sistema desarrollado, o que sean muy extensos y no sea tan relevante analizarlos línea a línea.

2 ANÁLISIS DEL ESTADO DEL ARTE

En el presente capítulo establecemos el marco teórico y contextual sobre el que se fundamente este Trabajo de Fin de Grado. Para ello, realizamos un análisis exhaustivo del **estado del arte** del **Procesamiento del Lenguaje Natural**, así como sus principales conceptos y evolución histórica. Posteriormente profundizaremos en la aplicación específica del **PLN en el ámbito clínico**, así como los **modelos PLN más actuales**, haciendo especial incapié a aquellos basados en arquitectras *Deep Learning* que han demostrado unos resultados excepcionales en tareas de comprensión, clasificación y generación del lenguaje.

2.1 Técnicas de Procesamiento del Lenguaje Natural (PLN)

El **Procesamiento del Lenguaje Natural (PLN)**, o Natural Language Processing (NLP) por sus siglas en inglés, es un campo que combina la inteligencia artificial y la lingüística. Su objetivo principal es dotar a los sistemas informáticos de la capacidad de entender, interpretar y procesar el lenguaje humano. Esto facilita enormemente la comunicación entre usuarios y máquinas utilizando lenguaje natural (ver [1]).

El trabajo en PLN comenzó con proyectos como la traducción automática en la década de 1940, aunque por aquel entonces aún no existía el término PLN. La idea era simple pero revolucionaria, ¿podríamos hacer preguntas a un ordenador en “lenguaje humano” y obtener respuesta directamente desde una base de datos? La investigación temprana incluyó sistemas de preguntas y respuestas, además del desarrollo de interfaces de lenguaje natural para bases de datos.

Ha evolucionado desde unos primeros sistemas basados en reglas explícitas, que presentaban limitaciones para cubrir todas las posibilidades y dominios, hasta los métodos estadísticos y de *machine learning*, dando lugar a las técnicas *deep learning* que dominan el campo actualmente.

Figura 2. Funciones básicas del PLN

Hoy en día, las metodologías de PLN permiten procesar texto no estructurado y abordar una amplia gama de tareas lingüísticas y semánticas. En la Figura 2, listamos alguna de las tareas fundamentales: como la medición de similitud, la extracción de temas, el *clustering (*método de aprendizaje no supervisado), la clasificación de texto, el reconocimiento de entidades (NER), la extracción de relaciones y el análisis de sentimiento. De todos éstos, nos centramos en dos de los componentes fundamentales del PLN [2] :

- **Natural Language Understanding (NLU) –** La Comprensión del lenguaje natural se enfoca en permitir que los sistemas “entiendan” el lenguaje humano. Como podemos observar en la Figura 3, el NLU abarca diferentes niveles de análisis del lenguaje [3]:
    - **Nivel Fonológico**: La etapa inicial para el lenguaje hablado. Se encarga de analizar las propiedades acústicas del habla, es decir, la frecuencia, intensidad y duración. Identifica y clasifica los sonidos del lenguaje (fonemas).

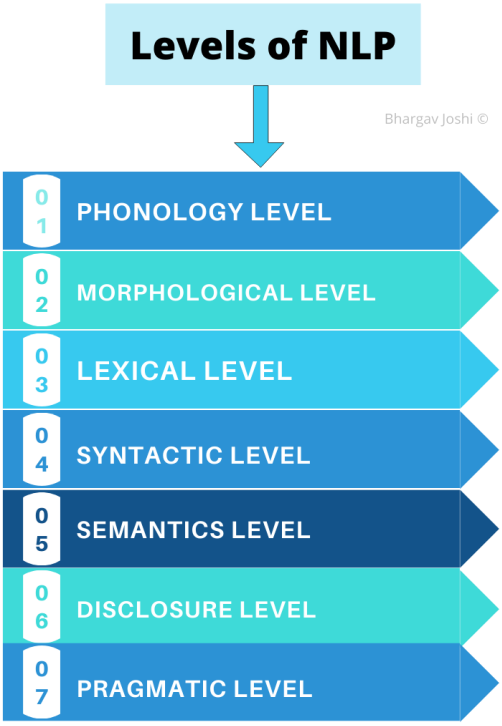

Figura 3. Niveles de Comprensión del Lenguaje Natural

- **Nivel Morfológico y Léxico:** Se enfoca en las palabras individuales, trata de formarlas a través del análisis morfológico. Técnicas como el *Part-of-Speech (PoS) tagging* operan a este nivel, identificando la función gramatical de cada palabra en una oración.
- **Nivel Sintáctico:** Analiza la estructura gramatical de las frases, identificando cómo se relacionan las palabras. La ambigüedad sintáctica, es decir, cuando una frase puede tener múltiples estructuras gramaticales válidas, también suponen un reto a este nivel. El orden de las palabras es vital en este análisis.
- **Nivel Semántico:** Se enfoca en el significado literal de las palabras, frases y oraciones. En este nivel, el PLN busca entender el sentido que se construye con la suma de elementos léxicos y sintácticos. Al igual que en los niveles anteriores, la ambigüedad semántica (es decir, las múltiples interpretaciones que pueda tener una misma frase) supone un reto.

    - **Nivel de Discurso** y **Nivel Pragmático**: El cual analiza el texto más allá de las oraciones individuales, enfocándose en la coherencia y en la cohesión del discurso completo. Permite comprender la estructura del texto e interpretar la relación entre oraciones, además de hacer una interpretación del lenguaje en su contexto de uso (es decir, la intención del hablante).
- **Natural Language Generation (NLG):** A diferencia de NLU, que se centra en la comprensión de un texto, la Generación de Lenguaje Natural es el proceso de generar texto coherente y gramaticalmente correcto a partir de una representación estructurada de datos. Un sistema NLG normalmente abarca tres etapas: **planificación de contenido, planificación de la oración** y **realización del texto**. En [4] se listan modelos de lenguaje modernos como T5, BART y mBART, los cuales son algunos ejemplos de arquitecturas avanzadas capaces de llevar estas tareas de generación de forma eficaz.

La capacidad del PLN para descomponer, comprender y generar lenguaje a través de los diversos niveles de análisis abre un amplio espectro de aplicaciones. No obstante, a pesar de estos logros y la constante innovación, el lenguaje humano presenta una complejidad intrínseca que siguen planteando un gran desafío. Estos retos, como la ambigüedad inherente y cómo varía el contexto, se vuelven aún más críticos cuando el PLN se aplica en ámbitos más especializados y sensibles.

2.2 PLN en el Ámbito Clínico

A pesar de los avances significativos, el PLN sigue enfrentando desafíos inherentes al lenguaje natural. Problemas como la ambigüedad, la dependencia del contexto, la variabilidad del lenguaje (sinónimos, sarcasmo, jerga…) o los errores, se magnifican en dominios especializados como el clínico.

El texto clínico a menudo presenta vocabulario altamente específico, estructura deficiente y una cantidad abundante de abreviaturas. La redundancia en las notas clínicas no solo dificulta la comprensión por parte de los clínicos, sino que también puede afectar el rendimiento de los modelos de PLN.

Los retos principales [5] a los que debe enfrentarse el PLN en la práctica clínica serían:

- La **disponibilidad** y **anotación de datos** clínicos de alta calidad para entrenamiento es limitada, el volumen de datos que se encuentra en línea es escaso, lo cual dificulta los enfoques de aprendizaje automático.
- Muchos sistemas históricos de PLN clínico se basaron en reglas, que no es sencillo escalar o generalizar para diferentes instituciones o especializaciones (cardiología, ginecología, …).
- No existe una solución de PLN "única para todo" en el ámbito clínico; la mayoría de los sistemas se centran en necesidades clínicas muy específicas. Tareas como la extracción de información de la medicación, las características de tumores, historial de tabaquismo, condiciones de elegibilidad para ensayos clínicos, requieren enfoques especializados.
- La **interpretabilidad** de los modelos de PLN es particularmente importante en el contexto de la atención médica para generar confianza y permitir la validación clínica. Ésta es un gran desafío debido a la gran cantidad de términos que podemos encontrar dentro del ámbito clínico, como podemos ver representado en la Figura 4 [6] pudiendo encontrar distintos nombres para un mismo síntoma, prueba o afección (incluidas abreviaciones o siglas)

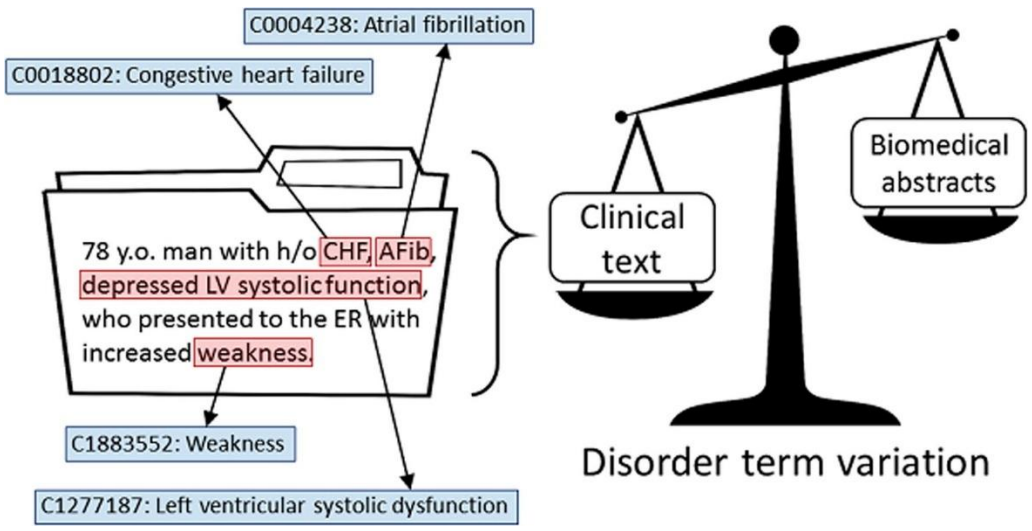

Figura 4. Variación de términos para trastornos

En conclusión, el PLN es un campo en rápida evolución que ha pasado de sistemas basados en reglas a arquitecturas neuronales complejas como Transformers, permitiendo abordar tareas cada vez más sofisticadas. Sin embargo, su aplicación efectiva, especialmente en entornos con alta complejidad como el clínico, requiere la consideración y superación de desafíos relacionados con la naturaleza del lenguaje, la disponibilidad de datos y la necesidad de soluciones adaptadas a contextos específicos.

2.3 Clasificación de Modelos de PLN

El campo del Procesamiento del Lenguaje Natural se encuentra en continua evolución, dando lugar a una amplia diversidad de modelos, cada uno con sus propias características y aplicaciones. Para poder comprender a fondo el panorama actual de PLN y sus implicaciones en distintos ámbitos (como el clínico) sería conveniente categorizar y analizar estos modelos desde diferentes perspectivas.

Esta sección abordará los distintos modelos de PLN en función de la **tecnología empleada** a lo largo de su desarrollo histórico, también los distinguiremos según los **requerimientos de hardware** que demandan para su implementación y, finalmente, según las condiciones de **acceso y licenciamiento** bajo las cuales se distribuyen y usan.

2.3.1 Por tecnología empleada

La evolución tecnológica en PLN ha dado lugar a una diversidad de modelos, cada uno con diferentes enfoques para procesar y comprender el lenguaje humano, marcando una clara progresión histórica en el campo.

Inicialmente, el PLN se apoyó en **modelos basados en reglas explícitas**. Estos sistemas operaban mediante reglas lingüísticas y sintácticas programadas manualmente, utilizando herramientas como expresiones regulares, autómatas y transductores de estados finitos [1]. Aunque capaces de una alta precisión en escenarios muy específicos, tenían la limitación de la enorme complejidad y

variabilidad del lenguaje natural, por lo que era imposible contemplar mediante reglas cada tipo de texto.

Para superar estas limitaciones, surgieron los **modelos estadísticos** y de ***Machine Learning***. Estos enfoques permitieron a los programas inferir patrones directamente a partir de datos. Aprendían de datos anotados, aunque en sus inicios aún requerían de la intervención humana para la anotación y definición de características relevantes. Dentro de esta categoría, se encontraban algunos modelos generativos como Naive Bayes o los Modelos Ocultos de Markov.

La disponibilidad de hardware más potente, como las GPUs, y el aumento masivo de la cantidad de datos disponibles en línea, impulsaron el auge de los **modelos basados en Redes Neuronales Profundas**. Como vemos en [2], esto incluye arquitecturas como las Redes Neuronales Recurrentes (RNNs) y sus variantes como Long Short-Term Memory (LSTM) empleadas en tareas como la generación de texto o extracción de información. Uno de los mayores avances fueron los **mecanismos de atención**, los cuales permitieron a los modelos evaluar la relevancia de diferentes elementos de entrada en una secuencia, combinándolos en un vector de contexto. Esto mejoró notablemente la capacidad de los modelos para manejar secuencias largas y enfocarse en la información más relevante (primeros esbozos de la 'atención', como vemos en [7]).

La arquitectura **Transformer** (revolucionó el PLN al basarse completamente en mecanismos de autoatención, eliminando la necesidad de recurrencia; es decir, no necesita ir palabra a palabra dentro de un texto si no que puede procesarlo todo al mismo tiempo, como vemos representado en la Figura 5 [8]. Más adelante, dentro de la sección de Herramientas, entraremos en detalle sobre cómo funciona la arquitectura *Transformer* y en particular los mecanismos de autoatención; por ahora, es suficiente con entender que estos mecanismos presentan un bloque codificador (izquierda) y un bloque decodificador (derecha) que lee y analiza cada token del input, pero no de forma individual, sino que añade un contexto y una relevancia a cada uno de los tokens que forman la secuencia y trata de leerla al completo, analizando el significado y la relación que tienen las palabras unas con otras.

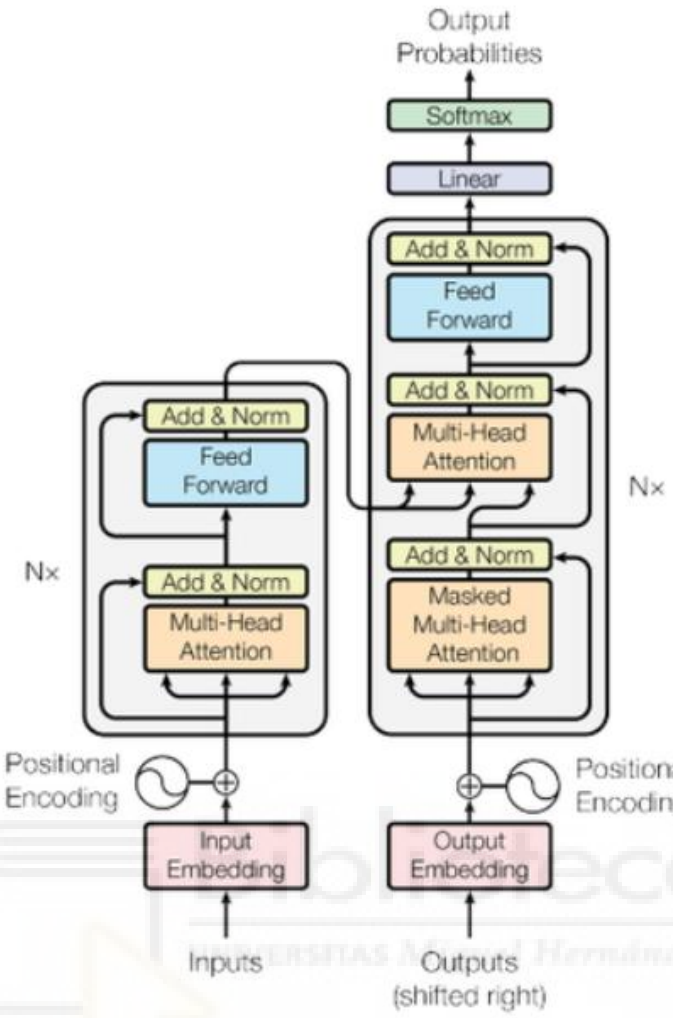

Figura 5. Arquitectura del Modelo Transformer, Codificador-Decodificador

Esto se tradujo en un desarrollo e impulso inmediato en el rendimiento de diversas tareas de PLN. La era actual está dominada por los **Modelos de Lenguaje Grandes (LLMs) y Modelos Pre-entrenados,** también basados en Transformers. Modelos como BERT, GPT, T5, BART y Llama, han sido pre-entrenados en una gran colección de textos y han logrado un rendimiento sin precedentes en una amplia gama de tareas (ver [4]). Estas tareas de aprendizaje implican un pre-entrenamiento masivo seguido de un *fine-tuning* para tareas o dominios específicos.

Para contrarrestar las limitaciones que presentan los LLMs estándar, como la longitud de la secuencia de entrada o los elevados requisitos computacionales, se han desarrollado **modelos optimizados y especializados**. Esto incluye

variantes como Sparse Transformer y Longformer, que utiliza mecanismo de atención eficientes para procesar secuencias más largas, o técnicas de compresión para ser capaces de reducir el tamaño del modelo sin perder rendimiento. También disponemos de la adaptación eficiente para ajustar modelos pre-entrenados con menos recursos.

En resumen, estos modelos, desde reglas simples hasta arquitecturas neuronales complejas y pre-entrenadas, constituyen la base tecnológica del PLN y pueden aplicarse a diversas tareas en dominios específicos.

2.3.2 Requerimientos de Hardware

La implementación y ejecución de los modelos PLN llevan asociados requisitos de hardware que varían drásticamente según la complejidad y tamaño del modelo, algo fundamental para la viabilidad económica y a la accesibilidad.

Los modelos **basados en reglas**, e incluso los de **Machine Learning** tradicional, generalmente presentan bajos requisitos de hardware. Estos pueden ejecutarse eficientemente en CPUs estándar y una infraestructura menos potente.

Por la otra parte, los **LLms** y las **Arquitecturas Transformer de gran tamaño** representan la categoría con los requisitos de hardware más elevados. Su entrenamiento implica altos costes computacionales, demandando hardware dedicado como múltiples GPUs de alto rendimiento, servidores robustos y una gran capacidad de almacenamiento para los modelos y datos. Este escalado hace inviable su acceso a escala global sin una infraestructura considerable. En [9] obtuvimos la Figura 6, una tabla con una aproximación de los requisitos de memoria en la GPU necesarios para ejecutar un modelo en función de su tamaño (en billones de parámetros).

Model Size	FP32 Memory	FP16 Memory	INT8 Memory	4-bit Memory
7B Parameters	~28GB	~14GB	~6GB	~4GB
13B Parameters	~52GB	~26GB	~12GB	~8GB
30B Parameters	~120GB	~60GB	~25GB	~15GB
175B Parameters	~700GB	~350GB	~150GB	~100GB

Figura 6. Requisitos de memoria de los LLMs más comunes

Para contrarrestar los altos costes, la investigación se ha centrado en desarrollar **modelos y técnicas optimizados** para la eficiencia y la reducción de requisitos. Esto incluye el diseño de modelos más compactos y la aplicación de técnicas de compresión como *pruning, quantization y destilación de conocimiento,* que permite reducir el tamaño del modelo manteniendo un rendimiento similar. Además, contamos con las técnicas de adaptación eficiente como el ajuste fino (*fine-tuning)* en datos específicos o *LoRA* (Low-Rank Adaptation), que permiten adaptar un modelo pre-entrenado con significativamente menos recursos (tanto computacionales como de almacenamiento) que un *fine-tuning* completo [4].

En conclusión, la elección del modelo está ligada a la infraestructura de hardware disponible y el presupuesto, siendo una consideración crítica en dominios que manejan grandes volúmenes de datos o requieren despliegue a gran escala.

2.3.3 Por acceso y licenciamiento

La elección de un modelo PLN no solo se limita a las capacidades técnicas, sino que también implica considerar las condiciones de acceso y licenciamiento bajo las que se distribuye. Estas condiciones determinan tanto la flexibilidad y capacidad de modificación del modelo, como los costos asociados a su implementación.

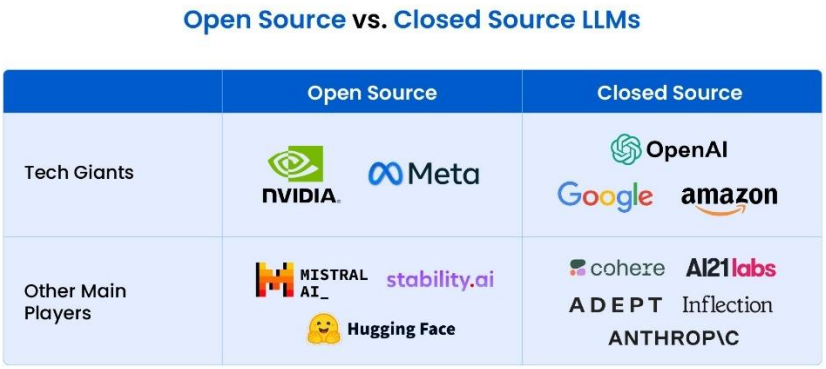

Figura 7. LLMs de código abierto vs código cerrado

Dentro de los LLMs, podemos encontrar modelos de **código cerrado (Closed Source)**, desarrollado y controlados por entidades privadas. Su código fuente es confidencial y su acceso se gestiona a través de APIs de pago o licencias comerciales. Como vemos en la Figura 7 [10], algunos de los ejemplos son muy conocidos, como OpenAI (con ChatGPT) o Google (con Gemini). El uso de estos modelos está estrictamente regido por los términos de servicio del proveedor, lo que implica una dependencia directa en sus condiciones y políticas.

Por otra parte, los modelos de **código abierto (Open Source)**, ofrecen su código fuente y arquitecturas de forma pública, generalmente bajo licencias que permiten su uso sin costes asociados. Esta transparencia y accesibilidad fomenta la colaboración, investigación e innovación, permitiendo a la comunidad inspeccionar, modificar y adaptar los modelos a necesidades específicas. Algunos de los ejemplos más conocidos podemos verlos en la imagen, como el modelo de Meta (Llama) o la librería HuggingFace. Es cierto que, al optar por un modelo de código abierto, generalmente se traslada la responsabilidad de la infraestructura y su gestión al usuario.

En la práctica, la decisión final a la hora de elegir un modelo de código abierto o uno cerrado, implica un balance entre la facilidad de implementación y el control que se desea sobre el modelo. Mientras que los modelos cerrados ofrecen servicios listos para ser utilizados y mayor atención al usuario, los modelos de código abierto proporcionan mayor control y capacidad de adaptarse a tareas en

específico, aunque requieran mayor experiencia técnica y recursos de infraestructura por parte del usuario.

Por otro lado, en el campo de la inteligencia artificial generativa, y más concretamente en los modelos de lenguaje de gran escala (LLMs), es fundamental distinguir entre dos conceptos clave: código abierto y pesos abiertos (open weights, en inglés). El primero se refiere a la publicación del software, bibliotecas y estructuras necesarias para entrenar o desplegar un modelo. No obstante, sin los pesos —es decir, los parámetros numéricos obtenidos tras el proceso de entrenamiento—, este código por sí solo no permite replicar el comportamiento del modelo. Por ello, el término pesos abiertos ha cobrado especial relevancia: implica que los coeficientes entrenados se publican junto con el código, lo que posibilita su reutilización directa y el ajuste fino por parte de desarrolladores e investigadores. Esta práctica, presente en modelos como LLaMA 2 o Mistral, favorece la transparencia, la reproducibilidad científica y la descentralización del conocimiento. Además, constituye un posicionamiento político frente a modelos cerrados que imponen restricciones comerciales o de acceso. En definitiva, avanzar hacia una apertura genuina en los LLMs no solo amplía las posibilidades técnicas, sino que democratiza el acceso a tecnologías fundamentales para el futuro digital.

2.4 LLMs Aplicados a la Medicina

Los Modelos de Lenguaje Grandes (LLMs), como PaLM, GPT-3, GPT-4, BLOOM o LLaMA (ver [11]), están siendo explorados intensivamente por su potencial transformador en el ámbito médico. Estos modelos prometen mejorar la toma de decisiones clínicas y optimizar diversas tareas, desde la extracción y estructuración de información de notas clínicas hasta la redacción asistida de informes o apoyo en tareas administrativas.

Sin embargo, los LLMs entrenados con datos generales carecen inherentemente de una comprensión profunda de la terminología y el conocimiento clínico específico. Para mitigar esta limitación, se ha hecho indispensable su adaptación a través de técnicas de aprendizaje por transferencia. Lo cual incluye tanto al *fine-tuning* como al instruction prompt tuning sobre grandes conjuntos de datos

médicos. Estas adaptaciones son esenciales para mejorar la recuperación de conocimiento relevante, reducir la generación de errores y capturar los matices que presenta el lenguaje médico.

En [7] podemos encontrar algunos modelos especializados como BioBERT, ClinicalBERT o Med-PaLM, diseñados para operar con mayor precisión en el contexto sanitario.

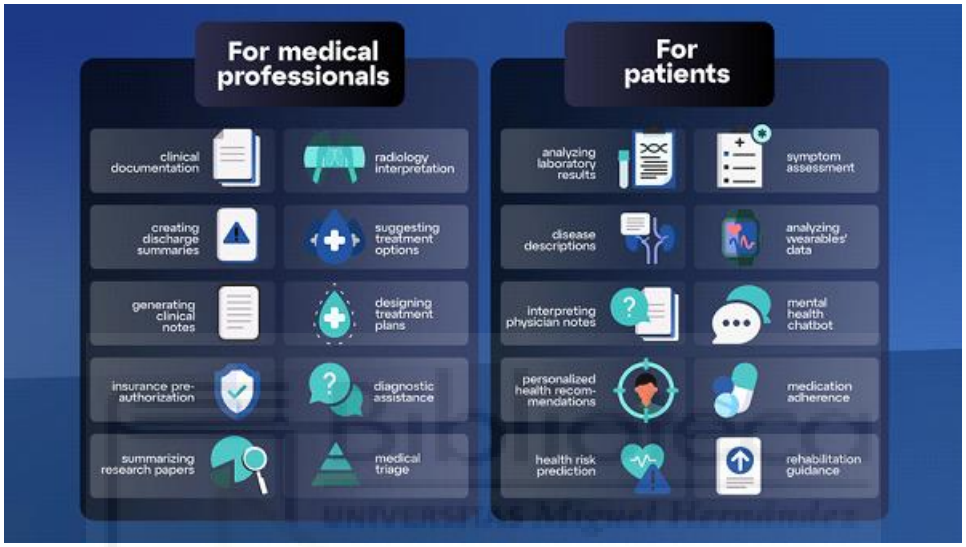

Figura 8. Ejemplo de casos de uso de los LLMs en medicina.

En la Figura 8 vemos algunos posibles casos de uso tanto para los profesionales médicos como para los pacientes [12]. Nos centramos en algunos de los más interesantes:

- **Extracción de la información clínica:** Los LLMs pueden ser usados para recuperar y estructurar datos clínicos a partir de texto libre.
- **Análisis de notas clínicas:** Pueden procesar y extraer información de notas (como las notas de radiología, ortopedia, enfermedades crónicas…).
- **Respuesta a preguntas médicas:** Están siendo evaluados intensivamente para responder preguntas médicas, lo que requiere comprensión lectora, capacidad de recordar el conocimiento médico y manipular conocimiento experto [13].

- **Apoyo administrativo:** Los LLM alivian la carga administrativa clínica, ayudan en la redacción de informes y pueden mejorar los procesos administrativos. En [11] se remarca la importancia de ésta faceta, dada la cantidad de aplicaciones y facilidades que ofrece en el día a día dentro de un hospital.
- **Investigación:** Mejoran la recopilación de datos para la investigación clínica y facilita la identificación de factores de riesgo y los patrones de progresión de enfermedades.

A pesar de estos avances, la implementación de LLMs en entornos clínicos reales requiere una evaluación rigurosa y continua. Es imprescindible garantizar la seguridad, fiabilidad, equidad y ausencia de sesgos antes de poder plantear su despliegue.

2.5 Conversión de voz a texto (Speech-to-Text)

La conversión de voz a texto (*Speech-to-Text,* STT), también conocida como Reconocimiento Automático del Habla (ASR), es una tecnología encargada de transformar el lenguaje hablado en texto escrito. Su importancia es fundamental para la interacción humano-computadora, llegando a ser indispensable para la automatización de la documentación en ciertos campos (ver [14]). Además de para mejorar la accesibilidad a personas con dificultades auditivas.

Históricamente, los sistemas ASR se basaban en los Modelos Ocultos de Markov (HMMs). No obstante, los avances en *deep learning* supusieron una mejora significativa en su precisión.

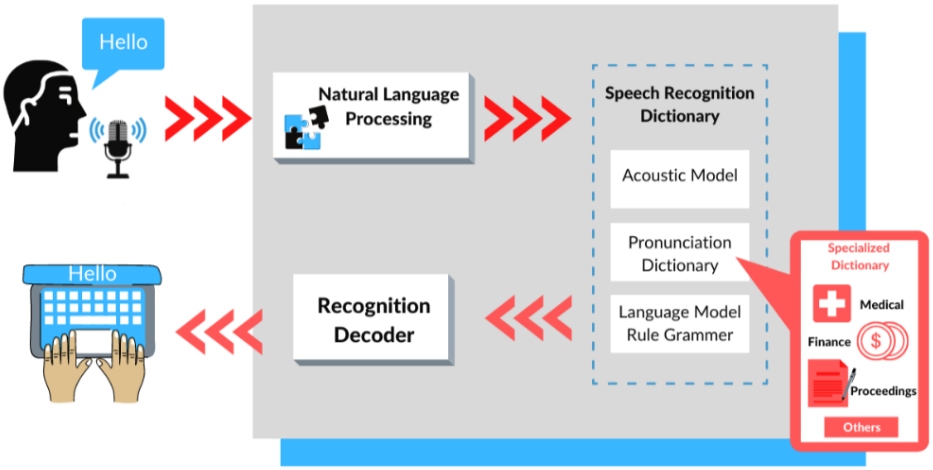

Figura 9. Arquitectura general de un sistema de reconocimiento de voz.

Como vemos en la Figura 9, un sistema ASR típico opera capturando el audio, realizando un preprocesamiento, y después extrayendo las características acústicas del mismo, para finalmente utilizar modelos entrenados para generar la transcripción textual.

Actualmente, las herramientas STT se encuentran en constante evolución. La precisión ha mejorado notablemente, destacando sistemas comerciales como Dragon 15 Professional (vemos en [15] que es considerado la herramienta STT más potente) y, en el ámbito del código abierto, Whisper de OpenAI. Whisper se posiciona recientemente como líder en precisión entre las herramientas de código abierto, y su adaptabilidad a dominios específicos mediante datos sintéticos lo convierte en una opción versátil para aplicaciones como la transcripción en tiempo real. Encontramos usos más complejos (como podemos ver en [16]) como la integración en sistemas de investigación con robots.

Paralelamente, herramientas como Google Cloud Speech API o Amazon Transcribe también se emplean ampliamente, si bien pueden presentar aún algunas limitaciones (como la ausencia de puntuación automática).

A pesar de los avances, la precisión en la transcripción sigue siendo susceptible a diversos factores. La principal fuente de problemas la encontramos en la calidad del habla, influenciada por la fluidez, pronunciación, tartamudeo,

palabras entrecortadas, los distintos acentos y las propias limitaciones del vocabulario para términos complejos, impactan directamente en el rendimiento.

La evaluación de la eficacia en los sistemas STT se realiza mediante métricas estandarizadas como el Word Error Rate (WER), Match Error Rate (MER), Word Information Lost (WIL) y Character Error Rate (CER). Es también habitual comparar el resultado con transcripciones realizadas por humanos para validar su precisión.

Las aplicaciones de los sistemas STT son amplias, abarcando tareas tan diversas como la comunicación humano-máquina, accesibilidad de personas discapacitadas, la documentación clínica en atención médica, apoyo en la educación y sirviendo como base para sistemas híbridos de transcripción y resumen.

3 MATERIALES, MÉTODOS Y EXPERIMENTOS

La implementación y desarrollo de nuestro proyecto de PLN requiere una selección estratégica del conjunto de herramientas, siendo fundamental garantizar la viabilidad del proyecto, optimizar el rendimiento de las operaciones de procesamiento de audio y texto, y asegurar la eficiencia de gestión de los recursos.

En este apartado, detallamos y justificamos las principales herramientas y tecnologías empleadas en este TFG. Comentaremos brevemente el entorno de ejecución y algunas de las librerías fundamentales empleadas, para después centrar toda la atención en los modelos de transcripción de voz a texto y los Modelos de Lenguaje Grandes (LLMs). La selección se ha realizado buscando un equilibrio entre la potencia de procesamiento, la viabilidad de implementación, la eficiencia en la inferencia y la capacidad de adaptación a las particularidades que presenta el lenguaje clínico.

3.1 Entorno de ejecución

Para el desarrollo y ejecución del proyecto era necesario un entorno de desarrollo robusto y flexible. Buscamos la mejor opción para editar código y experimentar con modelos de alta demanda computacional. Además, consideraremos la opción de descargar y ejecutar los modelos de lenguaje grande (LLM) de forma local.

3.1.1 Elección del entorno de ejecución

Para el desarrollo e implementación del proyecto hemos utilizado un entorno de ejecución adaptativo, ajustándonos a las necesidades computacionales de las distintas fases. Inicialmente, el desarrollo lo llevamos a cabo en un entorno local, utilizando **Visual Studio Code (VS Code)** como editor de código principal. VS Code es un editor de código fuente gratuito, de código abierto y multiplataforma desarrollado por Microsoft.

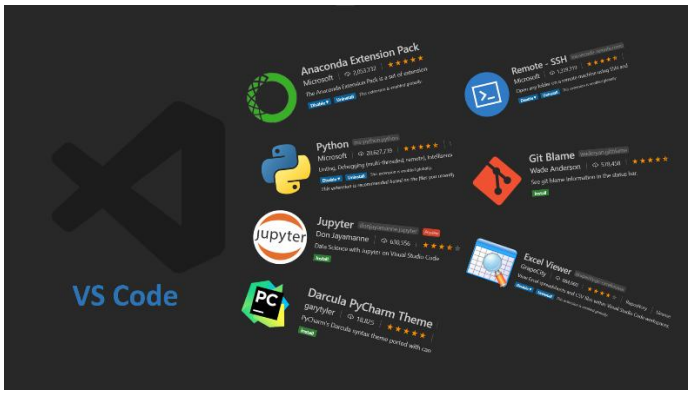

Figura 10. Extensiones de Visual Studio Code

Como vemos en la Figura 10 [17], VS Code destaca ante todo por su versatilidad, ofreciendo entre otras funcionalidades:

- **Integración nativa con Git:** Lo cual nos facilita enormemente el control de versiones de nuestras herramientas.
- **Interfaz de usuario con IA:** Gracias a la IA podremos depurar nuestro código y encontrar soluciones de forma mucho más eficiente.
- **Gestión de entornos virtuales:** Completamente integrado con Anaconda, permite gestionar varios entornos de desarrollo de forma fluida, lo cual nos facilita el aislamiento de cada modelo que hemos probado en este proyecto.
- **Variedad de extensiones:** Permite personalizar y expandir sus funciones, pudiendo instalar extensiones para desarrollo en Python, integración con Jupyter Notebooks, depuración y visualización de datos.

Complementando a VS Code, se empleó **Anaconda** para la gestión de entornos Python y librerías. Anaconda permite crear entornos virtuales aislados, lo que evitará conflictos entre paquetes y nos facilitará replicar el entorno de desarrollo.

Durante la fase de transcripción, comprobamos que los modelos se ejecutaban de manera eficiente en el entorno local, permitiendo un procesamiento adecuado de los archivos de audio y generando el texto en un tiempo razonable. Sin embargo, al abordar el procesado de LLMs, se hizo evidente la limitación de recursos del sistema local. La inferencia y, particularmente, el *fine-tuning* de estos modelos demandan una elevada cantidad de memoria GPU que el

hardware local no disponía. Fuimos capaces de integrar y ejecutar el sistema completo de forma local haciendo uso de CUDA, Ollama, y repartiendo la carga computacional entre GPU y CPU. Solventamos el problema para la carga e inferencia de los LLMs preentrenados, pero los entrenamientos con *fine-tuning* se alargaban durante horas, dificultando la realización de experimetnos.

Por esta razón, se migró la ejecución de las tareas más intensas computacionalmente a **Google Colaboratory (Google Colab).** Colab es un servicio en la nube que proporciona un entorno de Jupyter Notebook preconfigurado y permite la ejecución de código Python directamente desde el navegador web.

Su principal ventaja y el motivo de su adopción en este proyecto fue la posibilidad de acceso a GPUs dedicadas (tanto en la versión gratuita como la de pago). Google Colab resultó ser la solución idónea para ahorrar tiempo en la inferencia y para poder realizar el *fine-tuning* con distintos datos y parámetros de entrenamiento.

3.1.2 Ollama

Para poder gestionar y ejecutar nuestros Modelos de Lenguaje Grandes (LLMs) de forma eficiente en el **entorno local**, hemos integrado **Ollama** en el flujo de trabajo del proyecto. Ollama (Figura 11) es una plataforma de código abierto que facilita la descarga, despliegue y gestión de LLMs en máquinas locales [18].

Figura 11. Ejecución de modelos de forma local mediante Ollama

Su ventaja principal la encontramos en su simplicidad de uso y aplicación, permitiendo ejecutar modelos de gran tamaño con una configuración mínima y aprovechando el hardware disponible (GPU y CPU) de forma optimizada. Además de esto, cabe también mencionar algunos de sus puntos fuertes:

- **Privacidad y Seguridad:** En el contexto de nuestro TFG, en el que procesamos una transcripción de consulta médico-paciente, la privacidad de la información es crucial. Ollama permite ejecutar los LLMs de forma completamente local, asegurando que los datos sensibles no abandonen el entorno del sistema y se puedan procesar de forma segura.
- **Latencia reducida:** Al no depender de servidores externos, la inferencia de los LLMs ejecutados a través de Ollama suele presentar menor latencia.
- **Independencia y Autonomía:** Al ejecutar los modelos localmente y no depender de APIs externas o servicios de terceros, tendremos mayor control sobre el proceso de inferencia.
- **Flexibilidad:** Ollama facilita la descarga e intercambio entre diferentes modelos y versiones de LLMs, lo cual nos ha permitido experimentar con distintas arquitecturas y tamaños de modelos para encontrar la configuración óptima sin necesidad de configurar el entorno para cada modelo.

Gracias a estar características, Ollama se posiciona como una herramienta fundamental para nuestro desarrollo y experimentación de LLMs.

3.2 Herramientas de Speech-to-Text

El procesamiento de consultas médico-paciente en formato de audio, que constituye la base de nuestro proyecto, requiere una fase inicial de conversión del lenguaje hablado a texto escrito. Tal como hemos presentado previamente, haremos uso de las tecnologías *Speech-to-Text* (STT), también conocidas como Reconocimiento Automático del Habla (ASR).

Las tecnologías STT nos harán de puente entre la conversación en consulta y nuestro sistema, convirtiendo las señales acústicas de voz en una secuencia de

palabras comprensible por la máquina, permitiendo el posterior análisis mediante las técnicas de PLN.

Dada la propia naturaleza del proyecto, la precisión, robustez y eficiencia del modelo STT seleccionado serán factores determinantes para la calidad global del sistema.

3.2.1 Modelos STT considerados

La selección de un modelo Speech-to-Text (STT) adecuado es un paso crítico para asegurar el funcionamiento del sistema. Para esta investigación, se consideraron diferentes modelos y enfoques. Como podemos ver en la Figura 12, tratamos de incorporar al proyecto modelos de distintas generaciones y metodologías dentro del campo del ASR.

Figura 12.Diferentes modelos STT explorados

Se comenzó con una revisión de modelos de código abierto y plataformas establecidas que, aunque funcionales, requerían de una configuración y entrenamiento considerable para ser usados en nuestro proyecto para ser efectivos:

- **CMU Sphinx:** Se trata de una toolkit de ASR de código abierto. Es una de las más antiguas, desarrollada por la Carnegie Mellon University. Ofrece

flexibilidad y control, pero la implementación a distintos idiomas y acentos es compleja y requiere mucho tiempo y recursos.

- **Kaldi:** Parecido a CMU Sphinx, también se trata de un toolkit ASR de código abierto muy popular y potente. Su principal ventaja reside en que es altamente configurable, pero tiene una gran curva de aprendizaje y requiere conocimientos profundos de fonética y acústica.
- **DeepSpeech:** Se trata de un motor de código abierto desarrollado por mozilla, el cual está basado en redes neuronales profundas. Aunque es más sencillo de usar que Kaldi, su desarrollo y soporte se han reducido en los últimos años. Además, el rendimiento puede variar según el idioma.
- **AWS (Amazon Web Services):** Un servicio de STT en la nube ofrecido por Amazon. Gran facilidad de uso y es capaz de procesar audio en diversos idiomas. Tiene una gran precisión, pero al ser un servicio en la nube no podemos utilizar el servicio de forma local, además de llevar implicados costes de uso.
- **DeepGram:** Al igual que AWS, se trata de una plataforma de ASR basada en la nube. Cuenta con alta precisión y velocidad, incluso en condiciones de ruidos o con lenguaje específico. Se distingue por sus modelos de *deep learning*, que prometen una menor tasa de error de palabra (WER) y baja latencia. Al igual que otros servicios en la nube, usarlo implica costes y posibles problemas de privacidad por la gestión y manejo de los datos.

Posteriormente, centramos la atención en modelos y servicios más recientes y avanzado que nos dieron un rendimiento superior, especialmente aquellos que incorporan arquitecturas Transformers:

- **Wav2Vec:** Una familia de modeloos de META AI basado en aprendizaje autosupervisado. A diferencia de los modelos tradicionales que requieren transcripciones anotadas, Wav2Vec aprende directamente de los patrones lingüísticos y acústicos del habla. Una vez pre-entrenados, estos modelos pueden ser *fine-tuned* con una cantidad relativamente pequeña de datos etiquetados con muy buen rendimiento. Su capacidad para

aprovechar datos no anotados ha sido fundamental para avanzar en el ASR en diversos idiomas.

- **Google Speech-to-Text:** Se trata de un servicio comercial robusto de ASR ofrecido como parte de Google Cloud. Es ampliamente reconocido y utilizado por su alta precisión y fiabilidad, resultado de años de investigación y entrenamiento en grandes cantidades de audio. Proporciona transcripciones en tiempo real y por lotes en más de 125 idiomas y variantes, además de incluir funcionalidades avanzadas como identificar a los interlocutores y la capacidad de adaptarse mediante modelos personalizados. Su solidez lo convierte en una opción muy potente para aplicaciones comerciales que priorizan la precisión y la escalabilidad, por ello fue la primera opción al plantear este proyecto. Por otra parte, se trata de una opción en línea con coste por uso, por lo que además de suponer un sobrecoste, tampoco podríamos ejecutar el modelo de forma local. Esto implica que podríamos tener problemas con las políticas de privacidad para el manejo de datos sensibles.
- **Whisper:** Modelo lanzado por OpenAI, Whisper es un modelo de ASR quqe ha establecido un nuevo estándar en el campo. Se distingue por sus funciones multi-tarea y por ser un modelo multilingüe, entrenado en más de 680.000 horas de audio y texto de la web, que abarca una gran diversidad de idiomas y condiciones del audio. Su arquitectura está basada en Transformers, y no solo le permite transcribir voz, sino también traducir y clasificar el idioma, logrando una robustez y precisión excepcionales incluso con ruido o acentos variados. Esta versatilidad lo ha convertido en una herramienta muy popular para las aplicaciones de transcripción.
- **Faster Whisper:** Se trata de una reimplementación optimizada del modelo Whisper de OpenAI. Desarrollado para mejorar la eficiencia, Faster Whisper utiliza la librería CTranslate2 para lograr una inferencia significativamente más rápida. Además, hace un uso mucho más eficiente de la memoria (CPU y GPU) que el modelo original de Whisper. Esta optimización resulta vital para proyectos que necesitan procesar grandes volúmenes de audio o que requieren transcripciones de audio en tiempo

casi real, sin perder calidad y con la robustez de transcripción que caracteriza a Whisper.

Esta evaluación de los diferentes enfoques de STT resultó fundamental para determinar la solución más adecuada en términos de precisión y eficiencia para nuestra transcripción de consulta médico-paciente. En el siguiente punto, nos centraremos en la comparación de los modelos más prometedores.

3.2.2 Comparación de modelos STT

Para determinar el modelo Speech-to-Text (STT) más adecuado para la transcripción de las consultas, llevamos a cabo una evaluación comparativa entre los modelos que consideramos más prometedores del punto anterior: Wav2Vec, Google Speech, Whisper y Faster Whisper. Centramos la comparación en dos medidas clave: la **calidad** de la transcripción y el **tiempo** de procesamiento (velocidad de inferencia), los cuales son críticos para la eficiencia y precisión del sistema en un entorno real.

Para realizar la comparación hemos utilizado diferentes audios, la gran mayoría con lenguaje del ámbito clínico, con distinta duración, calidad del audio y número de hablantes. Para evaluar su rendimiento, llevaremos a cabo:

1. **Medición del tiempo:** El tiempo de procesamiento lo medimos como el tiempo total que el modelo tarda en transcribir el archivo de audio .wav cargado. Para ello utilizamos la librería de Python *timeit* (7.1).

2. **Medición de calidad:** Realizaremos la trascripción manual de los audios y los compararemos con la respuesta generada por el modelo. Para el cálculo del WER necesitaremos el número de inserciones, sustituciones y eliminaciones entre el texto original y la transcripción. Haciendo uso de una IA externa, calcularemos un WER aproximado y haremos la media con todas las transcripciones.

3. **Configuración de los modelos:**

- **Wav2Vec**: Utilizamos la implementación pre-entrenada para español. Para ello descargamos de Hugging Face el modelo *jonatasgrosman/wav2vec2-large-xlsr-53-spanish*.
- **Google Speech-to-Text:** A través del servicio en la nube, pudimos crear una clave de autorización e integrarlo en un Script de Python.
- **Whisper:** Hicimos las pruebas con el modelo medium, versión que ofrece el mejor balance entre tamaño y rendimiento.
- **Faster Whisper:** También utilizamos su versión medium para hacer una comparación directa con Whisper y comprobar si se aprecia la optimización de velocidad.

Resultados:

Aunque en las pruebas hemos añadido variedad de audios, incluyendo algunos con mala calidad de sonido e incluso algunos de más de 30 minutos de duración; para representar la Tabla 1, y tratando que el resultado sea lo más imparcial posible, hemos seleccionado un conjunto de 10 audios con duración inferior a 3 minutos y con abundante lenguaje médico.

MODELO STT	WER APROXIMADO (%)	Tiempo de Procesamiento (s)	Notas/Observaciones
Wav2Vec	**39.20%**	**28.17 s**	WER Y tiempos de procesamientos muy mejorables
Google Speech	**18.32%**	**12.90 s**	Muy buen WER y tiempo de procesamiento. Sin signos de puntuación.
Whisper	**29.91%**	**16.076 s**	Buen tiempo de procesamiento, WER mejorable
Faster Whisper	**16.80%**	**8.30 s**	Muy buen tiempo de procesamiento y WER

Tabla 1. Comparación de modelos STT

Análisis de Resultados:

- **Wav2Vec** presentó buen rendimiento en las pruebas con audios cortos y buena calidad de sonido, pero al introducir audios más largos y lenguaje del ámbito clínico su WER se ha descontrolado. Necesitaría un fine-tuning específico para el dominio médico.
- **Google Speech-to-Text** nos da muy buen tiempo de procesamiento y la transcripción apenas presenta errores. La transcripción no tiene signos de puntuación, aunque éstos no serían relevantes para nuestro proyecto.
- **Whisper** tiene buen tiempo de procesamiento, pero presenta una transcripción con errores en bastantes palabras. Además, omite ciertas partes del audio cuando se superponen las voces.
- **Faster Whisper** nos da muy buen rendimiento. El tiempo de procesamiento se ve muy recortado al hacer uso de CUDA, y la transcripción apenas presenta errores.

Como síntesis, la comparación entre los modelos de Speech-to-Text demostró que los servicios en la nube, como Google Speech-to-Text, si bien ofrecen una gran precisión, no terminan de compaginar con los requisitos de privacidad para el manejo de datos clínicos sensibles. Por otra parte, los modelos de código abierto Wav2Vec y Whisper eran prometedores, pero mostraron limitaciones en la precisión para capturar el lenguaje clínico. Tras estas comparativas, **Faster Whisper** se presenta como la opción más viable, al presentar el mejor equilibrio entre una alta calidad de transcripción y una eficiencia de procesamiento.

3.2.3 Elección de modelo STT: Faster Whisper

Basándonos en los resultados de las pruebas realizadas en el apartado anterior, seleccionamos **Faster Whisper** como el modelo de Speech-to-Text (STT) principal para este proyecto. Esta decisión la fundamentamos, además de en el bajo WER y tiempo de procesamiento, en su facilidad de implementación y la cantidad de horas de entrenamiento que tiene el modelo base.

Faster Whisper es una reimplementación del modelo Whisper de OpenAI. Optimiza el rendimiento gracias a CTranslate2, un motor de inferencia rápido

para modelos de Transformer. Podemos ver la mejora de Faster Whisper respecto a las versiones anteriores en la Tabla 2. Esta optimización permite que sea hasta 4 veces más rápido que el modelo Whisper original, manteniendo la misma precisión y utilizando menos memoria. Además, permite mejorar la eficiencia mediante la cuantización a 8 bits tanto en CPU como en GPU.

Estas mejoras en velocidad y eficiencia, las cuales no comprometen la calidad de la transcripción, fueron clave para su elección; presentando un rendimiento muy similar a herramientas de pago por suscripción, resulta una opción mucho más interesante teniendo en cuenta la cantidad de grabaciones que debemos procesar.

	latency (m)	vram (GB)	wer (%)	cer (%)
openai whisper	10.8	9.7	10.9	4.3
Transformers (fp32, B=1)	6.9	7.6	16.2	10
Transformers (fp32, B=8)	4.6	11	16.2	10
Transformers (fp16, B=1)	6.6	3.6	16.2	10
Transformers (fp16, B=8)	2.14	5.7	16.2	10
Transformers (fp16, B=16)	2.04	8.2	16.2	10
Transformers (fp16, B=32)	2	13.2	16.2	10
FasterWhisper (fp16)	2.58	**3.5**	11.98	4.7

Tabla 2. Comparación entre las distintas versiones de Whisper

Dentro de las diferentes variantes que ofrece Faster Whisper (tiny, small, medium, large), optamos por el **modelo medium**. En el punto 4.5 veremos más detenidamente las diferencias entre versiones, en las cuales la transcripción varía notablemente (tanto la calidad como el tiempo de procesamiento). La respuesta del modelo *medium* frente a variaciones en la calidad del audio y la presencia de terminología médica fue realmente buena, lo cual nos asegura transcripciones fiables para el siguiente paso dentro del PLN.

Ahora bien, ¿**cómo funciona** exactamente Faster Whisper? Al tratarse de una reimplementación optimizada de Whisper, debemos primero analizar y entender la arquitectura de entrenamiento de su predecesor.

Whisper se entrena como un sistema multitarea de gran escala, lo que le permite aprender a realizar diversas tareas relacionadas con el procesamiento del habla simultáneamente. Como podemos ver en la Figura 13 [19], el conjunto se entrena con un conjunto de datos de 680.000 horas de audio, permitiendo:

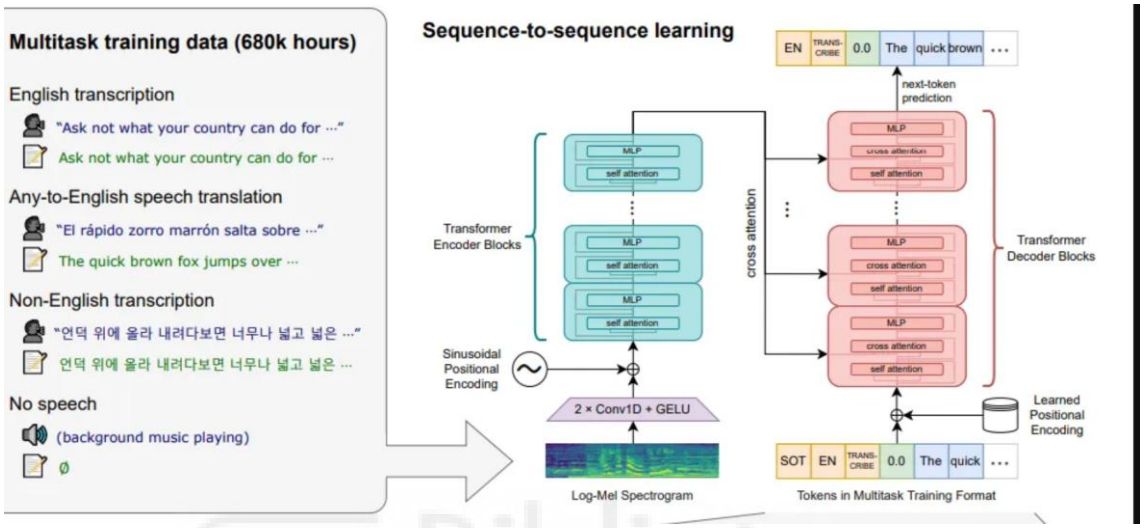

Figura 13. Entrenamiento multitarea y aprendizaje secuencia a secuencia en Whisper

- **Transcripción en inglés:** Audio en inglés para obtener una transcripción textual en inglés.
- **Traducción de cualquier idioma al inglés:** Captura audio en cualquier idioma y lo traduce automáticamente al inglés.
- **Transcripción en idiomas no ingleses:** Audio en diversos idiomas con su transcripción textual en el idioma original.
- **Fragmentos sin habla:** Segmentos el audio con ruido de fondo o música, sin voz, que permite al modelo identificar momentos en lo que no se está hablando y no requiere transcripción. Lo cual ha resultado uno de los mayores éxitos del modelo, puesto que los audios dentro de consulta tienen largos momentos de silencio, con ruidos y voces de fondo que no queremos que el modelo transcriba. Es de los pocos modelos que de verdad ha sido capaz de detectarlos e ignorarlos convenientemente.

Este enfoque multitarea confiere a Whisper una notable versatilidad, permitiendo no solo transcribir audio, sino también **identificar** el idioma y **traducirlo** directamente.

La arquitectura central de Whisper, y por extensión de Faster Whisper, se basa en una arquitectura Transformer de tipo **encoder-decoder,** como podemos ver en la otra parte de la Figura 13. Este diseño, común en los modelos de secuencia a secuencia, podemos desglosarlo en dos componentes principales:

- **Encoder (Bloques del codificador Transformer):** El diagrama de bloques azul que vemos en la figura, encargado de procesar el audio de entrada. Primero, convierte el audio en un espectrograma Mel (transformación basada en Wavelets en lugar de Fourier), la cual se trata de una representación frecuencial del sonido más estable frente a las variaciones del habla. Luego, añadimos un **Sinusoidal Positional Encoding,** el cual básicamente añade información sobre la posición temporal de los segmentos de audio. A continuación, pasamos por una serie de bloques de Transformer. Observamos que cada bloque contiene capas de **Multi-Layer Perceptron (MLP)** y **Self-Attention.** Las **MLP** son redes neuronales densas que aplican transformaciones no lineales, mientras el mecanismo de **Self-Attention** permite al modelo valorar la importancia de diferentes partes de la secuencia de entrada con respecto a otras, **lo** que permite al modelo capturar dependencias a largo plazo y características relevantes del audio. La capa de *self-attention* será explicada más en profundidad dentro del apartado de modelos LLM. La salida del encoder será una representación contextualizada del audio de entrada.
- **Decoder (Bloques del decodificador Transformer):** El decoder toma la representación del encoder y genera la secuencia de texto de salida. Utiliza **Learned Positional Encoding** para los tokens de texto. Los bloques del decoder también contienen capas de MLP y Self-Attention y, además, podemos observar cómo incorpora también una capa de **Cross-Atention.** Esta capa permite al decoder relacionar los tokens que está generando con las características relevantes que, dentro del encoder, habíamos localizado dentro del audio de entrada. El proceso de

decodificación es iterativo: en cada paso, el modelo predice cuál podría ser el siguiente token de la secuencia (hasta generar un token de fin de secuencia, EOT).

El formato de entrenamiento de Whisper es una de sus características más innovadoras, permitiendo al modelo realizar una amplia gama de tareas. Al inicio del bloque del decodificador en la anterior Figura 13, podemos ver cómo inyectamos los llamados 'Tokens con formato de entrenamiento multitarea'. Este tipo de entrenamiento que reciben los tokens lo vemos con más detenimiento en la Figura 14 [19]:

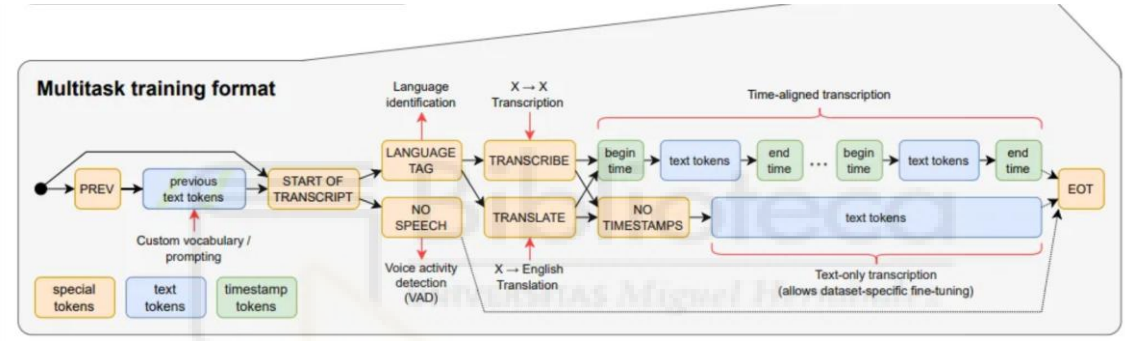

Figura 14. Formato del entrenamiento multitarea empleado en Whisper

Antes de empezar la transcripción, vemos cómo se insertan tres elementos:

- **Tokens especiales (*Special Tokens)*:** Los cuales actúan como *prompts* o instrucciones para el modelo, los cuales le indican las tareas específicas que deben realizar. Los tokens específicos para cada idioma como, por ejemplo, 'En' y 'Es' (inglés y español), son fundamentales. Estos tokens son aprendidos por el modelo durante el entrenamiento y se utilizan en la inferencia para "controlar" su comportamiento.
- **Token de texto (Text Tokens):** También llamados 'Tokens de texto **previo'**, los cuales permiten al modelo mantener el contexto de la conversación anterior. Esto resulta fundamental para mantener la coherencia en transcripciones largas o en diálogos. Al alimentar al decoder con el texto que ya ha generado, el modelo puede utilizar esta

información para predecir el siguiente token de la secuencia de forma más precisa. Esta idea es similar a cómo los LLMs utilizan prompts para generar texto coherente.

- **Tokens de marca de tiempo (Timestamp Tokens):** Los cuales nos generan marcas, las cuales resultan muy útiles para indicar el principio o final del audio, así como para realizar tareas de sincronización de texto y audio.

Tras esto podemos observar las distintas funciones del modelo, las cuales varían según los tokens especiales introducidos:

- **Inicio de la transcripción:** Indica, como su propio nombre indica, el inicio de la secuencia de transcripción o traducción. Suele ir precedida por los tokens que definen el idioma y la tarea.
- **Sin Habla (No Speech):** Utilizado junto con herramientas de detección de actividad de voz (VAD), es capaz de analizar la presencia o ausencia de voz humana. Indispensable para tareas de diarización, codificación y reconocimiento de voz. Genera un token '<*no_speech*>'. Este token especial se sustituye en el lugar de cualquier texto, permitiendo al sistema omitir la transcripción de esos segmentos.
- **Identificación de idioma (Language Identification):** En el caso de encontrar señal de voz, lo primero que debemos hacer es identificar el idioma del audio. Idealmente, nosotros mismos añadiremos en nuestro código una variable que incluya el token de idioma especificado, lo cual daría un resultado mucho más estable y evitando malas interpretaciones del ruido o el habla. Una vez completado este paso, encontramos dos posibilidades:
- **Transcripción del audio:** Transcribe el audio en el mismo idioma de origen detectado. Para transcribir el audio en el mismo idioma de origen (por ejemplo, español a español), se introduce el token de idioma correspondiente seguido del token de inicio (<*startoftranscript*>), entonces el decoder genera la secuencia de texto en ese idioma. Los tokens timestamp pueden intercalarse en la

secuencia de texto generada, indicando el momento exacto en el que alguna palabra o frase comienza o termina. Esto resulta de gran utilidad para realizar trabajos de alineación audio-texto, como podría ser añadir subtítulos o cualquier aplicación que requiera una sincronización precisa.

- **Traducción X a inglés:** Para traducir el audio de cualquier idioma al inglés, el prompt que introducimos al decoder se modifica para incluir el token '<translate>' además del token con el idioma de origen (es decir, si fuésemos a realizar una transcripción en inglés de un audio en español, nuestro decoder empezaría con los tokens especiales: <es><translate>>startoftranscript>). El modelo entonces procesa el audio y genera la transcripción directamente en inglés. Hay que destacar que, al realizar automáticamente la traducción, no guardará (al menos en el modelo básico pre-entrenado) información sobre las marcas de tiempo.
- **Transcripción solo de texto:** Básicamente como comentábamos en el punto anterior, se trata de la función que se limita a transcribir el texto. Cuando hacemos una traducción, ya hemos visto que no podremos añadir marcas de tiempo; en cambio, al transcribir en el mismo idioma que el empleado en el audio, podemos elegir si transcribir añadiendo time-lapses o dejar simplemente el texto. Transcribir solo el texto puede resultar especialmente interesante para tareas donde solo se necesita la transcripción (como es nuestro caso) o para tareas de fine-tuning donde el modelo se entrena con conjuntos de datos puramente textuales.

Como vemos, la posibilidad de añadir instrucciones detalladas al modelo, le confiere una flexibilidad y versatilidad total en distintos escenarios y tareas, pudiendo controlar su salida de manera más eficiente.

Teniendo a Whisper como base, Faster Whisper logra optimizar el proceso de inferencia. Su principal ventaja, además de una velocidad de inferencia superior (utilizando cuantización de modelos, como int8 o float16) o la eficiencia de recursos (gracias a la compilación con Ctranslate2), la encontramos en su

mantenimiento de la precisión. A pesar de todas sus optimizaciones, Faster Whisper nos ha ofrecido una calidad no solo igual, sino que superior a sus modelos anteriores, minimizando errores que podrían impactar negativamente en la clasificación por LLM.

Su superioridad en las pruebas commparativas realizadas con el resto de STT, refuerza su idoneidad para la tarea de transcribir audios de consultas médico-paciente de manera eficiente y confiable, sentando las bases para el resto del proyecto.

3.3 Modelos de Lenguaje Grandes (LLMs)

Una vez seleccionado el modelo para realizar la transcripción del audio a texto, el siguiente paso dentro del procesamiento del lenguaje natural implica comprender y analizar la información de dicha transcripción. Para esta función emplearemos los Modelos de Lenguaje Grandes (LLMs), modelos de inteligencia artificial basados principalmente en arquitecturas de redes neuronales tipo **Transformer** entrenados en volúmenes masivos de datos.

En este proyecto, necesitaremos que los LLMs sean capaces de identificar relaciones entre entidades clínicas, extraer los datos solicitados y resumir los fragmentos importantes. Resultan una herramienta indispensable para dar sentido a la información no estructurada y poder transformarla en datos útiles.

3.3.1 LLMs considerados

En primer lugar, y antes de empezar las pruebas con LLM, tratamos de explorar todas las opciones disponibles. Sin descuidar la capacidad de PLN del modelo, buscamos optimizar los recursos computaciones disponibles, y particularmente la capacidad de las Unidades de Procesamiento Gráfico (GPU) en entornos locales.

Inicialmente, contemplamos el uso de herramientas PLN más convencionales como **NLTK** (Natural Language Toolkit) y **spaCy.** Para tareas sencillas como la tokenización, el análisis sintáctico o el reconocimiento de entidades nombradas (NER) pueden ser muy eficientes, pero no disponen de suficiente capacidad para

las funciones que requerimos. Ambas presentaban grandes limitaciones a la hora de comprender el contexto de las conversaciones médicas o extraer información. Este análisis preliminar nos mostró la necesidad de recurrir a LLMs, los cuales nos ofrecen la solución para alcanzar el objetivo del proyecto a cambio de aumentar considerablemente los requisitos computacionales.

En la Figura 15 podemos ver los distintos modelos explorados, los cuales presentan distintas arquitecturas y enfoques:

Figura 15. Diferentes modelos LLMs explorados

- **BERT (*Bidirectional Encoder Representations from Transformers*):** Desarrollado por Google, fue el primer modelo en hacer uso de una arquitectura Transformer con entrenamiento bidireccional, la cual permite capturar el contexto completo de una palabra. Los puntos fuertes de este LLM serían el Reconocimiento de Entidades Nombradas (NER) y la clasificación de texto, que serían las funciones más importantes para nuestro proyecto. Esto, sumado a que su modelo base (110M de parámetros) tiene un tamaño que lo hace adaptable a distintos entornos de ejecución, lo convirtió en una opción interesante.
- **RoBERTa (*Robustly Optimized BERT Pretraining Approach*):** Una mejora de BERT, desarrollada por Facebook AI, que optimizó el preentrenamiento (modificando la estrategia de enmascarado y aumentando los datos de entrenamiento) para aumentar el rendimiento

del modelo. Conserva la misma arquitectura que BERT, y ofrece mayor robustez sin aumentar los requisitos de hardware respecto a su predecesor.

- **DistilGPT2:** Se trata de una versión más ligera de GPT-2. Este modelo fue entrenado por Hugging Face, y su diseño (mediante técnicas de destilación de conocimiento) se enfoca en reducir todo lo posible los requisitos de GPU sin perder rendimiento. Aunque fue considerado inicialmente por sus bajos requisitos computacionales y supuesta capacidad de PLN, tras las pruebas de *zero-shot* lo descartamos.
- **Mistral 7B:** Es un modelo de código abierto de Mistral AI. Tiene un gran rendimiento para su tamaño (7B), y su arquitectura se basa en Transformers con atención de grupos de consultas (GQA) y ventanas de atención deslizantes (SWA). Esta arquitectura nos permite manejar secuencias largas de forma más eficiente, y ofrece muy buen rendimiento pese a su tamaño reducido, lo que lo convierte en una muy buena opción para la ejecución local. Aunque tuvo buen rendimiento en la fase *zero-shot*, al realizar el fine-tuning mostró peores resultados que los modelos Llama.
- **Llama 2 (7B, 13B, 70B):** Desarrollado por Meta, se trata de un LLM de código abierto basado en la arquitectura Transformer. Su punto fuerte lo encontramos en su entrenamiento, el cual tiene una cantidad de datos significativamente mayor al resto de modelos de código abierto, lo que contribuye a su robustez y rendimiento. Evaluamos diferentes tamaños, siendo el de 7B es con diferencia el más manejable al ejecutarlo en GPU, y no siendo siquiera capaces de cargar el modelo de 70B por sus altos requisitos de VRAM.
- **Llama 3 (3B,8B):** También desarrollado por Meta. Presenta mejoras sustanciales en rendimiento y capacidades. Al igual que su predecesor, Llama 3 ofrece modelos de diversos tamaños. Ofrerce mayor precisión y capacidades de razonamiento gracias a un entrenamiento aún más extenso que el de su predecesor. Las versiones con menor número de parámetros (3B y 8B) resultaron ofrecer un gran rendimiento, requiriendo para su ejecución requisitos computacionales relativamente bajos.

La selección de los modelos se centró en identificar el LLM que, optimizando la viabilidad de ejecución en los entornos computacionales de los que disponemos, nos ofrezca la mayor eficiencia y capacidad de procesamiento.

3.3.2 Fine-Tuning, LORA y QLORA

Las tareas muy concretas o el lenguaje especializado, como el que podemos encontrar en el ámbito clínico, pueden requerir de una adaptación que optimice el rendimiento del modelo. Necesitaremos realizar un entrenamiento previo sobre nuestro modelo para que, además de comprender la información, sea capaz de extraerla y estructurarla de forma coherente.

Este proceso de adaptación es conocido como ***fine-tuning***, el cual básicamente continua el entrenamiento de un modelo ya pre-entrenado con un conjunto de datos (*dataset*) mucho más pequeño y específico. Sin embargo, realizar un *fine-tuning* del modelo completo supondría una gran carga computacional, necesitando grandes cantidades de memoria GPU para ejecutarlo en un entorno local. La solución más viable la encontramos en las técnicas de *fine-tuning* eficientes, en particular **Low-Rank Adaptation (LoRA)** y su variante cuantizada, **QLoRA.**

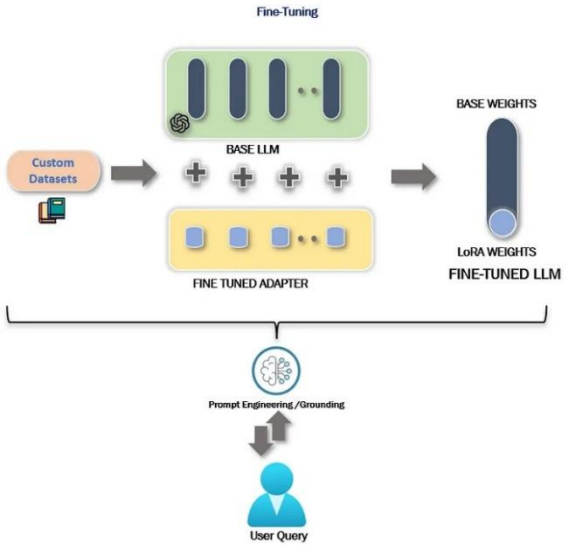

Figura 16. Proceso de fine-tuning de un LLM utilizando LORA

LoRA es una técnica que nos permite reducir drásticamente el número de parámetros a entrenar. Como vemos de forma simplificada en la Figura 16 [20], introducimos un conjunto de datos personalizados (*datasets*) de entrenamiento al modelo pre-entrenado; pero no entrenamos todo el modelo, si no que añadimos una 'actualización' con nuestros datos de entrenamiento (LoRA WEIGHTS). Como se demuestra en [21], esto supone una rebaja considerable en la VRAM necesaria, acelera el proceso de *fine-tuning* y lo más importante, sin perder conocimiento del modelo base.

En el modelo Llama 3 utilizamos **QLoRA (Quantized LoRA).** QLoRA permite aumentar aún más la eficiencia, combinando LoRA con la cuantización de 4 bits. El funcionamiento de QLoRA es el mismo, solo que carga el modelo base en una precisión más baja (4 bits) y los adaptadores LoRA entrenan sobre esta versión cuantizada. Esta técnica nos permite adaptar modelos de miles de millones de parámetros dentro de unas especificaciones de hardware más viables, permitiendo el *fine-tuning* de LLMs tan grandes como Llama.

En el caso de BERT y RoBERTa, pudimos realizar un fine-tuning completo para la tarea de reconocimiento de entidades nombradas (NER). Fue factible realizar el fine-tuning en todas las capas entrenables debido a la diferencia de tamaño respecto a Llama (125 millones contra 7 mil millones), entraremos en más detalle dentro del punto 4.5.

3.3.3 Comparación de los diferentes LLMs

Para seleccionar el LLM más adecuado para las tareas de extracción y análisis de información clínica dentro de una transcripción, compararemos la respuesta de los modelos tras realizar el proceso de fine-tuning. Al realizar las pruebas contábamos con una cantidad limitada de audios que añadir al dataset de entrenamiento, por lo que la calidad y cantidad de pre-entrenamiento que ofrecen los modelos de Llama fue un factor clave a la hora de tomar una decisión. Para probar diferentes arquitecturas de Transformer, también hicimos pruebas con el LLM RoBERTa.

Para realizar las pruebas iniciales, preparamos 20 ejemplos de transcripciones médicas y las integramos al modelo mediante fine-tuning para adaptarlo al

lenguaje y formato de extracción deseado. Realizamos las pruebas en un entorno virtual dentro de Google Colab, aprovechando su capacidad de GPU para ejecutar los LLMs.

- **RoBERTa**: Fue el que mostró peor rendimiento para la extracción estructurada de información clínica. Con el conjunto de entrenamiento que hemos utilizado para el *fine-tuning*, no generamos una respuesta lo suficientemente precisa ni completa para los objetivos del proyecto. Las pruebas sugieren que RoBERTa, al estar más orientado a la clasificación de tokens que a la comprensión contextual, requeriría un volumen de entrenamiento considerablemente más grande.
- **Llama 2:** Mostró una mejora sustancial en la extracción de información comparada con RoBERTa. Fue capaz de identificar la mayor parte de entidades, además de comprender mucho mejor las instrucciones requeridas. Sin embargo, encontramos que tiende a añadir texto explicativo o introductorio a la respuesta de forma complementaria a la información extraída. Aunque el contenido es correcto, no reproduce el formato JSON que esperamos obtener a la salida, lo cual supondría un post-procesamiento que complica la situación.
- **Llama 3:** Obtenemos respuestas estables y estructuradas, siendo hasta ahora el modelo más prometedor. Tras realizar el fine-tuning con el pequeño dataset de entrenamiento, Llama 3 demostró ser la más capaz en extraer la información de forma acertada, concisa y, más importante, en el formato JSON estructurado y sin texto adicional. De los LLMs probados hasta ahora, Llama 3 ha resultado ser el más eficaz para la tarea de análisis de consultas médicas, ofreciendo un gran balance entre rendimiento, tamaño y facilidad de implementación.

3.3.4 Elección del LLM: Llama 3

Tras ser capaces de lograr el fine-tuning y mejorar sus respuestas, nuestra selección final fue LLaMA 3 en su versión de 8 mil millones de parámetros (8B). Esta elección la basamos fundamentalmente en el equilibrio que ofreció entre rendimiento, eficiencia y accesibilidad, características vitales para nuestro

proyecto de investigación. LLaMA (Large Languaje Model Meta AI) representa una serie de modelos pre-entrenados por Meta AI, conocidos por su arquitectura innovadora y su capacidad de afrontar diversas tareas de PLN.

La arquitectura de LLaMA 3, igual que la de sus predecesores, se basa en los modelos *Transformer*, introducidos por Vaswani et al. en 2017. Sin embargo, LLaMA incorpora varias mejoras que optimizan sustancialmente su eficiencia y rendimiento. Lo primero que nos llama la atención es que Llama 3 solo presenta un bloque en su flujo de datos, a diferencia de la arquitectura Transformers clásica que presentaba un bloque con el codificador y otro con el decodificador. Llama solo dispone del bloque decodificador, que trata la entrada y la salida como partes de una única secuencia continua, lo cual favorece a las tareas de generación de texto y de PLN utilizando constantemente la entrada como “fuente de contexto”.

En la Figura 17, analizaremos más detenidamente cómo funciona y qué tareas lleva a cabo para procesar el texto [22].

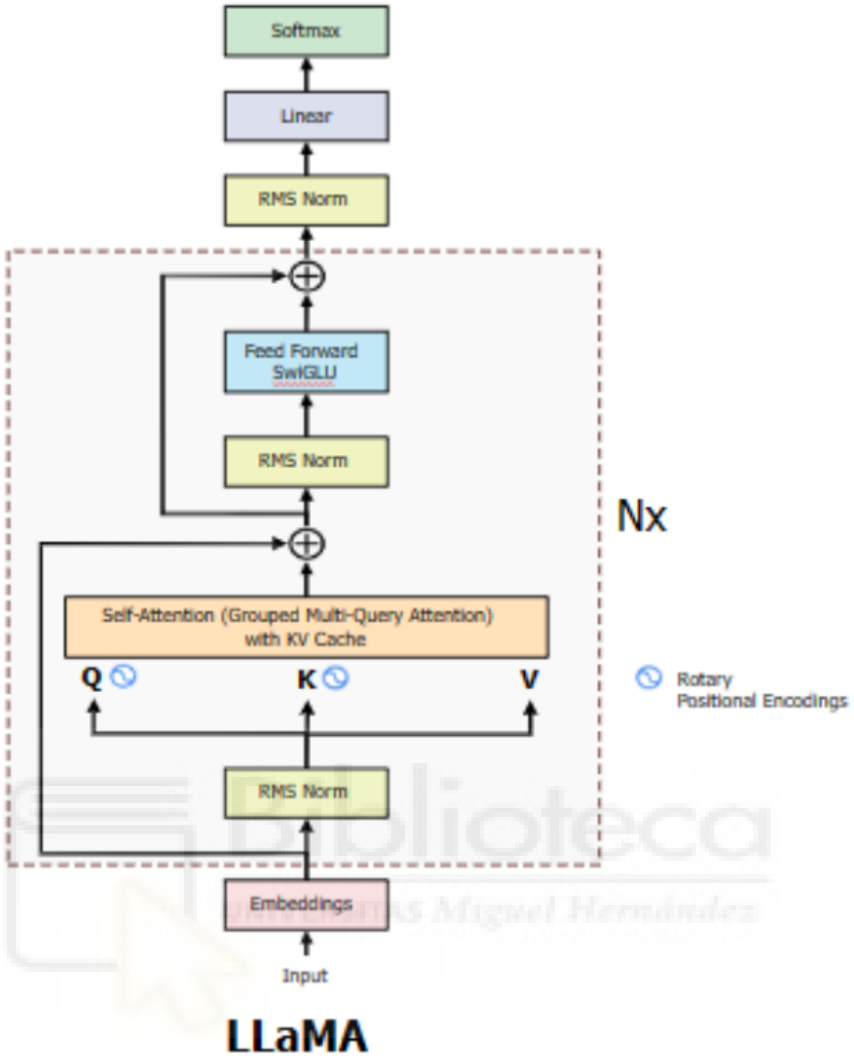

Figura 17. Arquitectura del encoder en LLaMA

Siguiendo el flujo de datos desde la entrada (INPUT), podemos desglosar la imagen en los siguientes componentes clave:

1. **Embeddings:** La entrada inicial al modelo (en nuestro caso particular sería una secuencia de texto) se transforma en representaciones vectoriales densas, conocidas como *embeddings*. Estos vectores capturan la información semántica y sintáctica de cada token.
2. **Normalización RMS:** Tras la capa de embeddings, se aplica una normalización RMS (*Root Mean Square Normalization*). A diferencia de la técnica *Layer Normalization* tradicional que utilizan muchos modelos de

Transformers (incluidas versiones de GPT-2), RMS simplifica el cálculo al normalizar las entradas según la raíz cuadrada de la media de los cuadrados de sus valores, omitiendo el cálculo de la media. Esto reduce significativamente la complejidad computacional, lo cual nos ayuda a estabilizar el entrenamiento y nos permite aumentar el rendimiento.

3. **Mecanismo de Autoatención (Self-Attentiton):** Se trata del elemento fundamental en el que se basa la arquitectura Transformer. En LLaMA 3, empleamos una versión optimizada denominada **Self-Attention** (**Grouped Multi-Query Attention) with KV Cache.** Vemos más en profundidad los mecanismos de autoatención dentro del punto 3.3.5.
4. **Conexión Residual y normalización RMS:** Tras salir de la capa de autoatención, añadimos la entrada original (conexión residual) a la matriz de salida obtenida. Esto otorga cohesión a la respuesta, recuperando los valores iniciales, y facilita el entrenamiento de las redes neuronales profundas. Tras esto, aplicamos otra normalización RMS.
5. **Red Feed-Forward:** Tras pasar la capa de autoatención y su normalización, los datos pasan a través de una red neuronal *Feed-Forward*. Esta capa es la encargada de procesar y transformar las representaciones de los tokens individualmente. LLaMA 3 utiliza activación SwiGLU (Swish-Gated Linear Unit) en esta red, la cual es una variante de la función de activación GLU (Gated Linear Unit) que incorpora la función Swish, proporcionando propiedades de suavidad que pueden mejorar la capacidad del modelo frente a representaciones complejas. Esta capa es fundamental dentro de la arquitectura Llama, pues permite al modelo aprender transformaciones complejas para cada token, enriqueciendo su significado y preparándolo para las siguientes repeticiones (o la salida).
6. De nuevo, tenemos una **conexión residual y normalización RMS**: similar al paso 4, la salida de la red *Feed-Forward* se suma a la entrada y la normalizamos con la RMS.

Vemos que este bloque, el cual abarca desde los *Embeddings* hasta esta segunda conexión residual (y su normalización, aunque en la imagen haya quedado fuera), se repetirá N veces. En LLaMA 3 8B, N será el número de capas

que componen el modelo, lo que permite capturar dependencias y patrones de lenguaje cada vez más complejos y abstractos.

Una vez la información haya sido procesada a través de las N capas de Transformer (32 capas en el caso de LLaMA 3 8B), podemos pasar a los dos últimos pasos:

7. **Capa Lineal (Linear)**: Esta capa actúa como un "traductor" de estas representaciones internas a un formato que el modelo pueda usar para generar la salida. Específicamente, esta capa proyecta los vectores de características a un espacio de dimensiones correspondientes al tamaño del vocabulario de salida del modelo. Es decir, para cada posición de la secuencia, la capa lineal generará una "puntuación" para cada posible token en el vocabulario, indicando cómo de probable es que ese token sea el próximo de la secuencia.
8. **Softmax**: Y finalmente, terminamos con la aplicación de la función Softmax. Los valores obtenidos en la capa anterior no son directamente probabilidades, sino puntuaciones que pueden ser positivas o negativas (y de cualquier magnitud). Esta función toma esas puntuaciones y las transforma en una distribución de probabilidad sobre todo el vocabulario. Es decir, cada puntuación se convierte en un número entre 0 y 1, y al sumar, todas las probabilidades para todos los tokens posibles son igual a 1. En esta distribución, el modelo puede seleccionar el token con la probabilidad más alta como su predicción para el siguiente elemento de la secuencia. Esta capa para nuestro proyecto es muy importante, puesto que permitirá al modelo "elegir" los tokens más probables que formarán las palabras claves de "síntomas, enfermedad o tratamiento", ayudando a que la salida generada se realice a partir de los datos de entrada.

Finalmente, la decisión de utilizar LLaMA 3 (especialmente en comparación con su predecesor LLaMA 2), la basamos en una gran lista de mejoras que se alinean directamente con los requisitos de nuestro TFG: obtenemos un **rendimiento superior**, ofreciendo respuestas más breves, concisas y directas; es el único modelo que ha sido capaz de generar una **salida en formato JSON**, requisito fundamental del proyecto para poder exportar los datos y trabajar con ellos;

encontramos una gran **accesibilidad,** nos otorgaron los permisos de acceso en cuestión de horas, y pudimos bajar y poner a funcionar el modelo (en zero-shot) con el propio código demostrativo de Hugging Face; su tamaño de **tokenizador** con aproximadamente 128000 tokens, más grande que el de LLaMA2, y entrenado en 7 veces más **datos de entrenamiento** que LLaMA2.

En resumen, y más allá de su arquitectura, concluimos en que LLaMA 3 nos ofrece un rendimiento superior, suprime las “alucinaciones” de texto no deseado que provocaban otros modelos, y obtenemos salidas muy buenas y en formato JSON. Todo esto lo convierten en la mejor elección para el desarrollo del proyecto.

3.3.5 Mecanismos de autoatención

Los mecanismos de autoatención (*self-attention)* son sin duda el corazón de la arquitectura Transformer. Fueron introducidos para revolucionar la forma en la que los modelos de lenguaje procesan secuencias. A diferencia de las redes neuronales rercurrentes, que procesaban la información secuencialmente, la autoatención permite considerar **simultáneamente** la relevancia de cada palabra o token dentro de una secuencia con respecto a todas las demás palabras pertenecientes a ésta.

Podemos ver, en la Figura 18, más en profundidad cómo funciona el mecanismo de Autoatención en Transformer [23].

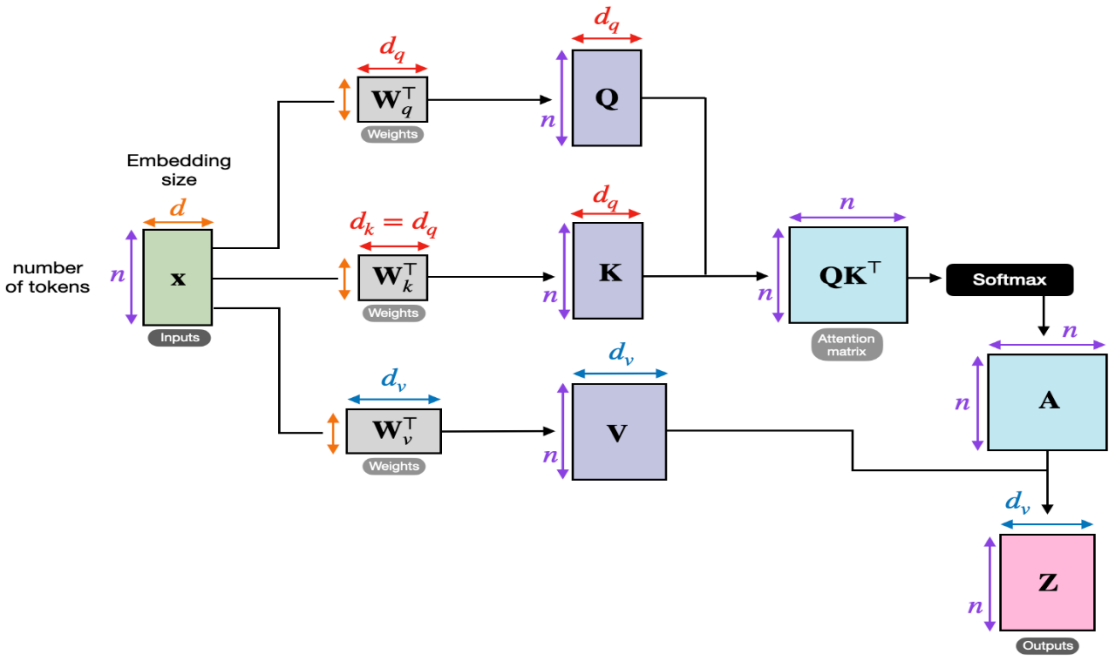

Figura 18. Mecanismos de autoatención en Transformer

Vemos cómo el sistema tiene una entrada 'inputs', la cual se compone por un número de tokens 'n' y un tamaño del embedding 'd' (representación vectorial de nuestros datos de entrada al modelo, como veíamos antes). Tras cada token, se derivan tres representaciones diferentes mediante transformaciones lineales utilizando matrices de pesos (Wq, Wk y Wv). Estas tres representaciones son:

- **Query (Q):** La cual representa la 'consulta' del token actual, buscando información relevante en otros tokens.
- **Key (K):** Actúa como una 'etiqueta' para cada token, indicando el contenido que puede ofrecer. Sus dimensiones son las mismas que Query.
- **Value (V):** Contiene la información real (o el 'valor') con el que cada token es extraído.

A continuación, el cálculo de autoatención calcula la **matriz de atención bruta,** la cual es el producto matricial entre la matriz de Queries (Q) y la transpuesta de la matriz de Keys (K) (ver [23]). El resultado resulta en una matriz de dimensiones nxn donde cada elemento (i,j) indica la similitud o relevancia entre la Query del token *i* y la Key del token *j*. Luego, **escala y normaliza** las puntuaciones obtenidas. Esto convierte las puntuaciones brutas en pesos de atención (A), una

matriz de probabilidades (nxn) donde cada fila suma 1. Estos pesos determinan cuánto deben “atender” a cada token de entrada para la generación de salida de un token específico. Por último, multiplica la matriz de pesos de atención (A) con nuestra matriz inicial de Values (V). La salida (Z, dimensiones: n x dimensiones de value) es una nueva representación para cada token de entrada, que ahora incorpora información contextual de toda la secuencia (con la relevancia calculada en los pasos previos).

Puesto que cada token de salida no se basa solo en su propia información, sino en una combinación inteligente de la información de todos los tokens de la secuencia, Llama es capaz de capturar estas dependencias contextuales y utilizarla a la hora de entrenar o interpretar textos largos. Aunque este mecanismo es ciertamente potente, su implementación directa puede ser computacionalmente muy exigente. Para abordar este desafío, Llama utiliza una técnica de optimización conocida como **KV Cache**.

De forma muy resumida, podríamos decir que KV Cache observó que en la generación autorregresiva las representaciones de K y V de los tokens ya procesados no cambian. Es decir, en lugar de recalcularlos para cada paso de la ejecución, podríamos guardarlos en cache (Atención Multi-Query Agrupada) y ahorrar recursos. Esto permite una reducción significativa en los tiempos de inferencia, además de un aumento de eficiencia de la memoria.

4 DISEÑO E IMPLEMENTACIÓN

En esta sección describimos el proceso de diseño y la implementación técnica de nuestro sistema para la extracción estructurada de consultas médicas. Detallaremos los componentes clave, las arquitecturas del sistema y los pasos seguidos en cada fase del desarrollo.

4.1 Arquitectura del sistema

Diseñamos la arquitectura de este sistema para procesar y analizar de forma secuencial y automatizada el audio de las consultas médicas, extrayendo de ellos información clínica estructurada. El flujo de trabajo completo, desde la entrada del archivo de audio hasta la obtención del archivo en formato JSON, lo ilustramos en la Figura 19.

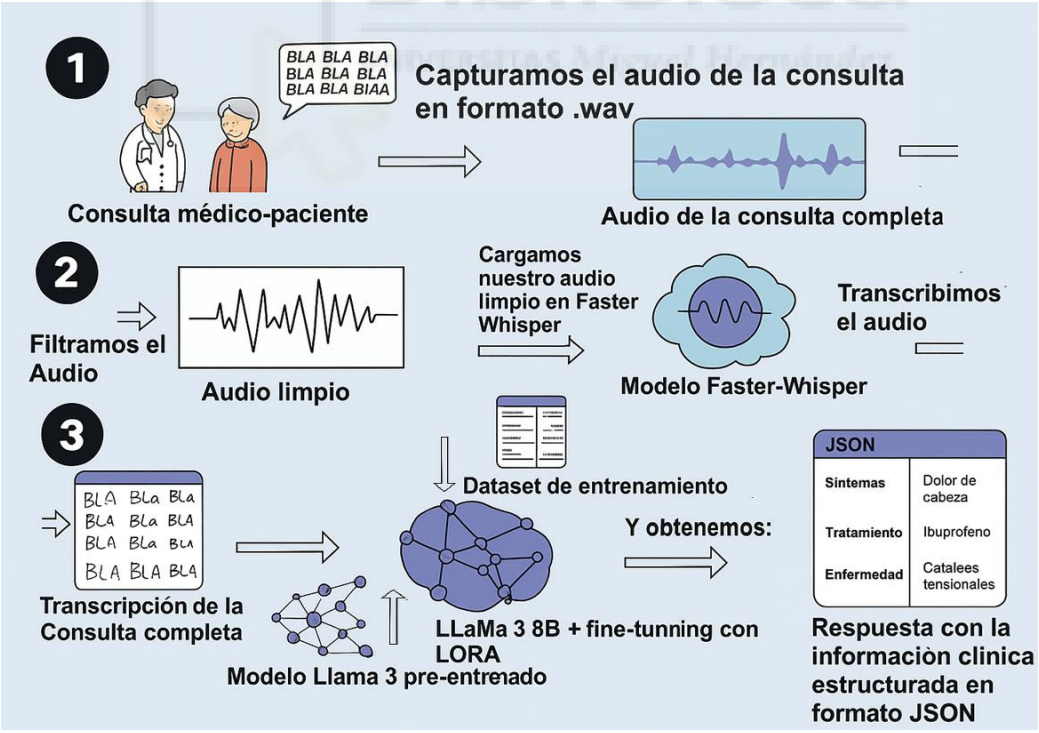

Figura 19. Arquitectura del sistema

Como podemos observar en la Figura 19, el sistema se compone de tres fases principales:

1. Una primera parte donde haremos la **captura y pre-procesamiento** del audio. Esta etapa implica la obtención del audio de la consulta, además de la limpieza del audio y conversión a .wav en caso de ser necesario.
2. Después de preparar el audio, realizaremos la **Transcripción Speech-to-Text (STT)**. Faster Whisper será el encargado de transcribir el contenido de la consulta a texto plano. Éste será utilizado directamente como entrada en el LLM.
3. En este último punto, haremos la **extracción estructurada** de los datos clínicos más relevantes de la consulta. Para ello, introduciremos la transcripción completa de la consulta como *prompt* dentro del modelo LLaMA 3, el cual habrá sido previamente adaptado mediante *fine-tuning* y QLoRA utilizando un dataset de entrenamiento específico para nuestra tarea. El objetivo de esta etapa será procesar el texto y extraer información como síntomas, enfermedades y tratamientos del paciente, para finalmente crear una respuesta con la información clínica estructurada en **formato JSON.**

Esta arquitectura secuencial otorga flexibilidad para futuras mejoras en cada componente, y además asegura una cadena de procesamiento clara, desde el audio de la consulta completa hasta el documento con la información extraída.

4.2 Procesado del audio

Esta etapa inicial aborda el proceso de captura y preparación del audio. Detallaremos las consideraciones sobre la adquisición del audio y las subsiguientes fases de preprocesamiento necesarias para optimizar el proceso de transcripción.

4.2.1 Consideraciones sobre la captura de audio y la privacidad

La calidad del audio resulta determinante para el rendimiento del sistema Speech-to-Text (STT). El sistema permite la captura del audio a través de

diversas metodologías y dispositivos, desde el uso de micrófonos (como los micrófonos de diadema inalámbricos), hasta las propias aplicaciones móviles diseñadas para la grabación de audio. Sin embargo, según la calidad que obtengamos del método de captura, podría ser necesario considerar las características del entorno de la consulta buscando minimizar el ruido ambiente y asegurar una voz clara y audible.

Un aspecto de suma importancia es la **privacidad** y **protección de datos**. Considerando la confidencialidad en las conversaciones entre médico y paciente, es necesario establecer protocolos estrictos para el manejo de los datos de audio y texto recogidos. Debemos garantizar el consentimiento informado de los participantes para la grabación y procesamiento. Fundamentalmente, una vez transcribimos el audio a texto y se ha verificado su calidad, el **archivo de audio** debe ser **eliminado** de forma segura para evitar cualquier brecha de privacidad. Del mismo modo, tras realizar la extracción de información relevante en formato JSON, también eliminaremos la transcripción de texto plano, conservando tan solo los datos estructurados necesarios para el análisis o la aplicación.

4.2.2 Conversión a formato '.wav'´

A la hora de trabajar con modelos de transcripción STT como Faster Whisper, obtenemos los mejores resultados utilizando archivos de audio en formato '.wav'. Es posible que, al utilizar distintas herramientas para capturar el audio, encontremos el archivo en diferentes formatos (.mp3, .mp4, .ogg, etc.). Antes de iniciar el filtrado y procesamiento del audio, comprobamos y en caso de ser necesario convertimos el formato a .wav para asegurar una entrada consistente al sistema STT.

Para realizar la conversión, utilizamos la librería de Python 'pydub'. Pydub es una herramienta versátil y permite la manipulación y conversión de audio en múltiples formatos.

El proceso implementado verifica la extensión del archivo de audio de entrada, probando a añadir las extensiones más comunes al nombre predefinido del

archivo; si no es .wav, pydub se encarga de cargarlo y exportarlo al formato deseado.

Esta conversión no solo nos garantiza la compatibilidad con el modelo, sino que también establece unas propiedades base para todos los audios, como la tasa de muestreo y la codificación, las cuales juegan un papel fundamental en la transcripción. Cabe destacar que pydub requiere la instalación de *ffmpeg* en el entorno para soportar los formatos de entrada y salida del audio.

La implementación detallada de esta función de conversión puede consultarse en el Anexo 7.2. ANEXO II: Conversión a .wav.

4.2.3 Filtrado del audio

Nuestro objetivo actual es optimizar la calidad del audio antes de hacer la transcripción para mejorar la precisión del sistema STT. Para ello, implementamos un proceso de filtrado. El objetivo principal será atenuar el ruido de fondo y eliminar las frecuencias irrelevantes que puedan interferir con el habla.

Realizamos pruebas con diversas técnicas y librerías para evaluar su eficiencia:

- **Detección de Actividad de Voz (VAD) con *'webrtcvad'*:** Exploramos el uso de sistemas VAD para suprimir automáticamente los segmentos de silencio dentro de la grabación. Sin embargo, las pruebas iniciales ya mostraron limitaciones. Aunque era capaz de detectar y cortar eficientemente los silencios al inicio y final de los archivos de audio, la eliminación de silencios intermedios (o momentos con ruido no verbal) resultó ser significativamente más compleja. El uso eficiente de esta herramienta requería de una configuración muy precisa de los parámetros, la cual no fue viable debido a las diferencias que presentaban los distintos audios empleados para los experimentos (calidad de grabación, volumen, distintos interlocutores, etc.). Por esta razón, decidimos no integrar VAD en la fase de filtrado.
- **Reducción de Ruido con '*noisereduce*':** También evaluamos técnicas de reducción de ruido, como las que ofrece la librería 'noisereduce'. Estas

técnicas buscan atenuar el ruido presente en la banda de frecuencia del habla. Estas técnicas pueden ser muy potentes para ciertos tipos de ruido, pero a la hora de aplicarlo en nuestras grabaciones no arrojó mejoras sustanciales en la calidad percibida para la transcripción. De hecho, en algunos casos, observamos que 'noisererduce' introducía distorsiones significativas en el habla, resultando en transcripciones con errores y repeticiones.

- **Filtro Paso-Banda con '*scipy.signal*':** Finalmente, el filtro con el que obtuvimos mejores resultados además de una mejora de la calidad de audio fue un filtro paso-banda. Vemos su código de forma más detallada en la Figura 20.

```
  #Definimos el filtro paso-banda
def butter_bandpass(lowcut, highcut, fs, order=5):
    nyq = 0.5 * fs #frec. de nyquist =
    low, high = lowcut / nyq, highcut / nyq
    b, a = butter(order, [low, high], btype='band')
    return b, a


  #Aplicamos el filtro a los datos de audio
def butter_bandpass_filter(data, lowcut, highcut, fs, order=5):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    y = lfilter(b, a, data)
    return y
  #Cargamos nuestro audio y lo filtramos a 200-3000 Hz
def limpiar_audio(archivo_audio):
    y, sr = librosa.load(archivo_audio)
    lowcut, highcut = 200, 3000
    y_filtered = butter_bandpass_filter(y, lowcut, highcut, sr)
    return y_filtered, sr
```

Figura 20. Código del filtro paso-banda para limpiar nuestro audio de frecuencias no deseadas

El filtro está diseñado para retener las frecuencias donde la mayor parte de la energía del habla se concentra y atenuar las frecuencias fuera de ese rango. Podemos desglosar el código en tres partes:

1. Definimos la función ***butter_bandpass****,* la cual será la encargada de diseñar los coeficientes de un filtro paso-banda Butterworth. No aplicamos el filtro directamente, sino que calculamos los parámetros necesarios para que el filtro pueda ser aplicado posteriormente. Introduciendo la frecuencia de corte inferior y superior, la frecuencia de muestreo y el orden del filtro, podemos calcular la frecuencia de Nyquist y con ella normalizar las frecuencias de corte. Tras esto, hacemos uso de la librería scipy.signal para crear un filtro Butterworth (con la función *butter*(), como vemos en la imagen de código) y devolvemos 'b' y 'a', los cuales son los dos conjuntos de coeficientes (coeficientes del numerador y denominador del filtro)
2. En la segunda función, ***butter_bandpass_filter***, introducimos los datos de audio (un array NumPy de muestras) y los mismos parámetros de frecuencias y orden que se usaron para diseñar el filtro. Llamamos a la función anterior para calcular los conjuntos de coeficientes, y hacemos uso de la función de scipy '*lfilter()*'. Esta función será la encargada de aplicar el filtro digital sobre los datos de audio.
3. Por último, la función ***limpiar_audio*** que usaremos para cargar el archivo de audio, además de definir en ella los valores del filtro que deseamos. Cargamos el audio haciendo uso de la librería *Librosa (*con la función *'librosa.load()'*), al cargar el audio con esta librería, forzamos el muestreo a **22050Hz** si no está ya en esa frecuencia, lo convertimos a **mono** en caso de estar en formato estéreo y normalizamos el volumen del audio. La función devuelve la señal de audio ya filtrada (*y_filtered*) y la frecuencia de muestreo (*sr*) con la que fue procesada.

Se estableció una banda de paso entre 200Hz y 3000Hz, un rango óptimo para la inteligibilidad de la voz, con un orden de filtro de 5. Este enfoque nos permitió eliminar eficazmente ruidos de baja frecuencia (como los zumbidos) y los ruidos de alta frecuencia (como los pitidos o silbidos), sin afectar a la claridad de las voces.

4.3 Speech to Text

La fase de Speech-to-Text (STT), o reconocimiento de voz a texto, es posiblemente el punto de mayor importancia de nuestro sistema. Su función es transformar las grabaciones de audio de las consultas médicas en texto escrito, para el posterior procesamiento y extracción de los datos mediante nuestro LLM. La precisión de esta fase es determinante, puesto que cualquier error en la transcripción puede estropear toda la información extraída.

Para esta tarea se implementó **Faster Whisper**, una versión optimizada del modelo Whisper de openAI, conocida por su eficiencia computacional y su alta calidad de transcripción en distintos idiomas. A continuación, detallaremos los pasos de implementación para la carga del modelo, la segmentación del audio y el proceso de transcripción utilizando Faster Whisper.

4.3.1 Carga del modelo STT

La primera etapa del proceso STT será la carga del modelo Faster Whisper en el entorno de ejecución. Siguiendo los pasos recomendados de [24] , primero llevaremos a cabo la carga del modelo a través de la función '*cargar_modelo_whisper*', la cual podemos ver a continuación en la Figura 21:

```python
def cargar_modelo_whisper(model_size: str = "medium") -> WhisperModel:
    """Carga el modelo Whisper una sola vez."""
    #ejecucion en GPU (si no es posible, en CPU):
    device = "cuda" if torch.cuda.is_available() else "cpu"
    #cuantificación int8:
    compute_type = "int8" if device == "cuda" else "int8"
    modelo = WhisperModel(model_size, device=device, compute_type=compute_type)
    print("Modelo Whisper cargado.")
    return modelo
```

Figura 21. Carga y configuración del modelo Faster Whisper

Durante la inicialización, seleccionamos el tamaño de modelo "medium" por su equilibrio entre precisión y requisitos de procesamiento. Dentro de la configuración también tendremos que seleccionar el dispositivo donde se

ejecutará, si hay una GPU disponible hará uso de “CUDA”, en caso contrario usará la CPU.

Por último, también especificamos el tipo de cómputo, en nuestro caso int8, el cual es una optimización clave para reducir el uso de memoria y acelerar la inferencia sin una pérdida significativa de precisión. Esta función nos garantiza que el modelo se configure de manera óptima para el hardware disponible, preparándolo para las tareas de transcripción. Con esto ya tenemos el modelo cargado, pero necesitamos llevar a cabo ciertos ajustes antes de poder introducir el audio y obtener la transcripción.

4.3.2 Segmentación del audio

Las grabaciones de audio en consultas médicas presentan duraciones variables. En nuestros experimentos hemos probado grabaciones de hasta 40 minutos, las cuales resulta inviable procesar de una sola vez considerando los límites de memoria de la GPU. Además, la transcripción de audios completos presentó generalmente un aumento de los errores en el texto final. Para mitigar estos problemas, decidimos implementar funciones de **segmentación de audio**.

El audio se divide en segmentos de duración predefinidas (por defecto elegimos 60 segundos en las pruebas finales). Procesamos cada segmento de forma independiente y vamos insertando los fragmentos transcritos dentro de la *string* ‘*transcripciones_completas’*. Este enfoque mejora la estabilidad del proceso y nos permite una gestión más eficiente de los recursos.

El código para la implementación de la lógica de segmentación se ha incluido en el punto del anexo 7.3 ANEXO III: Código para segmentar y transcribir el audio en Faster Whisper.

Podríamos resumir la función ‘*transcribir_audio_segmentado’* en cinco pasos:

1. Cargamos el archivo de audio usando ‘*librosa.load*’, lo remuestreamos a 16000Hz (la cual se trata de la frecuencia de muestreo esperada por Whisper) y obtenemos su duración total.
2. Calculamos el número de segmentos necesarios basándonos en la duración total y el tamaño de segmento. En nuestro caso, fijamos un

tamaño de segmento predefinido dentro de la función de 30 segundos, pudiendo configurar el parámetro según lo deseemos al llamar a la función (las pruebas finales las realizaremos estableciendo un valor de 60).

3. Una vez tenemos el audio dividido en segmentos iteramos sobre cada elemento, extrayendo el fragmento de audio correspondiente.
4. Aplicamos el filtro paso-banda que veíamos en el punto anterior a cada segmento de audio antes de la transcripción. Una vez tenemos el segmento limpio, hacemos uso de la función que veremos en el siguiente punto (4.3.3). Este paso es opcional, pudiendo deshabilitar los filtros para realizar pruebas desde el script principal main().
5. Una vez hemos transcrito cada segmento, podemos borrar el audio original. Este paso también es opcional, y podemos desactivarlo desde el script principal main().

4.3.3 Transcripción del audio

Una vez hemos cargado el modelo Faster Whisper y el audio está preparado, procedemos a la transcripción. En esta parte, la función clave que debemos utilizar de Whisper es '*transcribe'.*

Básicamente, creamos la función '*transcribir_array_audio',* a la cual pasaremos como variable nuestro segmento de audio extraído del array de segmentos, además de un sr (*sample rate*, 16KHz) y el modelo (Faster Whisper) a utilizar. Vemos con más detenimiento el código, insertado como Figura 22:

```
def transcribir_array_audio(audio_array: np.ndarray,
                            sr: int, modelo_whisper: WhisperModel) -> str:
   #cargamos el array de audio en formato float32
    audio_array_float32 = audio_array.astype(np.float32)
    #hacemos la transcripcion con faster whisper
    segments, info = modelo_whisper.transcribe(audio_array_float32, beam_size=5)
    #y devolvemos el SEGMENTO de texto transcrito
    return " ".join([segment.text for segment in segments])
```

Figura 22. Transcripción por segmentos en Faster Whisper

La función '*modelo_whisper.transcribe'* tendrá como parámetros el segmento de audio, dado como un array NumPy (*audio_array_float32*, y un valor '*beam_size'*. El Beam Size se trata del valor que configura la búsqueda en haz (beam), y podríamos definirlo como el valor que define cómo de 'avaricioso' será el modelo buscando la mejor transcripción posible. En nuestro caso decidimos establecer el valor en 5, el cual nos proporcionó el mejor equilibrio entre precisión y velocidad. Un tamaño del haz muy grande (por ejemplo 20) podría ser computacionalmente muy costoso y realmente no presenta mejoras proporcionales en la calidad de la transcripción.

Los resultados de la transcripción serán una lista de objetos llamada *Segment* que contienen el texto transcrito para cada segmento detectado por Whisper. Unimos estos segmentos para formar la transcripción final.

```
modelo_whisper = cargar_modelo_whisper() #cargamos el modelo
 #nombre del archivo de audio:
archivo_audio = ("Introduce el nombre del archivo de audio: ")
 #convertimos a .wav si es necesario:
archivo_audio=convertir_transcripcion_a_wav_si_necesario(archivo_audio)
print("\nIniciando transcripción...")

tiempo_inicio = timeit.default_timer() #iniciamos timeit

#transcribimos el audio segmento a segmento:
transcripcion_final = transcribir_audio_segmentado(
        archivo_audio,
        modelo_whisper,
        tamano_segmento=60,
        aplicar_filtro=False
    )

tiempo_fin = timeit.default_timer()
#Calculamos el tiemo total de transcripción:
tiempo_total =tiempo_fin-tiempo_inicio

print("\n--- Transcripción Completa ---")
print(transcripcion_final) #Mostramos el resultado
print(f"Tipo de variable: {type(transcripcion_final)}")
print(f"Longitud: {len(transcripcion_final)} caracteres")
print(f"Tiempo total: {tiempo_total:.2f} segundos")
```

Figura 23. Script principal STT

Por último, cabe destacar el script principal, el cual vemos en la Figura 23, que hará la función de director, ejecutando secuencialmente las funciones definidas durante este apartado. Este script coordina el flujo de trabajo completo para transformar el audio de consulta en una transcripción de texto coherente. En esta fase es donde definiremos parámetros globales de ejecución, como el tamaño de segmento o la decisión de usar o no los filtros paso-banda. Además, también añadimos la librería timeit (anexo 7.1) para realizar las mediciones de rendimiento y tiempo de ejecución, las cuales fueron fundamentales a la hora de realizar experimentos y decidir qué modelo elegir.

Grabamos un audio simulando una conversación entre médico y paciente desde nuestro teléfono móvil (formato .ogg), lo pasamos por nuestro sistema STT y vemos los resultados en la Figura 24:

```
Nombre del archivo de audio (sin formato): pruebamargarita
Archivo encontrado y procesando: pruebamargarita.ogg
El archivo 'pruebamargarita.ogg' no está en formato WAV. Intentando convertir a WAV...
¡Conversión exitosa! El archivo convertido se ha guardado como 'transcripcion.wav'.

Iniciando transcripción...
Cargando audio 'transcripcion.wav' y remuestreando a 16000 Hz...
Duración total del audio: 89.94 segundos.
Transcribiendo segmento 1/2 (0.0-60.0s)...
Transcribiendo segmento 2/2 (60.0-89.9s)...

--- Transcripción Completa ---
 ¡Hola, buenos días, doctor!  Hola, Margarita, ¿verdad? Eso es.  Oye, ensiéntese, por f
Tipo de la variable 'transcripcion_final': <class 'str'>
Longitud de la transcripción: 1077 caracteres
Tiempo total de transcripción: 9.60 segundos
```

Figura 24. Resultados de la transcripción utilizando nuestro script con Faster Whisper

Realiza la conversión al formato .wav correctamente, lo muestrea a 16kHz y realiza dos etapas de transcripción (parte el audio en segmentos de un minuto). En la Figura 25 presentamos la transcripción completa como referencia para futuros resultados, en la cual hemos tratado de simular una conversación breve que pudiese tener lugar en consulta:

```
transcripcion_memoria = transcripcion_final.split()
for i in range(0, len(transcripcion_final), 10):
    print(' '.join(transcripcion_memoria[i:i+10]))
```

```
¡Hola, buenos días, doctor! Hola, Margarita, ¿verdad? Eso es. Oye,
ensiéntese, por favor. Vale, y cuénteme, ¿qué le pasa? Ver
a doctor. Llevo un par de días que no puedo
ni dormir, ni descansar, ni estar en casa prácticamente porque
tengo un dolor de cabeza y unos mareos que no
son normales. Vaya, ¿qué me dice y qué parte le
duele? Me sube un hormigueo por el cuello, aquí detrás
de la cabeza, y uff, me dan unos mareos. Por
lo que me cuesta, es posible que hayan vuelto sus
cefaleas tensionales, Margarita. ¿Está usted aplicándose frío y tomando analgésicos?
No, doctor. La verdad que no lo estoy haciendo. No
me gusta tomar pastillas. Pues mira, Margarita, lo que le
voy a recetar... ustedes ibuprofeno de 600 miligramos eso no
me va a subir la tensión no se preocupe margarita
ver usted que con reposo aplicando ese frío y esta
medicación va usted a estar mucho mejor si en unos
días no mejora vuelva por aquí y quizás le hagamos
una resonancia para descartar que tenga nada más grave vale
muchas gracias doctor sin ningún problema gracias a ti margarita
```

Figura 25. Transcripción de consulta médica

Como podemos ver, se trata de un texto sin diarizar (es decir, no separa los interlocutores), desestructurado y con algunas palabras transcritas de forma incorrecta. Sin embargo, este modelo es el que menor WER ha presentado y el que ha tenido un comportamiento más estable (sin cuelgues ni “alucinaciones”).

Con esto, concluimos la primera parte del proyecto, la cual abarca desde la captura y el preprocesamiento del audio hasta su transcripción a texto. Una vez tenemos la transcripción, es el momento de llamar a nuestro LLM para procesar la información.

4.4 Text-To-Table con LLM

Una vez que nuestro audio de consulta médica ha sido transformado en texto mediante el proceso de Speech-to-Text, nuestra siguiente fase del proyecto se centra en el **Procesamiento de Lenguaje Natural (PLN)**. Nuestro objetivo principal en esta etapa es extraer y estructurar la información relevante de las transcripciones de texto no estructurado, transformándola en un formato que podamos analizar y utilizar para fines clínicos, de entrenamiento o de gestión.

En este contexto, los **LLMs** juegan un papel fundamental debido a su capacidad para comprender, interpretar y generar texto. Específicamente, nos serán útiles sus habilidades para realizar tareas de extracción de información y su capacidad de formatear datos.

Esta sección abordará la carga del LLM, la realización de pruebas antes de realizar el entrenamiento específico (pruebas Zero-Shot), el proceso de adaptación mediante *fine-tuning* y QLoRA, y la obtención final de la información estructurada en una tabla con formato JSON. Además, también veremos cómo cargar el modelo de forma local mediante Ollama.

4.4.1 Carga del modelo LLM

Para nuestra tarea de extracción de información médica, seleccionamos un modelo de la familia Llama 3.1, específicamente la variante Llama 3.1 8B-Instruct desarrollado por Meta. Este modelo, con 8 mil millones de parámetros, nos ofreció grandes resultados con su rendimiento, tamaño y eficiencia computacional, ideal para entornos como Google Colab.

La carga del modelo la realizamos utilizando la librería Transformers de Hugging Face, la cual nos simplifica la interacción con modelos preentrenados.

```
#Cargamos el modelo preentrenado:
model_id = "meta-llama/Llama-3.1-8B-Instruct"
#Cargamos su tokenizer correspondiente:
tokenizer = AutoTokenizer.from_pretrained(model_id)
#Establecemos los parámetros del modelo
model = AutoModelForCausalLM.from_pretrained(
        model_id, #versión del modelo (LLaMA 3.1-8B-Instruct)
        torch_dtype=torch.bfloat16, #Formato bfloat16
        device_map="auto", #Elige GPU o CPU según su disponibilidad
        load_in_4bit=True #Cargamos el modelo cuantizado en 4bits
    )

print(f"Modelo {model_id} cargado con éxito")
print(f"Tipo de datos del modelo: {model.dtype}")
```

Figura 26. Carga del modelo LLaMA 3.1-8B-Instruct

En la Figura 26 podemos ver el código empleado para inicializar el modelo preentrenado y su *tokenizer.* La función *AutoTokenizer.from(…)* carga el *tokenizer* asociado al modelo Llama 3.1, el cual es esencial para convertir el texto de entrada en *tokens* numéricos que puedan procesar y decodificar la salida numérica del modelo de vuelta a texto legible.

Por otro lado, la función *AutoModelForCausalLLM.from_pretrained(model_id,…)* carga el modelo LLM preentrenado. La función contiene (además del modelo que hemos inicializado previamente) los siguientes parámetros:

- ***Torch_dtype=torch.bfloat16*:** La cual especifica el tipo de datos para los pesos del modelo. El formato *bfloat16* (16 bits) ofrece un buen equilibrio entre precisión numérica y consumo de memoria (VRAM) en comparación a *float32*, y es más estable que *float16* para el entrenamiento.
- ***Device_map="auto"*:** Esta configuración permite que la librería *accelerate* (la cual está integrada en *Transformers*) distribuya automáticamente las capas del modelo a través de las GPUs disponibles o, en caso de ser necesario, las coloque en CPU. Esto nos ayuda a optimizar el uso de memoria GPU.
- ***Load_in_4bit=True***: Esta opción nos habilita la cuantización del modelo a 4 bits de precisión (dentro de *bitsandbytes*, también integrada en *Transformers).* La cuantización reduce significativamente la GPU requerida para cargar y ejecutar el modelo, lo cual nos permite ejecutar modelos más grandes en hardware con recursos limitados. Baja la precisión del modelo, aunque no observamos que tenga gran impacto sobre nuestros resultados, por lo que nos resulta de gran utilidad para reducir los requisitos de VRAM y los tiempos de ejecución. Esta carga reducida del modelo nos permitió incorporar técnicas más avanzadas como el *fine-tuning*, las cuales resultaban inviables con la carga computacional que suponía el modelo completo.

La carga del LLM puede requerir una descarga inicial si no está en caché, lo que puede tomar tiempo dependiendo de la conexión a internet. Además, es posible que para utilizar el modelo haga falta descargar e introducir un *token* identificativo

especial de Hugging Face con el acceso al modelo. Una vez cargado, el modelo está listo para empezar los experimentos.

4.4.2 Pruebas Zero-Shot

Una vez tenemos el modelo Llama-3.1-8B-Instruct cargado, procedemos a realizar pruebas en un escenario **Zero-Shot.** En el contexto de los LLMs, el aprendizaje Zero-Shot hace referencia a la capacidad del modelo para realizar tareas para las que no ha sido entrenado específicamente. El modelo se basa únicamente en el conocimiento general que adquiere durante el preentrenamiento y en la claridad del prompt.

El objetivo inicial de estas pruebas fue medir la capacidad del modelo para extraer la información (síntomas, enfermedades y tratamientos) y formatearla como una lista de objetos JSON. Las pruebas Zero-Shot nos proporcionan un punto de referencia para evaluar la efectividad de adaptación posteriores. Al comparar los resultados, podremos cuantizar y demostrar el impacto real y la mejora de la adaptación con LORA.

Lo primero que haremos será diseñar el ***prompt*** *(*instrucciones para el modelo*),* el cual es fundamental para las tareas Zero-Shot, ya que debe guiar al modelo con precisión. Para interactuar con Llama 3.1, estructuramos el *prompt* utilizando el formato de mensajes de chat, que típicamente se compone de un mensaje “sistema” y un mensaje “usuario”.

```
# Introducimos nuestra transcripción como input
input_text = transcripcion_final

prompt_template = f"""
Eres un experto en extracción de información médica.
Tu tarea es identificar y extraer SÍNTOMAS, ENFERMEDADES
y TRATAMIENTOS mencionados en el texto. Debes listar estas
entidades en formato JSON, asegurándote de que cada tipo
sea exactamente 'SINTOMA', 'ENFERMEDAD' o 'TRATAMIENTO'.
Si no se encuentra una entidad de un tipo específico,
esa clave debe estar ausente o ser una lista vacía.
No incluyas ninguna otra información, explicaciones o
texto adicional fuera del JSON.

  Texto: "{input_text}"
    """ #Nuestro Prompt Usuario
```

Figura 27. Prompt 'usuario' para la extracción de información clínica en LLaMA 3

El *prompt* de usuario lo podemos ver en la Figura 27, el cual contiene la instrucción específica para la tarea de extracción de entidades. El *prompt* incluye la transcripción obtenida de Faster Whisper (como variable *input_text)* junto con las especificaciones detalladas sobre qué información extraer y el formato de salida requerido. Le pedimos que extraiga los síntomas, enfermedades y tratamientos que encuentre dentro de la transcripción de consulta.

Después, en la Figura 28 podemos ver la variable *messages*, la cual incluirá tanto el *prompt* usuario como el *prompt* de sistema, el cual establece el rol del modelo y sus directrices fundamentales.

```
messages = [
    {"role": "system",
     "content": """Eres un asistente experto en reconocimiento
                  de entidades nombradas médicas. Siempre respondes
                  con JSON."""},
    {"role": "user",
     "content": prompt_template}
           ]

# Establecemos la longitud maxima de la cadena
MAX_CONTEXT_LENGTH = 8192 # Límite de tokens para Llama 3.1-8B-Instruct
```

Figura 28. Prompt 'sistema' para la extracción de información clínica en LLaMA 3

En nuestro caso hemos pedido al modelo que actúe como un experto en reconocimiento de entidades médicas y que nos devuelva siempre un JSON. El 'mensaje de sistema' es vital para instruir al modelo sobre su comportamiento y el formato de salida deseado. Debe ser explícito, incluyendo instrucciones claras sobre el rol y las instrucciones que debe seguir.

Antes de proceder con la inferencia, también realizamos una verificación de la longitud del *prompt* para asegurar que no exceda el límite de tokens del modelo (8192 tokens para Llama 3.1-8B-Instruct) y sea capaz de generar una respuesta completa.

La generación de la respuesta del modelo la realizamos mediante la función *model.generate()* de la librería Transformers. En la Figura 29 podemos ver con mayor detalle los parámetros utilizados, sus valores y su función.

Después, realizamos la *tokenización* del texto de entrada al modelo. Inicializamos la variable ***input_ids,*** a la cual le añadimos *messages*, variable que contiene los *prompts* que hemos establecido antes. Como podemos ver en la Figura 29, dentro de esta variable también activamos los parámetros para la tokenización, el añadido de prompt generativo y pedimos que su respuesta la devuelva como **tensores de PyTorch**.

Los tensores son la estructura fundamental que utilizan las bibliotecas de aprendizaje (como PyTorch o TensorFlow) en modelos como Llama 3. Como veíamos en el punto 3.3.4, el primer paso que siguen los modelos será convertir el texto a token, y éstos se organizaran en tensores. El modelo realiza todas las

operaciones matemáticas sobre estos tensores para entender y generar una respuesta.

```
#inicializamos los input_ids (prompts+transcripción)
input_ids = tokenizer.apply_chat_template(
        messages, #con los prompt
        tokenize=True,
        add_generation_prompt=True,
        #y los devolvemos como pytorch tensors
        return_tensors="pt"
    ).to(model.device)
#Parámetros del modelo para generar la salida
outputs = model.generate(
        input_ids, #nuestros ids de entrada
        max_new_tokens=500, #máximo de tokens
        do_sample=False, #Muestreo aleatorio
        top_p=0.9, #Nucleus Sampling
        temperature=0.7, #Nivel de creatividad
        num_return_sequences=1 #Número de respuestas por prompt
    )

respuesta = tokenizer.decode(outputs[0, input_ids.shape[-1]:],
                             skip_special_tokens=True)

print("\nRespuesta del Modelo (texto crudo):")
print(respuesta)
```

Figura 29. Configuración e inferencia del modelo

Cargamos el modelo como *outputs = model.generate(…)* y describimos con mayor profundidad los parámetros utilizados:

- **Inputs_ids** que contiene nuestros prompts de usuario y de sistema convertidos en tensores PyTorch.
- **Max_new_tokens** Este parámetro define el límite máximo de tokens que el modelo puede generar como respuesta. Se mantiene un valor alto para asegurar la salida JSON completa.
- **Do_sample=False** habilita el muestreo estocástico (aleatorio) durante el proceso de generación. Introduce un grado de aleatoriedad en la selección del próximo *token,* lo que ayuda a producir respuesta más diversas y menos predecibles o repetitivas. Para nuestro proyecto sería

perjudicial añadir respuestas aleatorias, así que lo establecemos como 'False'.

- **Top_p=0.9** implementa la estrategia de muestreo top-p, también conocida como *Nucleus Sampling.* Cuando veíamos el funcionamiento de LLaMA 3, recordamos que el último paso (Softmax) repartía a cada token una probabilidad de ser el próximo en la salida generada. El modelo considera solo el subconjunto de tokens que en conjunto suma una probabilidad acumulada a la salida que excede el umbral establecido (90% en nuestro caso). Esto nos ayuda tanto a gestionar de forma más eficiente la GPU, como a descartar tokens con baja probabilidad que podrían empeorar la coherencia en la respuesta.
- **Temperature=0.7** se trata del parámetro que controla el nivel de aleatoriedad de la generación. Valores más altos resultan en respuestas más creativas, mientras que valores más bajos producen respuestas más deterministas y enfocadas. El valor 0.7 es el recomendado, siendo un buen equilibrio entre coherencia y variación.
- **Num_return_sequences=1** Indicamos que el modelo generará una única respuesta para cada *prompt* de entrada.

La *response_text* resultante es la cadena de texto cruda generada por el LLM, que teóricamente debería contener el JSON estructurado solicitado.

```
Longitud actual del prompt (incluyendo transcripción): 321 tokens.
Límite máximo de tokens del modelo: 8192 tokens.

Generando predicciones ...

Respuesta del Modelo:
{
  "SÍNTOMAS": [
    "dolor de cabeza",
    "mareos",
    "cefaleas tensionales",
    "dolor",
    "mareo"
  ],
  "ENFERMEDADES": [
    "cefaleas tensionales"
  ],
  "TRATAMIENTOS": [
    "aplicarse frío",
    "tomar analgésicos",
    "ibuprofeno",
    "reposo",
    "medicación",
    "resonancia"
  ]
}
```

Figura 30. Resultados de la prueba Zero-Shot

Las pruebas Zero-Shot nos dieron una idea sobre las capacidades de Llama para la extracción de entidades médicas. Como podemos ver en la Figura 30, obtuvimos resultados muy positivos. Con transcripciones cortas y claras el modelo presentó buen funcionamiento (aunque añadía a menudo información redundante), pero en transcripciones más extensas presentó tendencia a perder el foco y distorsionar la respuesta.

En los resultados obtenidos, podemos observar que repite varias veces los mismos síntomas. Además, añade redundancias como “medicación” dentro del apartado de tratamientos, y elementos no confirmados como “resonancia”, la cual no indica explícitamente que vaya a realizarse.

Tras realizar las técnicas de *fine-tuning y LORA*, podremos medir la diferencia y ver si hay una mejora notable que nos acerque a cumplir los objetivos.

4.4.3 Preparación del dataset de entrenamiento

Antes de entrenar al modelo mediante las técnicas *fine-tuning*, el primer paso fundamental que debemos llevar a cabo es la creación de un *dataset* de entrenamiento con ejemplos que muestren el funcionamiento deseado. Cada ejemplo debe incluir una transcripción en el *prompt* de entrada y debe devolver una salida JSON estructurada. Este conjunto de pruebas de entrenamiento mostrará al modelo cómo debe aplicar su conocimiento médico y qué formato de salida esperamos.

Para construir el *dataset*, utilizamos ejemplos de transcripciones de audio de consultas clínicas. Cabe destacar que, si bien una pequeña parte de estos audios fueron tomados realmente en consulta, la gran mayoría de conversaciones médicas fueron inventadas y creadas específicamente para la realización del proyecto, debido a la dificultad de obtención y las restricciones de privacidad que suponían los datos reales. Estas conversaciones sintéticas fueron grabadas y procesadas a texto mediante Faster Whisper, tratando de simular de la manera más fiel posible el tipo de entrada que el sistema recibiría en un entorno real.

Para construir nuestro conjunto de entrenamiento según la arquitectura presentada en [25], nuestro *dataset* debe incluir:

1. Un mensaje de sistema: Al igual que en Zero-Shot, define el rol general del modelo y sus instrucciones para la extracción. Añadimos este *prompt* en todos los ejemplos del *dataset*, el cual proporciona las reglas fundamentales para el LLM. Con el fin de comparar y ser capaces de apreciar los resultados de forma más clara, utilizaremos el mismo mensaje de sistema que utilizábamos en las pruebas Zero-Shot.
2. Un mensaje de usuario: El cual contiene la transcripción de la consulta médica que el código debe procesar, junto con una instrucción clara de qué hacer sobre ella. Debemos de ser mucho más breves y explícitos que en el *prompt* usuario que generamos anteriormente en Zero-Shot. Es decir, algo como: “Procesa el siguiente texto: [nuestra transcripción] y devuelve la salida formato JSON”.
3. Un mensaje de asistente: El cual representa la salida esperada del modelo para la transcripción que hemos introducido. Es necesario que este

mensaje incluya la salida JSON estructurada en el formato exacto que deseamos que el modelo aprenda a generar (*ner_esperado*).

Estos mensajes se combinan para formar una secuencia de conversación para cada ejemplo. Realizamos el entrenamiento con un *dataset* de 60 ejemplos. La mayoría de ellos simulando el resultado esperado, utilizando en la mayoría de los casos transcripciones sintéticas simuladas para la realización del proyecto. También tratamos de añadir dentro del conjunto de ejemplos sobre cómo actuar cuando no encuentre una o más categorías.

En la Figura 31, mostramos algunos de los ejemplos de entrenamiento utilizados dentro del dataset:

```
training_examples = [
    {
        "transcripcion": "Hola, tengo la piel muy roja y me pica mucho. Sobre
        "ner_esperado": {
            "sintomas": ["piel roja", "picazón", "escamas plateadas"],
            "enfermedad": ["psoriasis"],
            "tratamiento": ["crema tópica", "exposición solar controlada"]
        }
    },
    {
        "transcripcion": "Doctor, he venido porque tengo mucha cansansio y di
        "ner_esperado": {
            "sintomas": ["dificultad para respirar", "dolor de pecho"],
            "enfermedad": [],
            "tratamiento": ["pruebas de corazón"]
        }
    },
    {
        "transcripcion": "Hola, ¿podrías ayudarme a configurar mi nueva impre
        "ner_esperado": {
            "sintomas": [],
            "enfermedad": [],
            "tratamiento": []
        }
    }
]
output_dataset_file = "dataset.jsonl"
```

Figura 31. Algunos ejemplos del dataset de entrenamiento

En el primer ejemplo tenemos una transcripción sintética similar a la que utilizamos para el Zero-Shot. Introducimos en la transcripción toda la información clínica que buscamos, la cual extraemos y añadimos en el NER (reconocimiento de entidades nombradas).

Con el segundo ejemplo, tratamos de hacer ver al modelo que solo debe extraer y clasificar la información nombrada en la transcripción, y que no debe inventar texto. En el ejemplo de la imagen, podría deducir que la “enfermedad” puede ser estrés o ansiedad, pero no debe añadirlo porque no se ha mencionado explícitamente en la transcripción.

Por último, también enseñamos al modelo a dejar en blanco todos los campos si la transcripción no tiene nada que ver con el ámbito clínico o no menciona nada relacionado con las categorías deseadas.

Una vez tenemos preparado nuestro JSON, lo transformaremos al formato JSONL (JSON Lines), el cual es ampliamente utilizado por las librerías de entrenamiento de LLMs. Este proceso se realiza escribiendo cada variable *messages* (la cual incluye el *prompt* de usuario y sistema) a una nueva línea del archivo *dataset.jsonl*. Lo vemos más detenidamente dentro del código mostrado en la Figura 32:

```
with open(output_dataset_file, "w", encoding="utf-8") as f:
    for example in training_examples:
        # El rol de usuario contiene la instrucción y el texto a procesar
        user_content = f"""
        Ahora, procesa el siguiente texto:\n
        Texto: \"{example['transcripcion']}\"\n
        Salida JSON:"""

        # El rol de asistente contiene la salida JSON esperada:
        assistant_content = f"```json\n{json.dumps(example['ner_esperado'],
                                                   ensure_ascii=False,
                                                   indent=2)}\n```"

        # Construye la conversación para este ejemplo
        messages = [
            {"role": "system", "content": SYSTEM_PROMPT},
            {"role": "user", "content": user_content},
            {"role": "assistant", "content": assistant_content}
        ]

        # Escribe el JSON de la conversación en una nueva línea
        f.write(json.dumps({"messages": messages},
                           ensure_ascii=False) + "\n")

print(f"""Dataset '{output_dataset_file}' creado
          con {len(training_examples)} ejemplos.""")
print("Ahora puedes usar este archivo para el fine-tuning.")
```

Figura 32. Datos de entrenamiento formateados a JSONL

Para cada 'ejemplo' dentro de la cadena 'ejemplos de entrenamiento', añadimos los mensajes de contexto usuario, sistema y asistente. Esta creación y formateo del dataset de entrenamiento se trata de un paso fundamental. Además, la calidad y variedad de los ejemplos se verá reflejada en la capacidad del modelo para aprender y generalizar la tarea que le requerimos.

Con nuestro dataset ya preparado, es momento de integrarlo en nuestro modelo LLM.

4.4.4 Fine-tuning y LORA

Una vez tenemos el *dataset* de entrenamiento preparado, el siguiente paso será realizar el proceso de *fine-tuning* del modelo Llama-3.1-8B-Instruct utilizando la técnica LoRA (Low-Rank Adaptation of Large Language Models) y la

cuantización a 4 bits (QLoRA). Esta combinación de herramientas, junto con el entorno de desarrollo de Google Colab, nos permitirán adaptar eficazmente el modelo a la tarea específica de NER con nuestros recursos computacionales actuales.

Para comentarlos con más detenimiento, podemos separar el proceso de entrenamiento en cuatro pasos diferenciados:

1. **Configuración del Modelo Base y Cuantización (QLoRA):**

Para manejar un modelo del tamaño de Llama 3 en entornos de VRAM limitada decidimos utilizar la técnica QLoRA, la cual cuantiza los pesos del modelo base a 4 bits, lo cual reduce drásticamente el consumo de memoria de la GPU. Aunque la carga de trabajo se reduce aproximadamente a la mitad, la eficiencia y capacidad del modelo no se ve proporcionalmente afectada.

Llevamos a cabo la configuración de cuantización con una de las librerías que más problemas de compatibilidad presentó en nuestro entorno de desarrollo: *BitstAndBytesConfig*, de *Transformers*. Para su configuración debemos indicarle el tipo de cuantización y el tipo de datos de cómputo. A continuación, en la Figura 33, podemos ver con más detenimiento cada uno de estos parámetros:

```
#Configuramos BitsAndBytes:
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4", # Formato de cuantización NormalFloat4
    bnb_4bit_compute_dtype=torch.bfloat16, #Tipo de datos para cómputo
    bnb_4bit_use_double_quant=False, # No usamos doble cuantización
)
print(f"Cargando modelo base {model_id} con QLoRA...")
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    quantization_config=bnb_config, # configuración de cuantización
    device_map="auto"               # GPU si está disponibel
)
model.config.use_cache = False # Desactivar caché durante el entrenamiento
model.config.pretraining_tp = 1 # Configuración específica Llama 3
model = prepare_model_for_kbit_training(model) # IMPORTANTE para QLoRA

# Cargamos el tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name)`
# Llama no tiene pad token por defecto, usamos el EOS token:
tokenizer.pad_token = tokenizer.eos_token
# Importante para que el padding no interfiera con la generación:
tokenizer.padding_side = "right"
```

Figura 33. Configuración del modelo base y cuantización (QLoRA)

Comenzamos estableciendo los parámetros de BitsAndBytes, los cuales se encargarán de la cuantización del modelo. Dentro del entorno de Google Colab hemos encontrado constantemente problemas de compatibilidad con las versiones de BitsAndBytes, Accelerate y PyTorch. Dentro del anexo incluiremos información sobre la versión empleada de cada una de estas bibliotecas (7.5). Establecemos un tipo de cuantización *nf4 (normal float* de 4 bits*)* y de nuevo configuramos la realización de cálculos en el formato recomendado *bfloat16* (ver *[26]).*

Una vez tenemos *BitsAndBytes* configurado, cargamos el modelo ya pre-cuantizado y lo preparamos para el entrenamiento con adaptadores LoRA con la función *prepare_model_for_kbit_training().* Por último, cargamos el tokenizer predeterminado del modelo y definimos el *padding.* El *padding* será el relleno, cuya función es aumentar la longitud de las secuencias de tokens que no alcancen el número máximo añadiendo un token especial. Llama 3.1 no tiene token de padding explícito por defecto, por lo que utilizaremos el token de fin de

secuencia (EOS – *End Of Sequence*), además de especificar que se rellene por el lado derecho.

2. Configuración de LoRA

LoRA nos permite realizar el entrenamiento sobre un pequeño conjunto de datos en lugar de tener que entrenar todo el modelo, lo cual reduce la carga computacional y la memoria requerida. Podremos configurar ciertos parámetros del entrenamiento para regular la capacidad de aprendizaje y el impacto de los cambios.

```
# Estos parámetros controlan cómo se "aprenderán" los adaptadores LoRA
lora_config = LoraConfig(
    # Escala de los pesos LoRA
    lora_alpha=16,
    # Dropout para regularización durante el entrenamiento de LoRA
    lora_dropout=0.1,
    # Rango de la matriz LoRA (mayor R = más aprendizaje, más VRAM)
    r=64,
    # No aplicar LoRA a los sesgos (bias)
    bias="none",
    # Tipo de tarea: Modelos de Lenguaje Causal (generación de texto):
    task_type="CAUSAL_LM",
    # Módulos del modelo base donde se aplicarán los adaptadores LoRA:
    target_modules=[
        "q_proj", "k_proj", "v_proj",
        "o_proj", "gate_proj", "up_proj",
        "down_proj",
    ],
)
```

Figura 34.Parámetros de configuración LoRA

El valor de *lora_alpha* define cuánto “peso” tendrá nuestro entrenamiento sobre el preentrenamiento del modelo. Por otro lado, el valor *r* controla la capacidad de aprendizaje de los adaptadores. Con un valor *r* más alto, los adaptadores podrán aprender transformaciones más complejas y serán mucho más específicos para la tarea. Los valores de *r* más comunes oscilan entre 8 y 256, sin embargo, los valores de *r* más altos también tienen asociado un mayor uso de VRAM.

Por otro lado, *lora_dropout* será un valor que regule el modelo para evitar su sobreentrenamiento, al cual hemos asignado su valor más típico (un 10%). La función de la variable de sesgo (*bias)* informa al modelo si usaremos nuestros adaptadores LoRA a los vectores de sesgo de las capas del modelo (es decir, a los vectores que se suman en la fase de atención tras la multiplicación de los pesos) y en qué capas lo usaremos. Los valores posibles para la variable *bias* indicarán directamente al modelo en qué capas debe aplicar estos adaptadores: *none* (en ninguna capa), *all* (en todas las capas) o *lora_only* (solo en las capas que se aplica lora). En nuestro caso, optamos por no aplicarlas.

Por último, podemos ver el array *target_modules*, el cual contiene las distintas capas del modelo base en las que aplicaremos nuestros adaptadores LoRA. Podemos ver las proyecciones de los mecanismos de autoatención Q (consulta), K (clave) y V (valor), además de *o_proj* el cual será la proyección de la salida de la fase de autoatención. Además, añadimos las proyecciones de *gate, up* y *down*, las cuales son capas que forman parte de la *FeedForward*, también conocida como la capa MLP (*Multi-Layer Perception).* Incluir todas estas capas permite a LoRA modificar las transformaciones clave del modelo, tanto en la atención como en la posterior red de avance.

En el punto anterior dejamos el modelo preentrenado cargado, cuantizado y tokenizado para el entrenamiento, y en este punto hemos establecido el tipo y rango de entrenamiento. Lo único que resta para empezar el entrenamiento será definir sus argumentos.

3. **Argumentos de Entrenamiento:**

Llevaremos a cabo el ajuste de los argumentos de entrenamiento con la función *TrainingArguments()*. En la Figura 35, podemos ver la lista de argumentos completa que utilizaremos para realizar el *fine-tuning* con LoRA, junto con una pequeña descripción de éstos.

```
# Argumentos de Entrenamiento
# Directorio donde se guardarán los checkpoints y logs
output_dir = "./results"
training_arguments = TrainingArguments(
    output_dir=output_dir,
    # Número de veces que el entrenador verá todo el dataset:
    num_train_epochs=3,
    # Número de ejemplos por batch por GPU:
    per_device_train_batch_size=2,
    # Pasos para acumular gradientes antes de actualizar pesos:
    gradient_accumulation_steps=2,
    # Optimizador (optimizado para memoria con 8-bit AdamW):
    optim="paged_adamw_8bit",
    # Guardar checkpoint cada X pasos:
    save_steps=25,
    # Loggear progreso cada X pasos:
    logging_steps=25,
    # Tasa de aprendizaje:
    learning_rate=2e-4,
    # Regularización para evitar overfitting:
    weight_decay=0.001,
    # Usar float16 (media precisión):
    fp16=False,
    # Usar bfloat16:
    bf16=True,
    # Clipping de gradiente para estabilidad:
    max_grad_norm=0.3,
    # Si nuestros pasos son -1, entrena por num_train_epochs:
    max_steps=-1,
    # Proporción de pasos para warm-up de la tasa de aprendizaje:
    warmup_ratio=0.03,
    # Agrupa ejemplos de longitud similar para mayor eficiencia:
    group_by_length=True,
    # Tipo de scheduler de tasa de aprendizaje:
    lr_scheduler_type="cosine",
    # Visualizar el progreso del entrenamiento en TensorBoard:
    report_to="tensorboard",
)
```

Figura 35. Argumentos de Entrenamiento con LoRA

En este punto nos centraremos en cuatro de estos parámetros de entrenamiento, los cuales posiblemente sean los más relevantes y los que definirán el éxito o fracaso de éste:

- ***Num_train_epochs:*** Determina el número de veces que el entrenador iterará sobre el dataset de entrenamiento completo. Es fundamental hacer pruebas hasta encontrar el valor óptimo. Un valor de *epochs* demasiado bajo puede resultar en que el modelo no sea capaz de seguir las instrucciones, por el contrario, si introducimos un valor de *epochs* demasiado alto podría llegar a "memorizar" los ejemplos del entrenamiento y no generalizar bien cuando introduzcamos nuevos datos. Teniendo en cuenta el tamaño y las funciones NER del dataset, con 3 *epochs* fue suficiente para permitir que el modelo aprendiera los patrones de extracción.
- ***Learning_rate:*** La tasa de aprendizaje podríamos definirla como la magnitud de ajustes que el modelo aplica a sus parámetros internos (pesos y sesgos) en cada paso del aprendizaje. Éste es posiblemente el parámetro más importante en el entrenamiento de los modelos *Deep Learning*. Si definimos una tasa de aprendizaje demasiado alta, el modelo trata de abarcar demasiada información en poco tiempo y es complicado que "encuentre" el punto óptimo. Si la tasa de aprendizaje es demasiado baja, el modelo hará ajustes tan pequeños y cautelosos que podrían alargar excesivamente el entrenamiento, además de que aumenta la probabilidad de entrar en bucle buscando una respuesta óptima. Establecimos una tasa de aprendizaje de 2×10^{-4} (0.0002), un valor común y efectivo para el *fine-tuning* de LLMs.
- ***Per_device_training_batch*** y ***gradient_accumulation_steps:*** Los dos parámetros trabajan juntos para determinar el tamaño de *batch* efectivo que ve el modelo en cada actualización de pesos. El *batch* es una pequeña porción del dataset que utilizamos para realizar una actualización (por cada iteración) de los parámetros del modelo. Realizamos el entrenamiento sobre estos pequeños conjuntos, no sobre el dataset completo. El parámetro *Per_device_training_batch* indica el número de ejemplos que se procesan concurrentemente en la GPU, en entornos con limitaciones de VRAM es necesario reducirlo todo lo posible (en nuestro caso lo establecimos en 2). Por otro lado, el *gradient_accumulation_steps* permite simular un batch de mayor tamaño. Cuando entrenamos un *batch*, en vez de devolver inmediatamente su

valor (un vector gradiente), lo acumulamos tantas veces como hayamos establecido en *gradent_accumulation_steps* y entrenamos al siguiente batch, esto nos permite realizar una única actualización con todos los parámetros de golpe. En nuestro caso, entrenamos 2 *batch* simultáneamente y hemos establecido una acumulación de gradiente de 2, por lo que nuestro '*batch* efectivo' sería de 4. Con esto, conseguimos el mismo efecto que si usásemos un *batch* de mayor capacidad, pero con menor requerimiento de GPU.

4. **Ejecución del Entrenamiento:**

El proceso de entrenamiento del modelo lo llevaremos a cabo mediante el ***SFTTrainer*** *(Supervised Fine-Tuning Trainer)* de la librería ***trl***. Este *trainer* está específicamente diseñado para facilitar el entrenamiento de modelos de lenguaje. Entre otras funciones, SFTTrainer permite automatizar el formateo del dataset y aplicar la configuración LoRA sobre el modelo base cuantizado, además de gestionar el entrenamiento en su totalidad. De nuevo, nos ceñiremos a la estructura recomendada en [27].

En la Figura 36 podemos ver el código empleado para inicializar y ejecutar el modelo, así como su posterior guardado.

```python
#Cargamos nuestro Dataset de entrenamiento:
print("Cargando dataset 'dataset.jsonl'...")
dataset = load_dataset("json", data_files="dataset.jsonl", split="train")

#Inicializamos el SFTTrainer
trainer = SFTTrainer(
    model=model, #con el modelo preentrenado
    train_dataset=dataset, #nuestro dataset
    peft_config=lora_config, #la configuración lora que definimos
    args=training_arguments, #y nuestros argumentos de entrenamiento
)

print("\nIniciando fine-tuning del modelo...")
trainer.train() #Entrenamos al modelo

print("\nFine-tuning completado.")

#Y guardamos en memoria los adaptadores LoRA entrenados y el tokenizer.
new_model_path = "./llama-3.1-8b-instruct-med-ner-lora"
trainer.model.save_pretrained(new_model_path)
tokenizer.save_pretrained(new_model_path)
print(f"Adaptadores LoRA y tokenizer guardados en: {new_model_path}")
```

Figura 36. Fine-tuning LoRA con SFTTrainer

Para inicializar el SFTTrainer, haremos uso de todas las piezas que preparamos en los puntos anteriores. Cargamos el modelo pre-entrenado, el *dataset* de entrenamiento, las configuraciones de LoRA y los argumentos de entrenamiento.

El proceso de *fine-tuning* inicia con la llamada al método *trainer.train()*. Mientras se ejecuta, el modelo va ajustando iterativamente los parámetros de los adaptadores LoRA. Para aprender a identificar y extraer correctamente la información en el formato JSON deseado, hará uso de los gradientes de la función de pérdida. La función de pérdida cuantifica las diferencias entre las predicciones del modelo y las respuestas correctas, indicando cuánto se está "equivocando". El progreso del entrenamiento se registra y puede ser monitorizado para facilitar su supervisión y seguimiento, en la Figura 37 podemos ver los resultados del entrenamiento:

Figura 37. Resultados tras realizar fine-tuning con LoRA

Además de mostrar a tiempo real cómo realiza cada una de las etapas, los datos obtenidos al realizar el *fine-tuning* nos ofrecen una visión general de la capacidad y eficiencia del modelo. Los capturamos y sintetizamos en una tabla para analizar el rendimiento obtenido.

Medida	Valor
Tiempo de ejecución	1698.81 segundos
Ejemplos/Segundo	0.106
Pérdida (Loss) Final	0.6336
Precisión de Token Promedio	0.8992 (89.92%)
Pasos de entrenamiento/segundo	0.026
Epochs completadas	3
Tamaño efectivo del b*atch*	4

Tabla 3. Resultados del fine-tuning con LoRA (dataset de 60 ejemplos)

Con esta configuración de parámetros obtuvimos resultados muy prometedores. Comentamos brevemente los resultados obtenidos:

- El **tiempo de ejecución** es razonable teniendo en cuenta que el entrenamiento se realiza sobre un modelo de 8 mil millones de parámetros. Aun así, es algo lento considerando que tardamos con el fine-tuning aproximadamente **30 minutos**, sin contar con la carga del modelo,

del tokenizador, la transcripción, etc. En esta parte se hace más notable la necesidad de poder guardar el modelo y ejecutarlo dentro de un entorno local.

- La **pérdida** (*train loss*) cuantifica la diferencia entre las predicciones del modelo y los valores verdaderos. En esta ocasión, obtenemos una pérdida de **0.63,** el cual es un valor muy bueno. En el contexto de nuestro modelo y datos de entrenamiento, un valor de pérdida tan bajo significa que el modelo es capaz de predecir con alta precisión las entidades que debe extraer y el formato con el que debe presentarlas (JSON)
- La **precisión de token promedio** *(mean token accuracy)* mide la precisión con la que el modelo predice cada token individual en la secuencia de salida. Obtenemos una precisión de token promedio de 89.92%, lo cual es un valor muy alto para tratarse de un *fine-tuning* inicial de 60 ejemplos a un modelo preentrenado con más de 8.000 millones.
- El valor que obtenemos de los **ejemplos / segundo** (train samples per second) indica la eficiencia de la configuración. Aunque **0,106** ejemplos por segundo (es decir, procesar el 10,6% de un ejemplo del dataset por segundo) pueda parecer muy bajo en valores absolutos, es un reflejo del coste computacional que tiene entrenar un modelo tan grande como Llama 3.1-8B. Mantener este valor por encima de **0,1** en un entorno como Google Colab significa que nuestra configuración de acumulación de gradientes está ayudándonos a maximizar el uso de GPU eficiente.
- Los **pasos de optimización / segundo (***train steps per second)* indica el número de veces que el modelo actualiza sus pesos por segundo. Dado que utilizamos un valor de *gradient_accumulation_steps* de 2, tendremos 2 batches con 2 ejemplos cada uno; es decir, un **tamaño efectivo del Batch** de **4** ejemplos antes de realizar una única actualización de pesos. En nuestro caso obtuvimos un valor de *train steps / second* de **0.026,** un valor normal y esperado para esta configuración. Debido al cambio de tamaño de los pasos de acumulación de gradiente, tendremos actualizaciones menos frecuentes pero que se basan en una mayor cantidad de información de datos.

En resumen, los resultados tras realizar el fine-tuning nos demuestran una alta precisión de token y pérdidas bajas, lo cual indica que el modelo está aprendiendo exitosamente a realizar la tarea de extracción de información clínica en formato JSON. Guardamos el tokenizador y los adaptadores LoRA resultantes del *fine-tuning* y procedemos a la extracción de datos.

4.4.5 Extracción de datos en formato JSON

En esta última sección nos enfocaremos en la **inferencia del modelo** y la **extracción de entidades clínicas en formato JSON** a partir de texto no estructurado. Para cargar nuestro modelo no basta con llamarlo dentro de una variable como hacíamos en puntos anteriores, si no que tendremos que cargar el modelo preentrenado e "inyectarle" el tokenizer del entrenamiento anterior además de nuestros adaptadores LoRA.

No analizaremos de nuevo cómo hacer la carga de un modelo preentrenado puesto que el procedimiento será idéntico al que veíamos en la Figura 33. La diferencia es que tras cargar el modelo en la variable *base_model*, cargaremos nuestros propios parámetros para el tokenizer (*lora_model_path*), como podemos ver en la Figura 38:

```
lora_model_path = "./llama-3.1-8b-instruct-med-ner-lora"

# Carga del tokenizer (desde los adaptadores guardados para consistencia)
tokenizer = AutoTokenizer.from_pretrained(lora_model_path)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"

# Carga y adjunción de los adaptadores LoRA al modelo base
print(f"Cargando adaptadores LoRA desde {lora_model_path} y adjuntándolos...")
model = PeftModel.from_pretrained(base_model, lora_model_path)
print("Adaptadores LoRA adjuntados al modelo base.")

# Se establece el modelo en modo de evaluación para una inferencia consistente
model.eval()

print("¡Modelo fine-tuneado listo para inferencia!")
```

Figura 38. Carga del LLM entrenado

Con el modelo preentrenado cargado y el *autotokenizer* completado, hacemos uso de la biblioteca **peft** para la carga mediante *PeftModel.from_pretrained()* de los adaptadores LoRA. Las pequeñas matrices de peso específicas entrenadas durante el *fine-tuning* se "inyectan" al modelo preentrenado. Esto permite al modelo beneficiarse de los miles de millones de parámetros del modelo base mientras dirige su comportamiento hacia la tarea especializada que requerimos.

Finalmente, hacemos uso de *model.eval()*. Ésta es una función que activa el **modo evaluación** para deshabilitar funcionalidades propias del entrenamiento (como el *Dropout*) e indicar al modelo que buscamos una inferencia consistente y predecible.

Una vez el modelo está cargado y configurado, el siguiente paso es la **inferencia,** fase donde el modelo procesa una entrada de texto y genera la información estructurada deseada en formato JSON. Nuestra función de inferencia *(run_inference())* también comparte gran parte de código con el apartado Zero-Shot, puesto que utilizaremos los mismos prompts (usuario y sistema) que veíamos en la Figura 27 y Figura 28, así como los mismos parámetros para la generación de respuesta que veíamos en la Figura 29.

Podemos ver la función de inferencia más en detalle en la Figura 39. Recortamos los *prompt* usuario y sistema para mostrar el código más limpio, pero mantenemos el resto del código compartido para incidir en que utilizaremos los mismos parámetros y configurarción que en la ejecución del modelo preentrenado.

```python
def run_inference(text_input, model, tokenizer):
    system_prompt = "(....)"
    user_prompt = f"(...)"
    messages = [
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": user_prompt}
    ]

    # Aplicamos la plantilla del tokenizer para obtener los  input id
    input_ids = tokenizer.apply_chat_template(
        messages, #nuestros prompts
        tokenize=True,
        add_generation_prompt=True, # Importante para Llama 3.1 Instruct
        return_tensors="pt" #devolvemos pydub tensors
    ).to(model.device)

    # Generamos la respuesta
    print("\nGenerando respuesta...")
    outputs = model.generate(
        input_ids, #nuestros input_ids
        max_new_tokens=500, #maximo de tokens generados
        do_sample=True,
        top_p=0.9, #nucleus sampling
        temperature=0.7, #aleatoriedad
        num_return_sequences=1
    )

    # Decodificamos la salida
    response_text = tokenizer.decode(outputs[0,
                                     input_ids.shape[-1]:],
                                     skip_special_tokens=True)
    return response_text #y devolvemos la respuesta
```

Figura 39. Función de inferencia del modelo entrenado

Una vez tenemos nuestra respuesta, la etapa final del proceso implica el **procesamiento y validación de la salida generada por el modelo**, asegurando que el texto devuelto sea un JSON válido. Los LLM, a pesar de su alta precisión, a veces tienden a incluir texto adicional o cometer pequeños errores de formato que requieren un paso de limpieza antes de que la salida pueda ser utilizada por otros sistemas informáticos. Aplicaremos una función de *parseo* (del inglés *parsing*) sobre el json obtenido. Adjuntamos la función de limpieza dentro del anexo 7.4.

Resumiendo, para obtener la inferencia del modelo que hemos entrenado y previamente cargado, ejecutaremos nuestra función *run_inference*() introduciendo nuestro modelo entrenado, nuestro tokenizer cuantizado y nuestra transcripción del audio, como podemos ver en la Figura 40:

```
texto = transcripcion_final #TRANSCRIPCION DEL AUDIO
print(f"\n--- Extraemos la información clínica relevante ---")
output_1 = run_inference(texto, model, tokenizer)
#Presentamos la salida conforme sale del modelo:
print("Salida del modelo:\n", output_1)

#Limpiamos y estructuramos la salida:
print("\n\n--- Parseando y mostrando salidas en formato estructurado ---")
parse_and_print_json(output_1)
```

Figura 40. Ejecución de la Inferencia del modelo entrenado

Finalmente, para demostrar la capacidad del modelo con una situación práctica, aplicamos la función de inferencia sobre la transcripción que veíamos en la Figura 25.

En la Figura 41 podemos ver la salida de texto crudo del modelo:

```
Cargando modelo base meta-llama/Llama-3.1-8B-Instruct en 4 bits...
Loading checkpoint shards: 100% ████████████████████ 4/4 [01:19<00:00, 16.87s/it]
Cargando adaptadores LoRA desde ./llama-3.1-8b-instruct-med-ner-lora y adjuntándolos...
Adaptadores LoRA adjuntados al modelo base.
¡Modelo fine-tuneado listo para inferencia!

 Ejemplo sacado del dataset:

Generando respuesta...
Salida del modelo:
 ```json
{
  "sintomas": [
    "dolor de cabeza",
    "mareos"
  ],
  "enfermedad": [
    "cefaleas tensionales"
  ],
  "tratamiento": [
    "aplicación de frío",
    "analgésicos",
    "ibuprofeno"
  ]
}
```
```

Figura 41. Resultados tras la inferencia del modelo entrenado

Conseguimos cargar en memoria el modelo (con nuestros adaptadores LoRA), realizar el proceso de *tokenizacion,* realizar la inferencia y, tras realizar el *fine-tuning,* obtenemos una respuesta más refinada que la que obteníamos en la inferencia Zero-Shot.

También adjuntamos en la Figura 42 el resultado tras validar los datos y limpiarlos con técnicas de *parsing*.

```
Entidades NER extraídas (JSON parseado):
  Categoría: SINTOMAS
    - dolor de cabeza
    - mareos
  Categoría: ENFERMEDAD
    - cefaleas tensionales
  Categoría: TRATAMIENTO
    - aplicación de frío
    - analgésicos
    - ibuprofeno
--------------------------------------------------
```

Figura 42. Extracción de información clínica en formato JSON parseado

Hemos sido capaces de eliminar la mayoría de redundancia (que veíamos en la Figura 30). Y más importante aún, el modelo ha sido capaz de analizar y comprender el contexto de la conversación, eliminando "resonancia" de tratamientos. Esto es un resultado muy positivo, pues no solo demuestra su capacidad para extraer información médica relevante de la transcripción, sino que también muestra cierto grado de comprensión sobre la propia conversación.

4.4.6 Integración de LLMs Locales con Ollama (Alternativa de Despliegue)

Aunque el desarrollo y las pruebas principales con LLMs se llevaron a cabo en Google Colab (debido a sus recursos de GPU en la nube), es interesante considerar alguna alternativa de despliegue local que permita una mayor autonomía y control sobre los datos. En este contexto, **Ollama** nos pareció una opción interesante para gestionar y ejecutar los LLMs.

Ollama es una plataforma que facilita la descarga, configuración y ejecución de los LLMs. Permite ejecutarlos en un entorno local e interactuar con ellos a través de una API sencilla, algo necesario teniendo en cuenta los requisitos de privacidad que supondría la implementación del proyecto.

En una fase inicial del desarrollo exploramos la posible implementación del sistema utilizando Ollama. La interacción con el LLM local lo gestionamos a

través de la librería de Ollama en Python, permitiéndonos enviar *prompts* y recibir respuestas.

Al ejecutar el sistema de forma local, el sistema podría no ser capaz de soportarlo por falta de VRAM. Esto lo solucionamos segmentando el audio y transcribiéndolo secuencialmente. Después, pasamos cada fragmento de transcripción por el LLM y tratamos de localizar cualquier posible 'síntomas, 'edad' o 'tratamiento' dentro del texto. Al encontrarlo lo extrae del texto y lo guarda en un array, para después juntar todas las respuestas generadas y, volviendo a llamar al modelo, sintetizar la información y presentarla en forma de tabla. Por lo que necesitaríamos llamar al modelo a través de Ollama en dos ocasiones:

```
def extraer_info_transcripcion(texto_transcripcion):
    modelo_ollama = 'llama3.2:3b'
    pregunta = f"""
    Tu tarea es identificar y extraer SÍNTOMAS, ENFERMEDADES y TRATAMIENTOS
    mencionados en el texto. Debes listar estas entidades en formato JSON.
    Si no se encuentra una entidad de un tipo específico, esa clave debe estar
    ausente o ser una lista vacía.
    No incluyas ninguna otra información, explicaciones o texto adicional.
      Texto: "{input_text}"
        """
    try:
        respuesta = ollama.chat(model=modelo_ollama,
                                messages=[{'role': 'user',
                                           'content': pregunta}])
        print("Respuesta de Ollama:", respuesta['message']['content'])
        return respuesta['message']['content']
    except Exception as e:
        print(f"Error al obtener respuesta de Ollama: {e}")
        return ""
```

Figura 43. Extracción de información por segmentos utilizando Ollama

Para la **extracción de información por segmento** (*extraer_info_ segmento)* emplearemos una transcripción de texto y la enviaremos al LLM alojado localmente en Ollama. Como podemos ver en la Figura 43, en este caso utilizamos llama3.2 con 3B parámetros (tratando de reducir al máximo los requisitos computacionales). El prompt está diseñado para solicitar la extracción

de información específica (síntomas, enfermedad, tratamiento) de ese fragmento de texto, utilizando un formato de salida predefinido.

Y para la **síntesis y creación de la tabla final** (*sintetizar_y_crear_tabla*): Una vez hemos extraído la información de cada segmento, esta función se encarga de sintetizar y consolidar los datos. Como vemos en la Figura 44, las respuestas parciales que habíamos obtenido hasta el momento se combinan y las enviamos de nuevo al LLM local. El *prompt* final le indica cómo ordenar la información y presentarla en un formato tabular estructurado (Markdown). Eliminando duplicidades y consolidando los detalles relevantes.

```
def sintetizar_y_crear_tabla(info_segmentos_llm):
    modelo_ollama = 'llama3.2:3b'
    texto_combinado="\n".join(str(segmento) for segmento in info_segmentos_llm)
    pregunta_tabla = f"""A continuación te proporciono información extraída
                de diferentes segmentos de una transcripción médica.
                Cada segmento contiene información sobre los Síntomas,
                Enfermedad y Tratamiento del paciente. Si no se encontró
                información en un segmento, la respuesta estará vacía.

    Analiza toda esta información y sintetízala para crear una tabla final con
    la información. Evita añadir comentarios, simplemente haz la tabla e
    introduce la información de cada apartado (Síntomas, Enfermedad,
    Tratamiento). Evita las repeticiones, no pongas dos veces el mismo síntoma
    o la misma enfermedad.
    Información de los segmentos (respuestas del LLM):
    {texto_combinado}

    Devuelve la tabla en formato Markdown.
    """

respuesta = ollama.chat(model=modelo_ollama,
                        messages=[{'role': 'user',
                                   'content': pregunta_tabla}])
return respuesta['message']['content']
```

Figura 44. Síntesis y creación de tabla formato Markdown utilizando Ollama

A pesar de que la integración con Ollama fue un éxito y las pruebas preliminares mostraron resultados prometedores, nos encontramos limitaciones significativas a la hora de refinar los resultados. El principal problema radicó en la capacidad

de GPU disponible localmente, que reducía las posibilidades a la hora de ejecutar LLMs de mayor tamaño. Aunque obtuvimos resultados funcionales, no mostraban el nivel de calidad y robustez que buscábamos, y la fase de *fine-tuning* y experimentación intensiva hubiese sido inviable con la memoria virtual disponible.

Por estas razones, tomamos la decisión de migrar el entorno de desarrollo a Google Colab. Esta plataforma nos permitió experimentar con modelos de LLM más sofisticados y de manera más eficiente, aunque también presentó mayores problemas de compatibilidad entre las versiones de herramientas y bibliotecas utilizadas.

4.5 Principales resultados y validación

Con el sistema completamente funcional, usaremos este apartado para poner a prueba el modelo. Para justificar la configuración del *fine*-tuning que utilizamos, variaremos los parámetros de entrenamiento y observaremos los resultados. Además, expondremos al modelo a distintos tipos de transcripción y observaremos tanto sus puntos fuertes como las áreas a mejorar.

4.5.1 Parámetros de entrenamiento

En la Tabla 3 enumerábamos en una tabla los valores obtenidos tras realizar el *fine-tuning* con QLoRA y un dataset de 60 ejemplos. Probamos a ejecutar el entrenamiento variando algunos de los parámetros principales para poder observar y medir el impacto de éstos.

La primera prueba que deberíamos realizar es variar y observar el impacto del **tamaño del *dataset***. Para realizar nuestros experimentos utilizamos un conjunto de 60 ejemplos, así que podemos comparar nuestros resultados con los que obtendríamos recortando el *dataset* a solo 20 ejemplos.

	60 Ejemplos Dataset	20 Ejemplos Dataset
Pérdida (Loss) Final	0.6336	1.1824
Precisión de Token Promedio	0.8992 (89.92%)	0.7578 (75.78%)
Tiempo de Ejecución	1698.81 segundos	587.52 segundos
Ejemplos/Segundo	0.106	0.102
Actualización de pesos	0.026	0.026
Épocas Completadas	3.0	3.0
Tamaño Efectivo del Batch	4	4

Tabla 4, Resultados del entrenamiento disminuyendo el dataset sintético

Al reducir el tamaño del conjunto de entrenamiento a un número tan limitado, aumentamos muy significativamente el valor de pérdida final. Es normal, puesto que ésta se calcula comparando los resultados que obtiene el modelo con los esperados del *dataset*, y es difícil generalizar un conjunto tan pequeño de parámetros dentro de un modelo preentrenado de 8.000 millones.

El tiempo de ejecución se reduce de forma proporcional al número de ejemplos que hemos eliminado, es decir, a un tercio del tiempo que consume el modelo original. La tasa de ejemplos y los pasos de optimización (es decir, la actualización de pesos) conservan los mismos valores.

Aunque el tiempo de ejecución resulta mucho más razonable, vemos que la pérdida de eficacia es demasiado significativa. Al realizar la inferencia del modelo, obteníamos resultados muy similares a las pruebas Zero-Shot.

El siguiente paso lógico fue aumentar el tamaño del dataset para comprobar si la reducción de la pérdida resulta proporcional. Obtuvimos los siguientes resultados, representados en la Tabla 5:

	60 Ejemplos Dataset	100 Ejemplos Dataset
Tamaño Efectivo del Batch	4	4
Pérdida (Loss) Final	0,6336	0,6018
Precisión de Token Promedio	0.8992 (89.92%)	0.8997 (89.97%)
Tiempo de Ejecución	1698.81 segundos	3072.14 segundos
Ejemplos/Segundo	0,106	0,106
Pasos de Optimización/Segundo	0,026	0,026
Épocas Completadas	3.0	3.0

Tabla 5. Resultados del entrenamiento aumentando el dataset sintético

Sorprende observar que tanto la pérdida final como la precisión de token promedio apenas han mejorado al aumentar el conjunto de datos de entrenamiento. Esto puede ser debido a la propia calidad y variedad de los datos, los cuales hemos simulado tratando de abarcar la mayor cantidad de escenarios posibles, pero no pueden igualar a la calidad de un entrenamiento constante y real. Los resultados con el dataset de 100 ejemplos siguen siendo excelentes, pero ligeramente inferiores a los obtenidos con nuestro dataset de 60 ejemplos. Además, vemos un aumento considerable del tiempo, de nuevo aproximadamente proporcional a la cantidad de datos nuevos que añadimos. Por lo que concluimos que 60 ejemplos será un número adecuado para las condiciones y recursos de nuestro proyecto.

La siguiente prueba que podría resultar interesante consiste en cambiar el valor de **gradiente acumulado**, el cual recordamos que acumula los resultados del *batch* (porciones del *dataset* completo) y los pasa al modelo al mismo tiempo. En nuestro caso, acumulábamos dos resultados y, de igual forma, también ejecutábamos el *batch* con dos ejemplos, por lo que utilizábamos un ‘batch efectivo’ de 4. Si no utilizamos el gradiente de acumulación (es decir, ponemos

su valor a 1) y realizamos el entrenamiento utilizando únicamente el *batch* obtenemos los siguientes valores:

	grad_accum=2	grad_accum=1
Tamaño Efectivo del Batch	4	2
Pérdida (Loss) Final	0.6336	0.6267
Precisión de Token Promedio	0.8992 (89.92%)	0.9007 (90.07%)
Tiempo de Ejecución	1698.81 segundos	2087.39 segundos
Ejemplos/Segundo	0.106	0.098
Frecencia de actualización de pesos	0.026	0.048
Épocas Completadas	3.0	3.0

Tabla 6. Entrenamiento del modelo con diferentes valores de Gradiente Acumulado

Lo primero que observamos es que la pérdida final será ligeramente superior sin utilizar el gradiente acumulado, aunque se trata de una diferencia despreciable. En cambio, al disminuir el número del gradiente acumulado la ejecución fue aproximadamente un 25% más lenta, resultando en una demora extra de aproximadamente 6 minutos.

Además de eso, podemos ver que la frecuencia de actualización de pesos ha subido drásticamente. Esto es debido a que anteriormente acumulábamos dos resultados del *batch* antes de pasarlos al modelo, ahora vamos pasándolos uno a uno, por lo que debemos hacer aproximadamente el doble de pasos. Como consecuencia, también reducimos el número de ejemplos que procesa el modelo por segundo.

Aumentar el número de acumulación de gradientes excede las capacidades que nos ofrecen tanto Google Colab como nuestro entorno local, por lo que no probamos con distintos valores, aunque podemos asumir que seguiríamos

mejorando el tiempo de ejecución a costa la VRAM del sistema y una pequeña pérdida de precisión.

4.5.2 Pruebas del sistema

Para evaluar la capacidad del modelo en diversos escenarios de uso, realizamos inferencias sobre un aserie de transcripciones con características variadas. Este análisis cualitativo se enfoca en demostrar la robustez, precisión y capacidad de generalización del sistema ante diferentes tipos de entrada de texto clínico.

A continuación, se muestran los resultados de la extracción para cada caso representativo. Para no saturar la memoria, utilizamos el modelo preentrenado para generar un resumen de la transcripción antes de mostrar los datos.

Ejemplo 1. Audio de consulta completa

En el ejemplo de la Figura 45 simulamos una consulta parecida a la que utilizamos en los anteriores puntos de la memoria, donde la información clave sobre el estado del paciente se presenta de forma clara y directa, permitiendo una extracción precisa de las entidades.

```
Resumen autogenerado de la transcripción:
El paciente refiere dolor abdominal persistente desde hace una semana,
acompañado de fiebre baja. El médico, tras la exploración, diagnostica
apendicitis aguda y recomienda cirugía de urgencia para su tratamiento.

Entidades NER extraídas (JSON parseado):
   Categoría: SINTOMAS
     - dolor abdominal
     - fiebre
   Categoría: ENFERMEDAD
     - apendicitis aguda
   Categoría: TRATAMIENTO
     - cirugía de urgencia
----------------------------------------
```

Figura 45. Resultados de inferencia del Ejemplo 1. Transcripción clínica completa

Este resultado muestra la alta precisión del modelo a la hora de extraer y clasificar entidades médicas en un escenario concreto, donde la información es explícita y es sencillo categorizar las entidades extraídas. El éxito al extraer los síntomas, enfermedad y tratamiento propuesto valida la capacidad fundamental del modelo para procesar lenguaje clínico directo.

Ejemplo 2. Escenario sin información clínica

Este ejemplo es vital para validar que el modelo no sufre “alucinaciones” de información médica cuando el texto de entrada carece por completo de ella. Para comprobar su robustez, introducimos la dedicatoria de Miguel de Cervantes al Duque de Béjar (Figura 46):

```
Resumen autogenerado de la transcripción:
En este texto, Miguel de Cervantes Saavedra dedica su obra
"El ingenioso Hidalgo Don Quijote de la Mancha" al Duque
de Béjar, quien ostenta varios títulos nobiliarios. Cervantes elogia al
Duque por su aprecio hacia los libros y su apoyo
a las artes.

Entidades NER extraídas (JSON parseado):
   Categoría: SINTOMAS
   Categoría: ENFERMEDAD
   Categoría: TRATAMIENTO
--------------------------------------------------
```

Figura 46. Resultados de inferencia del Ejemplo 2. Escenario sin información clínica

La generación de listas vacías para todas las categorías en ausencia de cualquier contenido clínico relevante es una clara demostración del control del modelo. Este resultado valida su capacidad para discernir cuándo no hay información médica presente, evitando generar respuestas erróneas o irrelevantes, lo que es fundamental para la fiabilidad del sistema en aplicaciones prácticas.

Ejemplo 3. Categorías sin rellenar

Buscando los límites en la capacidad del modelo para discernir y no extraer información que se presenta como posibilidad o especulación, obtuvimos resultados realmente sorprendentes (Figura 47).

```
Resumen autogenerado de la transcripción:
El paciente describe una opresión en el pecho y dificultad
para respirar después de esfuerzos. El doctor aún no ha
llegado a un diagnóstico, pero ha mencionado que 'podrían ser
problemas cardíacos' y ha solicitado pruebas adicionales.

Entidades NER extraídas (JSON parseado):
   Categoría: SINTOMAS
     - opresión en el pecho
     - dificultad para respirar
   Categoría: ENFERMEDAD
   Categoría: TRATAMIENTO
-----------------------------------------
```

Figura 47. Resultados de inferencia del Ejemplo 3. Categorías sin rellenar

Aunque se mencionan explícitamente “problemas cardiacos” o “algo del corazón” dentro de la transcripción, es capaz de discernir que es una mención de una posible condición y no un diagnóstico confirmado. La precisión es fundamental dentro del ámbito clínico, es preferible omitir información no confirmada a la inclusión de datos inciertos, por lo que estos resultados fueron muy positivos.

Ejemplo 4. Menciones clínicas sin contexto

Para mostrar las limitaciones del modelo en la comprensión contextual, tratamos de llevar al límite sus capacidades para identificar la situación. Buscamos ver cómo responde ante la presencia de lenguaje clínico en un entorno fuera de consulta. Para ello, capturamos el audio de un vídeo donde se habla sobre la neumonía y lo pasamos por el sistema a ver cómo reacciona:

```
La neumonía, una vez curada, se resuelve de forma definitiva.
Aunque los antibióticos son fundamentales en su tratamiento, no son
lo único importante. Es crucial que el paciente se mantenga
bien hidratado, pueda expectorar adecuadamente, realice ejercicios respiratorios y,
todo, que se eviten las complicaciones derivadas de la neumonía
o de las enfermedades que la causan o predisponen.

Entidades NER extraídas (JSON parseado):
   Categoría: SINTOMAS
   Categoría: ENFERMEDAD
     - neumonía
   Categoría: TRATAMIENTO
     - antibióticos
------------------------------------------------
```

Figura 48. Resultados de inferencia del Ejemplo 4. Menciones clínicas sin contexto.

Observamos cómo al eliminar la conversación médico-paciente el sistema presenta tendencia a extraer cualquier término médico relevante que pudiese incluir dentro de las categorías definidas. En este caso, podemos ver cómo relaciona la neumonía como enfermedad o los antibióticos como tratamiento, aunque solo han sido nombrados de forma genérica.

Aunque la identificación de *tokens* es precisa, la atribución contextual aún requiere refinamiento en el ámbito clínico.

5 CONCLUSIONES Y LÍNEAS FUTURAS

Tras haber detallado las metodologías empleadas y tras analizar los resultados obtenidos en las fases de *fine-tuning* e inferencia, esta sección final está destinada a recopilar y consolidar los aprendizajes clave del proyecto. Presentaremos las **conclusiones** que hemos alcanzado, destacando los logros y la capacidad que los modelos han mostrado, así como también reconoceremos las limitaciones que presenta el enfoque actual. Finalmente, plantearemos las **líneas de investigación y desarrollo futuro** que podrían expandir y perfeccionar el sistema de extracción de entidades médicas.

5.1 Conclusiones

El proyecto consistió en desarrollar e implementar un sistema completo para la extracción de entidades médicas a partir de grabaciones de audio dentro de consulta clínica. Este sistema opera en dos fases: primero, hace una transcripción de voz a texto utilizando herramientas STT sobre el audio grabado, y, posteriormente, procesa el texto noestructurado utilizando técnicas avanzadas de procesamiento del lenguaje natural (PLN) y *deep-learning*. Mediante el *fine-tuning* del modelo de lenguaje Llama-3.1-8B-Instrut, logramos construir una herramienta capaz de identificar y estructurar información clave en formato JSON.

Los principales **logros y conclusiones** obtenidos al realizar el trabajo son los siguientes:

- **Rendimiento y elección de la herramienta de transcripción (STT):** Se ha validado la eficacia de **Faster Whisper** como herramienta de transcripción para el proyecto. Aunque presenta cierto porcentaje de fallo al transcribir, su alto rendimiento y precisión en la transcripción de conversaciones clínicas a texto ha sido fundamental para la calidad del *input* del modelo PLN, garantizando que el sistema de extracción trabaje sobre una base de datos textual fiable y minimizando la propagación de errores desde la fase inicial.

- **Viabilidad de la extracción de entidades médicas con LLMs:** Demostramos la eficacia de los LLM, especialmente de Llama 3.1, como base para tareas de reconocimiento de entidades nombradas (NER) en el ámbito clínico. El *fine-tuning* ha permitido adaptar un modelo de propósito general a las complejidades y particularidades de la terminología clínica.
- **Rendimiento del *fine-tuning*:** Los resultados tras el entrenamiento han sido altamente satisfactorios. Con una pérdida final de tan solo 0.63 y una precisión de token promedio de casi el 90%, el modelo nos muestra una capacidad de aprendizaje excepcional, asimilando con éxito los patrones y el formato de salida deseados con un dataset tan limitado como el que hemos empleado.
- **Capacidad de Generalización:** Las pruebas de inferencia en textos nuevos, que nunca habían sido previamente introducidos al modelo, han validado la habilidad del modelo para generalizar el conocimiento adquirido. Fue capaz de extraer correctamente la información clínica proporcionada con distintos textos e idiomas.
- **Robustez del Formato:** No solo demostró su consistencia con el manejo de terminología médica, sino que también supo actuar frente a la ausencia de ésta. Cuando no tuvo información, fue capaz de evitar la "alucinación" de datos y dejó la categoría vacía cuando fue necesario. Además, demostró su adherencia al formato JSON especificado, produciendo salidas bien estructuradas.
- **Optimización de Recursos:** Supuso un reto crear un sistema funcional utilizando LLMs de última generación con recursos computacionales tan limitados. La implementación de técnicas como QLoRA fueron claves para permitir el *fine*-tuning de un modelo de 8 mil millones de parámetros en un entorno como Google Colab, superando el desafío inherente a los altos requisitos computacionales que presentan cargar y entrenar los LLM.
- **Naturaleza genérica de las entidades extraídas y la necesidad de precisión:** Si bien las categorías de extracción empleadas (síntoma, enfermedad y tratamiento) se tratan de información genérica y con poca aplicación real, constituyen un buen punto de partida para aplicaciones futuras. La identificación precisa de terminología médica más compleja,

como las abreviaturas o siglas propias de ciertas ramas clínicas (como cardiología o ginecología), requeriría de un *dataset* de entrenamiento mucho más amplio y específico que el nuestro. Sin embargo, el acceso a la información de consultas médicas es muy limitado, complicando la obtención de datos para el entrenamiento.

- **Limitaciones con el Contexto:** A pesar de los logros, identificamos una limitación crítica en el enfoque actual. El modelo tiende a extraer y clasificar todas las menciones a términos médicos, incluso si no están explícitamente contextualizados como una condición que el paciente "sufre". Esto resalta la complejidad de la comprensión contextual dentro del dominio médico.

En definitiva, el proyecto ha concluido con el desarrollo de un sistema funcional y prometedor para la extracción automatizada de información médica, sentando las bases para futuras aplicaciones que puedan mejorar la gestión y el análisis de datos clínicos.

5.2 Líneas de Futura Mejora

El trabajo desarrollado en este TFG abre varias vías de investigación y mejora que podrían explorarse en futuros proyectos. Estas líneas futuras buscan potenciar aún más la precisión, robustez y aplicabilidad del sistema, abordando directamente algunas de las limitaciones identificadas en el punto anterior.

La principal mejora del sistema y la que más impacto sin duda presentaría es la **expansión y refinamiento del *Dataset* de Entrenamiento.** Nuestro tamaño de dataset (60 ejemplos) fue un punto de partida sólido, pero aumentar el volumen de datos de entrenamiento (incluyendo además una mayor diversidad de casos clínicos y estilos de redacción) podría mejorar la capacidad de generalización del modelo y su rendimiento en casos complejos o ambiguos, como los problemas de manejo del contexto que comentábamos en el punto anterior.

Además, sería interesante estudiar la posibilidad de realizar una **diarización de los hablantes**, separando de forma automática las voces de médico y paciente.

Esto transformaría el sistema, facilitando la asignación de entidades extraídas al hablante correcto, lo que tendría un impacto significativo en la eficiencia.

Otra posible mejora del sistema sería la transcripción de audio en tiempo real. Actualmente el sistema necesita el audio completo para procesarlo, segmentarlo y transcribirlo (y en nuestro contexto no tendría demasiado sentido porque no podríamos mandar el texto al LLM hasta que la transcripción estuviese completa). Pero con mayores recursos de GPU, sería posible capturar la transcripción a tiempo real e ir pasando segmentos al modelo (como hicimos de forma local con Ollama en el punto 4.4.6)

El proyecto sienta una base prometedora para la aplicación de LLMs en la extracción de información médica, y las líneas futuras brevemente delineadas abren el camino a un desarrollo continuo y a la creación de herramientas más potentes, fiables y sofisticadas dentro del campo de la salud digital.

6 BIBLIOGRAFÍA

[1] D. Khurana, A. Koli, K. Khatter, and S. Singh, “Natural language processing: state of the art, current trends and challenges,” *Multimed Tools Appl*, vol. 82, no. 3, 2023, doi: 10.1007/s11042-022-13428-4.

[2] L. Siddharth, L. Blessing, and J. Luo, “Natural language processing in-and-for design research,” *Design Science*, vol. 8, 2022, doi: 10.1017/dsj.2022.16.

[3] B. Joshi, “Levels in Natural Language Processing (NLP),” *Medium*, 2022, [Online]. Available: https://bhargavjoshi55.medium.com/levels-in-natural-language-processing-nlp-8e1ddc1cfd1a

[4] S. Singh and A. Mahmood, “The NLP Cookbook: Modern Recipes for Transformer Based Deep Learning Architectures,” *IEEE Access*, vol. 9, 2021, doi: 10.1109/ACCESS.2021.3077350.

[5] K. Kreimeyer *et al.*, “Natural language processing systems for capturing and standardizing unstructured clinical information: A systematic review,” 2017. doi: 10.1016/j.jbi.2017.07.012.

[6] R. Leaman, R. Khare, and Z. Lu, “Challenges in clinical natural language processing for automated disorder normalization,” *J Biomed Inform*, vol. 57, 2015, doi: 10.1016/j.jbi.2015.07.010.

[7] A. Galassi, M. Lippi, and P. Torroni, “Attention in Natural Language Processing,” *IEEE Trans Neural Netw Learn Syst*, vol. 32, no. 10, 2021, doi: 10.1109/TNNLS.2020.3019893.

[8] J. Jordan, “Understanding the Transformer architecture for neural networks.” [Online]. Available: https://www.jeremyjordan.me/transformer-architecture/

[9] Amgadoz, “I compared the different open source whisper packages for long-form transcription,” Reddit. [Online]. Available: https://www.reddit.com/r/LocalLLaMA/comments/1brqwun/i_compared_the_different_open_source_whisper/?tl=es-es

[10] R. A. Khan, “Open Source vs. Closed Source LLMs: Which is Better for Enterprises?,” *Astera*, 2025, [Online]. Available: https://www.astera.com/es/type/blog/open-source-vs-closed-source-llms/

[11] M. J. Boonstra, D. Weissenbacher, J. H. Moore, G. Gonzalez-Hernandez, and F. W. Asselbergs, “Artificial intelligence: Revolutionizing cardiology with large language models,” 2024. doi: 10.1093/eurheartj/ehad838.

[12] B. Meskó and E. J. Topol, “The imperative for regulatory oversight of large language models (or generative AI) in healthcare,” *NPJ Digit Med*, vol. 6, no. 1, p. 120, Jul. 2023, doi: 10.1038/s41746-023-00873-0.

[13] K. Singhal *et al.*, “Large language models encode clinical knowledge,” *Nature*, vol. 620, no. 7972, 2023, doi: 10.1038/s41586-023-06291-2.

[14] A. Vinnarasu and D. V. Jose, “Speech to text conversion and summarization for effective understanding and documentation,” *International Journal of Electrical and Computer Engineering*, vol. 9, no. 5, 2019, doi: 10.11591/ijece.v9i5.pp3642-3648.

[15] M. Kambouri, H. Simon, and G. Brooks, “Using speech-to-text technology to empower young writers with special educational needs,” *Res Dev Disabil*, vol. 135, 2023, doi: 10.1016/j.ridd.2023.104466.

[16] A. Pande and D. Mishra, “The Synergy between a Humanoid Robot and Whisper: Bridging a Gap in Education,” *Electronics (Switzerland)*, vol. 12, no. 19, 2023, doi: 10.3390/electronics12193995.

[17] G. Na, “10 Steps to Setup a Comprehensive Data Science Workspace with VSCode on Windows,” 2020, [Online]. Available: https://medium.com/data-science/10-steps-to-setup-a-comprehensive-data-science-workspace-with-vscode-on-windows-32fe190a8f3

[18] B. P. C, “Ollama Tutorial: Running LLMs Locally Made Super Simple.” [Online]. Available: https://www.kdnuggets.com/ollama-tutorial-running-llms-locally-made-super-simple

[19] Y. Shizuya, “Automatic Speech Recognition -Whisper Variants Comparison: What Are Their Features And How To Implement Them?” [Online]. Available: https://pub.towardsai.net/whisper-variants-comparison-what-are-their-features-and-how-to-implement-them-c3eb07b6eb95

[20] K. Singh, “[Machine Learning] Fine Tuning Open Source Large Language Models (PEFT QLoRA) on Azure Machine Learning,” *Medium*, 2023, [Online]. Available: https://blog.devgenius.io/fine-tuning-large-language-models-on-azure-machine-learning-358338f4e66a

[21] Z. Hu *et al.*, “LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models,” in *EMNLP 2023 - 2023 Conference on Empirical Methods in Natural Language Processing, Proceedings*, 2023. doi: 10.18653/v1/2023.emnlp-main.319.

[22] V. Yaadav, “Exploring and building the LLaMA 3 Architecture : A Deep Dive into Components, Coding, and Inference Techniques,” *Medium*, [Online]. Available: https://medium.com/@vi.ai_/exploring-and-building-the-llama-3-architecture-a-deep-dive-into-components-coding-and-43d4097cfbbb

[23] S. Raschka, “No Understanding and Coding the Self-Attention Mechanism of Large Language Models From ScratchTitle.” [Online]. Available: https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html

[24] SYSTRAN, “Faster Whisper transcription with CTranslate2,” SYSTRAN/faster-whisper. Accessed: Feb. 16, 2025. [Online]. Available: https://github.com/SYSTRAN/faster-whisper?tab=readme-ov-file

[25] HuggingFace, “Big data? Datasets to the rescue!,” LLM COURSE. [Online]. Available: http://huggingface.co/learn/llm-course/chapter5/4?fw=pt

[26] HuggingFace, “Bitsandbytes,” Transformers. [Online]. Available: https://huggingface.co/docs/transformers/main/quantization/bitsandbytes

[27] HuggingFace, “LoRA (Low-Rank Adaptation),” LLM COURSE. [Online]. Available: https://huggingface.co/learn/llm-course/chapter11/4

7 ANEXOS

7.1 ANEXO I: Medición de Tiempos con Timeit

Para evaluar la eficiencia y el rendimiento de los distintos modelos que hemos usado a lo largo del proyecto, sobretodo a la hora de realizar la transcripción, utilizamos la librería estándar de Python 'timeit'. Esta librería nos proporciona una forma precisa de medir el tiempo de ejecución de pequeños fragmentos de código, esto nos permite aislar el entorno de ejecución y minimizar la influencia de factores externos.

Nuestro objetivo fue medir el tiempo antes y después de la ejecución de las funciones o bloques de código a evaluar utilizando '*timeit.default_timer()*, que ofrece la mayor precisión disponible.

A continuación, representamos en la Figura 49 un breve código ejemplo que representa cómo se aplicó la función de *timeit* en el proyecto.

```
import timeit

#Marcamos el tiempo de inicio
tiempo_inicio=timeit.default_timer()

#..Procesos y funciones que queramos medir en segundos..#

#Marcamoso el tiempo final
tiempo_fin=timeit.default_timer()

#Calculamos y mosotramos la diferencia:
tiempo=tiempo_fin-tiempo_inicio
print(f"El proceso tardó: {tiempo:2f} segundos")
```

Figura 49. Código ejemplo de la función timeit

7.2 ANEXO II: Conversión a .wav

El script Python que podemos ver en la Figura 50 implementa la conversión de archivos de audio a formato .wav utilizando Pydub. Es fundamental para asegurar la compatibilidad con el modelo de transcripción STT.

```python
def convertir_transcripcion_a_wav_si_necesario(audio):

# El nombre de la transcripción que acabamos de capturar:
    nombre_archivo_original = audio
    # Intentamos 'adivinar' la extensión del archivo
    ruta_completa_original = None
    extension_original = None
    #lista de las posibles extensiones
    posibles_extensiones = ['.wav', '.mp3', '.ogg', '.flac',
                            '.m4a', '.aac', '.mp4']

#bucle que prueba las diferentes extesiones en el nombre original:
    for ext in posibles_extensiones:
        if os.path.exists(nombre_archivo_original + ext):
            ruta_completa_original = nombre_archivo_original + ext
            extension_original = ext.lower()
            break
    if ruta_completa_original is None: #si no encuentra ninguna, sale con error
        print(f"""Error: No se encontró el archivo '{nombre_archivo_original}'
                  con ninguna de las extensiones comunes.""")
        return None
 #si encuentra el archivo, lo anuncia y...
    print(f"Archivo encontrado y procesando: {ruta_completa_original}")
 #en caso de ser ya un archivo .wav lo devuelve tal cual:
    if extension_original == '.wav':
        print(f"""El archivo '{ruta_completa_original}' ya está en formato WAV.
              No se requiere conversión.""")
        return ruta_completa_original
    else: #si es otro formato, lo convierte
        print(f"""El archivo '{ruta_completa_original}' no está en formato WAV.
                Intentando convertir a WAV...""")
        try:
            # Cargar el archivo de audio en la ruta originanl usando pydub
            audio = AudioSegment.from_file(ruta_completa_original,
                                           format=extension_original.lstrip('.'))

            # Definir el nombre del archivo de salida .wav
            nombre_archivo_wav = f"transcripcion.wav"

            # Exportar a WAV
            audio.export(nombre_archivo_wav, format="wav")
            print(f"""¡Conversión exitosa! El archivo convertido se ha guardado
                  como '{nombre_archivo_wav}'.""")
            return nombre_archivo_wav
        except Exception as e:
            print(f"Error al intentar convertir el archivo: {e}")
            return None
```

Figura 50. Código con la conversión a formato .wav

7.3 ANEXO III: Código para segmentar y transcribir el audio en Faster Whisper

En la Figura 51 y Figura 52 veremos con mayor profundidad el funcionamiento de la función de segmentar el audio (4.3.2), además de cómo transcribe cada segmento llamando a la función *transcribir_array_audio*.

```
def transcribir_audio_segmentado( #valores predefinidos
    archivo_audio: str,
    modelo_whisper: WhisperModel,
    tamano_segmento: int = 30,
    sr_whisper: int = 16000, #convertimos el .wav a 16kHz
    aplicar_filtro: bool = False,
  #valores de corte superior e inferior del filtro:
    lowcut_freq: float = 200,
    highcut_freq: float = 3000
) -> str | None:
    if modelo_whisper is None:
        print("Error: El modelo Whisper no ha cargado correctamente.")
        return None

#definimos la variable donde almacenaremos la transcripción:
    transcripciones_completas = []
    try: #cargamos el audio y calculamos los segmentos totales
        print(f"""Cargando audio '{archivo_audio}'
                y remuestreando a {sr_whisper} Hz...""")
         #archivo de audio:
        y, sr_original = librosa.load(archivo_audio, sr=sr_whisper)
         #duración total en segundos:
        duracion_total = librosa.get_duration(y=y, sr=sr_whisper)
        print(f"Duración total del audio: {duracion_total:.2f} segundos.")
        #Calculamos el total de segmentos necesarios:
        num_segmentos = int(np.ceil(duracion_total / tamano_segmento))

      #calculamos inicio y fin de cada segmento dentro del array de segmentos
        for i in range(num_segmentos):
            inicio_seg = i * tamano_segmento
            fin_seg = min((i + 1) * tamano_segmento, duracion_total)

            inicio_sample = int(inicio_seg * sr_whisper)
            fin_sample = int(fin_seg * sr_whisper)
            segmento_audio_array = y[inicio_sample:fin_sample]
```

Figura 51. Función para segmentar y transcribir el audio con Faster Whisper (I)

```
        #le aplicamos el filtro paso-banda antes de transcribirlo
            if aplicar_filtro:
                print(f"Aplicando filtro a segmento {i+1}/{num_segmentos}...")
                segmento_audio_array = butter_bandpass_filter(
                    segmento_audio_array, lowcut_freq, highcut_freq, sr_whisper
                )
      #y transcribimos:
            print(f"""Transcribiendo segmento
                    {i+1}/{num_segmentos} ({inicio_seg:.1f}-{fin_seg:.1f}s)
                    ...""")
      #Llamamos a la función que hemos definido anteriormente para transcribir:
            transcripcion_segmento_actual = transcribir_array_audio(
                segmento_audio_array, sr_whisper, modelo_whisper
            )
      #y añadimos el segmento a la transcripccion completa:
            transcripciones_completas.append(transcripcion_segmento_actual)
# Al terminar el bucle devolvemos la transcripcion completa:
        return " ".join(transcripciones_completas)

    except FileNotFoundError:
        print(f"""Error: Archivo de audio no encontrado
                  en la ruta: {archivo_audio}""")
        return None
    except Exception as e:
        print(f"Error durante la transcripción: {e}")
        return None
```

Figura 52. Función para segmentar y transcribir el audio con Faster Whisper (II)

Vemos que hay, además, una función 'archivo_audio_a_transcribir', la cual no es más que la llamada al archivo de audio. En estas pruebas, al manejar gran cantidad de audios, cargamos todos en el directorio y a través de una sencilla función fuimos seleccionándolos. En el sistema final, esta variable podría ser una función de petición de archivo (como actualmente) o simplemente el nombre del archivo con su extensión si ejecutamos el código en el mismo directorio que lo contiene.

7.4 ANEXO IV: Función para la limpieza del resultado (parsing)

La función *parse_and_print_json* que adjuntamos en la Figura 53 trata de extraer el bloque JSON de la respuesta generada por el modelo buscando delimitadores comunes como "json...".

```
def parse_and_print_json(response_text):
    try:
        match = re.search(r"```json\s*(.*?)\s*```", response_text, re.DOTALL)
        if match:
            json_str = match.group(1)
        else:
            json_str = response_text.strip()
            json_start = json_str.find('{')
            if json_start != -1:
                json_str = json_str[json_start:]
            if not json_str.endswith('}'):
                last_brace = json_str.rfind('}')
                if last_brace != -1:
                    json_str = json_str[:last_brace + 1]

        parsed_entities = json.loads(json_str)

        print("\nEntidades NER extraídas (JSON parseado):")
        if isinstance(parsed_entities, dict):
            for category, entities in parsed_entities.items():
                if isinstance(entities, list):
                    print(f"  Categoría: {category.upper()}")
                    for entity in entities:
                        print(f"    - {entity}")
                else:
                    print(f"""  Advertencia: {category}
                          no es una lista: {entities}""")
        else:
            print(f"""ERROR: El JSON parseado no es el diccionario esperado.
                  Tipo: {type(parsed_entities)}""")
            print(f"Salida parseada: {parsed_entities}")

    except json.JSONDecodeError as e:
        print(f"""\nERROR: No se pudo parsear la salida
                          como JSON válido. Error: {e}""")
        print("La salida generada por el modelo fue:")
        print(json_str)
    print("-" * 50)
```

Figura 53. Función de parseo del JSON obtenido

Si no los encuentra, la función trata de localizar llaves o corchetes de apertura y cierre, intentando aislar la estructura JSON principal. Este paso de limpieza es crucial para garantizar que solo la estructura JSON es procesada.

7.5 ANEXO V: Librerías y versiones empleadas

En la siguiente tabla presentamos la versión correspondiente a las librerías que hemos empleado en este proyecto.

Librería	Versión	Librería	Versión
accelerate	1.7.2000	peft	0.15.2
bitsandbytes	0.46.0	pydub	0.25.1
datasets	3.6.2000	scipy	1.15.3
faster-whisper	1.1.2001	torch	2.6.0+cu124
librosa	0.11.0	transformers	4.52.4
numpy	2.0.2	trl	0.18.1

Tabla 7. Librerías del proyecto