

**Escuela Técnica Superior de Ingeniería
Universidad de Huelva**

Grado en Ingeniería Informática

Trabajo Fin de Grado

**Desarrollo de un Sistema Asistente de Música
Basado en Aprendizaje Automático y Procesamiento
del Lenguaje Natural con Modelos de Lenguaje
Generativos**

Autor: Antón Maestre Gómez
Tutor: Jacinto Mata Vázquez

junio, 2025

A mi abuelo

Por ayudarme a ser la persona que soy. Siempre te recordaré.

Resumen

Este Trabajo de Fin de Grado presenta el desarrollo de un asistente de música, basado en aprendizaje automático y procesamiento de lenguaje natural, orientado a mejorar la interacción entre el usuario y su biblioteca musical. El objetivo principal es construir un asistente conversacional que sea capaz de interpretar peticiones escritas en lenguaje natural y clasificarlas según la intención en tareas como añadir canciones, eliminarlas, visualizar la lista de reproducción o vaciarla.

Para ello, se ha investigado en profundidad sobre las metodologías existentes en el campo del aprendizaje profundo y los modelos de lenguaje generativos, especialmente aquellos basados en arquitecturas LLM. A partir de este análisis, se entrenaron y ajustaron modelos generativos como LLaMA y Falcon, aplicando técnicas de fine-tuning y optimización de hiperparámetros.

El sistema se implementó como un chatbot conversacional conectado a una base de datos musical. Esto permite al usuario comunicarse de manera directa, mejorando la experiencia general. Los resultados del entrenamiento mostraron una mejora continua, con el modelo alcanzando una precisión notable y capacidad de generalización en la clasificación de intenciones.

En conclusión, el trabajo demuestra que los modelos generativos de lenguaje pueden integrarse con éxito en sistemas conversacionales. Esto abre la posibilidad de desarrollar asistentes musicales más avanzados, que utilicen el contexto conversacional para ofrecer una experiencia adaptativa y personalizada al usuario.

Palabras clave: Asistente de música, aprendizaje automático, procesamiento de lenguaje natural, modelos generativos, LLaMA, Falcon, chatbot conversacional, fine-tuning, optimización de hiperparámetros, clasificación de intenciones, base de datos, personalización.

Abstract

This Bachelor's Thesis presents the development of a music assistant based on machine learning and natural language processing, aimed at enhancing the interaction between users and their music library. The primary objective is to build a conversational assistant capable of interpreting user requests written in natural language and classifying them according to intent, such as adding songs, removing them, viewing the playlist, or clearing it.

To achieve this, an in-depth investigation was conducted into existing methodologies in the field of deep learning and generative language models, particularly those based on large language model (LLM) architectures. Based on this analysis, generative models such as LLaMA and Falcon were trained and fine-tuned, employing hyperparameter optimization techniques.

The system was implemented as a conversational chatbot connected to a music database, enabling users to communicate directly, thereby enhancing the overall experience. The training results demonstrated continuous improvement, with the model achieving remarkable accuracy and generalization ability in intent classification.

In conclusion, this work demonstrates that generative language models can be successfully integrated into conversational systems. This paves the way for the development of more advanced music assistants that use conversational context to provide an adaptive and personalized user experience.

Keywords: Music assistant, machine learning, natural language processing, generative models, LLaMA, Falcon, conversational chatbot, fine-tuning, hyperparameter optimization, intent classification, database, personalization.

Agradecimientos

Quiero expresar mi agradecimiento a todas las personas que han hecho posible la realización de este trabajo.

En primer lugar, agradezco a mi tutor Jacinto Mata por su orientación y ayuda brindadas durante el desarrollo del proyecto. También agradezco a mi profesor Krisztian Balog por compartir sus conocimientos sobre la creación de servidores web.

Mi agradecimiento también a la Universidad de Stavanger por facilitarme unas instalaciones y herramientas propicias para llevar a cabo este trabajo.

Finalmente, me gustaría dedicar un agradecimiento especial a mi familia por su paciencia y comprensión a lo largo de todo el proceso. Su apoyo incondicional ha sido la base de mi motivación para concluir este proyecto.

Antón Maestre.

Huelva, 2025

Índice general

| | |
|--|-------------|
| Resumen | II |
| Abstract | III |
| Agradecimientos | IV |
| Índice de Figuras | VII |
| Índice de Tablas | VIII |
| 1. Introducción | 1 |
| 1.1. Motivación | 1 |
| 1.2. Objetivos | 1 |
| 1.3. Competencias | 2 |
| 1.4. Estructura de la Memoria | 3 |
| 2. Marco Teórico | 5 |
| 2.1. Aprendizaje Automático | 5 |
| 2.2. Aprendizaje Profundo | 6 |
| 2.2.1. Epochs y tamaño de lote | 7 |
| 2.2.2. Tasa de aprendizaje, <i>weight decay</i> y <i>warm up ratio</i> | 8 |
| 2.2.3. Función de pérdida | 9 |
| 2.2.4. Overfitting | 9 |
| 2.2.5. Early stopping y validación | 10 |
| 2.2.6. Gradientes | 10 |
| 2.3. Aprendizaje por Transferencia | 11 |
| 2.4. Procesamiento del Lenguaje Natural | 12 |
| 2.4.1. Preprocesamiento del texto | 12 |
| 2.4.2. Representación del texto | 13 |
| 2.4.3. Clasificación de texto en PLN | 14 |
| 2.5. Transformers | 14 |
| 2.5.1. Redes Neuronales Recurrentes (RNN) | 14 |
| 2.5.2. Fundamentos de los Transformer | 15 |
| 2.5.3. Arquitectura de los Transformers | 16 |
| 2.6. Modelos Generativos de Lenguaje | 18 |
| 2.6.1. Zero-shot, Few-shot y Fine-tuning | 18 |
| 2.6.2. Ventajas y limitaciones | 19 |
| 2.7. Fine-tuning y optimización de hiperparámetros | 19 |
| 2.8. Modelos de Aprendizaje Empleados | 20 |
| 2.9. Medidas de Evaluación | 21 |
| 2.9.1. Accuracy | 21 |
| 2.9.2. Precisión | 21 |
| 2.9.3. Exhaustividad (recall) | 22 |
| 2.9.4. F1-Score | 22 |
| 2.9.5. Macro-promedio | 22 |
| 2.9.6. Matriz de confusión | 23 |

| | |
|---|-----------|
| 2.10. Trabajos relacionados | 24 |
| 2.11. Tecnologías y Recursos Utilizados | 24 |
| 2.11.1. Python | 25 |
| 2.11.2. Google Colab | 25 |
| 2.11.3. Hugging Face | 26 |
| 2.11.4. GitHub | 26 |
| 2.11.5. LoRA | 26 |
| 2.11.6. Optuna | 26 |
| 2.11.7. SQL | 27 |
| 2.11.8. Flask / Dialoguekit | 27 |
| 3. Metodología, Experimentación y Resultados | 28 |
| 3.1. Descripción General de la Metodología | 28 |
| 3.2. Preparación de los Datos | 29 |
| 3.2.1. Descripción de los Datos | 30 |
| 3.2.2. División de Datos | 30 |
| 3.2.3. Preparación para su uso | 30 |
| 3.2.4. Tokenización y Codificación | 31 |
| 3.3. Selección de los Modelos | 32 |
| 3.4. Ajuste de Hiperparámetros | 33 |
| 3.4.1. Métodos de Búsqueda | 33 |
| 3.4.2. Hiperparámetros Evaluados | 34 |
| 3.4.3. Tamaño del Conjunto de Datos Utilizado | 34 |
| 3.4.4. Criterios de Selección | 35 |
| 3.5. Entrenamiento de los Modelos | 35 |
| 3.5.1. Baselines | 35 |
| 3.5.2. Configuración de Entrenamiento | 36 |
| 3.5.3. Monitorización del Entrenamiento | 37 |
| 3.6. Evaluación y Análisis de Resultados | 38 |
| 3.7. Análisis de Errores | 41 |
| 3.8. Implementación del ChatBot | 42 |
| 3.8.1. Base de Datos | 42 |
| 3.8.2. Creación e interacción en el Servidor Web | 43 |
| 3.8.3. Conexión Remota entre el Chatbot y los Modelos | 45 |
| 3.8.4. Arquitectura del Sistema | 45 |
| 3.8.5. Resultado final | 46 |
| 3.8.6. Posible mejora | 46 |
| 4. Conclusiones y Trabajo Futuro | 48 |
| 4.1. Conclusiones | 48 |
| 4.2. Trabajo Futuro | 50 |
| 4.3. Planificación Temporal del Trabajo Realizado | 51 |
| Referencias | 53 |
| A. Anexos | 55 |
| A.1. Código y programa del ChatBot en Google Drive | 55 |
| A.1.1. Tratamiento de los modelos generativos | 55 |
| A.1.2. Implementación del ChatBot | 56 |
| A.1.3. Ejecución del sistema | 56 |

Índice de Figuras

| | |
|--|----|
| 2.1. Diagrama de una neurona en la red neuronal | 7 |
| 2.2. Diagrama de comparación entre Machine Learning y Deep Learning | 7 |
| 2.3. Aprendizaje del modelo en base al valor de weight decay | 8 |
| 2.4. Gráfica del método Early stopping | 10 |
| 2.5. Diagrama de funcionamiento del Aprendizaje por Transferencia | 11 |
| 2.6. Representación espacial one-hot encoding para un vocabulario con tres tokens. | 13 |
| 2.7. Diagrama de comparación entre arquitecturas CBOW y Skip Gram | 14 |
| 2.8. Arquitectura de una RNN | 15 |
| 2.9. Procesamiento del vector atención del token murciélagos | 16 |
| 2.10. Arquitectura de un Transformer | 17 |
| 2.11. Diagrama de fine-tuning a un modelo | 19 |
| 2.12. Ejemplo de una matriz de confusión | 23 |
| 2.13. Logos de las herramientas utilizadas. | 25 |
| 3.1. Pipeline general del marco de experimentación | 29 |
| 3.2. División del conjunto original de datos | 31 |
| 3.3. Preparación de datos según su uso | 31 |
| 3.4. Diagrama de funcionamiento de Optuna | 33 |
| 3.5. Resultados de Optuna durante la búsqueda de hiperparámetros | 36 |
| 3.6. Evolución del entrenamiento de Llama monitorizada con TensorBoard | 38 |
| 3.7. Evolución del entrenamiento de Falcon monitorizada con TensorBoard | 39 |
| 3.8. Matriz de confusión para el modelo LLaMA. | 40 |
| 3.9. Matriz de confusión para el modelo Falcon. | 41 |
| 3.10. Estructura y comunicación con la base de datos. | 43 |
| 3.11. Interfaz del ChatBot | 44 |
| 3.12. Comunicación remota entre el entorno local y los modelos. | 45 |
| 3.13. Esquema de funcionamiento del ChatBot | 46 |
| 3.14. Ejemplos para añadir canciones en el ChatBot | 47 |
| 3.15. Ejemplo para eliminar una canción en el ChatBot | 47 |
| 3.16. Ejemplo para ver/limpiar la lista de canciones en el ChatBot | 47 |

Índice de Tablas

| | |
|--|----|
| 3.1. Espacio de búsqueda de Optuna para los hiperparámetros | 34 |
| 3.2. Mejores combinaciones de hiperparámetros obtenidas para cada modelo | 36 |
| 3.3. Resultados de evaluación sobre el conjunto de test. | 39 |
| 3.4. Resultados por clase para los modelos evaluados (conjunto de test). | 40 |
| 4.1. Planificación temporal del trabajo realizado. | 51 |

CAPÍTULO 1

Introducción

Durante este primer capítulo, daremos contexto al proyecto aportando una visión general del mismo. Partiremos exponiendo los motivos que han impulsado su realización, para pasar después a los objetivos que se persiguen. Luego, las competencias que se esperan adquirir y, finalmente, se detallará una descripción sobre la estructura del documento.

1.1. Motivación

Estos últimos años han sido testigos del surgimiento de numerosos asistentes virtuales que tenían como objetivo mejorar la vida cotidiana de los usuarios. Tanto en tareas del hogar, del trabajo o aquellas relacionadas con el entretenimiento, diversos modelos han tratado de facilitar la experiencia a sus clientes. Particularmente en el campo de la música, estos sistemas buscan mejorar la interacción de sus oyentes con sus listas de reproducción. No obstante, muchos pecan de ser poco flexibles al estar regulados por comandos estrictos, creando así ciertas limitaciones en la comunicación.

Con la llegada de los modelos de lenguaje generativos, surge una nueva posibilidad. Ahora los asistentes pueden desempeñar papeles más complejos, como la comprensión de las intenciones en las peticiones del usuario escritas de forma natural. La motivación de este proyecto surge precisamente de esto. Mediante el desarrollo de un asistente conversacional basado en un modelo de lenguaje generativo, se logrará que los oyentes disfruten de una experiencia más fluida y flexible al interactuar sobre sus listas de reproducción.

Así pues, dicho sistema estará basado en los fundamentos del aprendizaje automático y el procesamiento del lenguaje natural. Su trabajo se basará en la comprensión de las solicitudes del usuario y su gestión de las listas de reproducción en base a ello.

1.2. Objetivos

El propósito principal de este trabajo es desarrollar un asistente musical conversacional que utilice modelos de lenguaje generativos para interpretar y clasificar las intenciones del usuario en una interacción natural.

Para alcanzar este propósito, se han planteado los siguientes objetivos específicos:

- Explorar las metodologías y enfoques existentes en el ámbito del aprendizaje automático y PLN aplicados a la comprensión de textos.
- Desarrollar y personalizar modelos generativos basados en arquitecturas Transformer, como LLaMA y Falcon.
- Diseñar e implementar un chatbot conversacional capaz de gestionar tareas musicales mediante lenguaje natural.
- Optimizar los modelos mediante el ajuste de hiperparámetros usando herramientas como Optuna.
- Evaluar la efectividad del sistema con métricas adecuadas que reflejen su capacidad para comprender y ejecutar instrucciones del usuario.
- Aprender a elaborar documentación científica en LaTeX que describa de forma rigurosa el desarrollo del proyecto.
- Explorar el uso de plataformas como Google Colab para la ejecución y experimentación con modelos de manera eficiente.

1.3. Competencias

La principal competencia adquirida con la realización de este trabajo ha sido **CE7-C - Capacidad para conocer y desarrollar técnicas de aprendizaje computacional y diseñar e implementar aplicaciones y sistemas que las utilicen, incluyendo las dedicadas a extracción automática de información y conocimiento a partir de grandes volúmenes de datos.**

Además, se han alcanzado otras competencias del plan de estudios relacionadas con la línea de este TFG:

- **Capacidad para conocer y desarrollar técnicas de aprendizaje computacional.** Mediante el estudio y aplicación del aprendizaje profundo, se ha alcanzado esta capacidad trabajando sobre modelos basados en arquitecturas *Transformer*. Específicamente, se han desarrollado modelos generativos como *LLaMA* o *Falcon*, analizando su funcionamiento y poniéndolos en práctica mediante el entrenamiento con hiperparámetros optimizados.
- **Capacidad para implementar aplicaciones y sistemas que la utilicen.** Esta capacidad se ha obtenido a través de la implementación del sistema conversacional, que integra los modelos generativos ya entrenados en un chat con el usuario. La construcción del chatbot ha requerido la integración de una base de datos musical y la planificación del flujo de la conversación.

- **Capacidad para la extracción automática de información.** El desarrollo de esta capacidad se ha logrado con la detección de las intenciones del usuario en los mensajes escritos en lenguaje natural. Mediante el uso de los modelos generativos entrenados, el sistema es capaz de extraer automáticamente si la petición es para visualizar la lista de reproducción, borrar todas sus canciones, o añadir o quitar una canción.
- **Capacidad para adquirir conocimiento a partir de grandes volúmenes de datos.** Durante el desarrollo del sistema, se han trabajado con datos musicales que fueron procesados y empleados para entrenar los modelos generativos. Además, se ha diseñado una base de datos musical para poder interactuar con las listas de reproducción del usuario.

1.4. Estructura de la Memoria

A partir de aquí, la memoria se organiza de la siguiente forma:

- En el [Capítulo 2](#), Marco Teórico, se presentan y explican los fundamentos y conceptos clave que sustentan el desarrollo de este trabajo. Esta sección aborda en profundidad las arquitecturas de los modelos de lenguaje generativos *LLaMa* y *Falcon*, así como los principios del aprendizaje automático (ML), aprendizaje profundo y por transferencia, y el procesamiento del lenguaje natural (PLN), los cuales son esenciales para comprender el funcionamiento de los modelos utilizados. También se exploran las diversas técnicas y metodologías relacionadas con la optimización de estos modelos y se justifican las decisiones sobre las arquitecturas seleccionadas.
- En el [Capítulo 3](#), Metodología, Experimentación y Resultados, se describe el enfoque general del proyecto, detallando los pasos seguidos en el desarrollo del sistema. Este capítulo cubre desde el preprocesamiento de datos hasta la implementación de los modelos en el agente conversacional. Se especifican los métodos y técnicas empleados en el entrenamiento de los modelos generativos LLaMA y Falcon, se explica el uso de herramientas como Optuna para la optimización de parámetros y la integración del sistema en el chatbot. Además, se ofrece una evaluación y análisis del rendimiento de los modelos, detallando los errores y las posibles causas de los mismos. Finalmente, se expone la estructura y diseño de la base de datos musical que interactúa con el asistente, garantizando una gestión eficiente de la información del usuario y las listas de reproducción.
- En el [Capítulo 4](#), Conclusiones, se sintetizan los hallazgos del proyecto, destacando los logros más relevantes y la efectividad de las soluciones implementadas. Además, se identifican las limitaciones que surgieron durante el desarrollo y se proponen áreas de mejora. Este capítulo también discute el impacto potencial de los modelos y sistemas

desarrollados, así como las posibles aplicaciones futuras en otros dominios, abriendo nuevas líneas de investigación en la interacción entre los usuarios y los asistentes musicales basados en lenguaje natural.

- Finalmente, en el [Apéndice A](#), Anexo, se presenta el código fuente utilizado en el proyecto, proporcionando un panorama completo sobre los aspectos prácticos y metodológicos del trabajo.

CAPÍTULO 2

Marco Teórico

2.1. Aprendizaje Automático

El aprendizaje automático o Machine Learning (ML) se trata de un subcampo de la Inteligencia Artificial centrado en el desarrollo de algoritmos computacionales que mejoran automáticamente a través de la experiencia. Esto permite a la máquina aprender de los datos, identificar patrones y tomar decisiones en base a ello con la mínima intervención humana.[\[1\]](#).

Los algoritmos de aprendizaje automático se dividen principalmente en tres categorías, siendo las dos primeras las más comunes:

- **Aprendizaje Supervisado** [\[2\]](#). En el aprendizaje supervisado, el sistema se entrena con ejemplos que ya tienen la respuesta correcta. Cada ejemplo viene con un conjunto de características, como entradas, y una etiqueta asociada. La tarea del algoritmo es aprender una función que relacione esas entradas con la salida deseada. Una vez entrenado, se espera que pueda generalizar esa relación a nuevos datos que nunca ha visto.

Este tipo de aprendizaje se suele aplicar en dos casos concretos. Cuando lo que se quiere es asignar una categoría o clase predefinida, hablamos de clasificación. Por ejemplo, si se tiene un correo electrónico y se quiere saber si es spam o no. En cambio, si lo que se busca es estimar un valor numérico, como el precio de una vivienda según su tamaño y ubicación, estamos ante un problema de regresión.

- Cuando lo que se quiere es asignar una categoría o clase predefinida, hablamos de clasificación. Por ejemplo, si se tiene un correo electrónico y se quiere saber si es spam o no.
 - En cambio, si lo que se busca es estimar un valor numérico, como podría ser el precio de una vivienda según su tamaño y ubicación, estamos ante un problema de regresión.
- **Aprendizaje no Supervisado** [\[2\]](#). En el aprendizaje no supervisado, las cosas cambian. Aquí el sistema no recibe etiquetas. Solo tiene los datos, y su tarea es buscar patrones por sí mismo. Una aplicación común es el agrupamiento (*clustering*): encontrar conjuntos de instancias que se parecen entre sí. Si muchos datos comparten

certas características, es posible que pertenezcan al mismo grupo.

- **Aprendizaje por Refuerzo [3]**. El aprendizaje por refuerzo funciona de manera diferente. En lugar de ejemplos con respuestas, el algoritmo interactúa con un entorno. Prueba acciones, recibe recompensas o castigos, y con el tiempo aprende qué hacer. No se le dice directamente cuál es la acción correcta. Aprende a través de la prueba y el error, ajustando su política de comportamiento para conseguir más recompensa acumulada. Con suficiente práctica, acaba desarrollando una estrategia que le permite actuar de forma eficaz.

Este proyecto se ha fundamentado en Aprendizaje Supervisado, ya que la tarea a resolver es una clasificación de textos según la intención del mismo y partiendo de un conjunto etiquetado. Más concretamente, es una clasificación con 4 posibles etiquetas:

- **remove**: La finalidad de la solicitud es retirar una canción de la playlist.
- **clear**: Se busca eliminar todas las canciones de la lista de reproducción.
- **add**: El usuario busca añadir una canción a su playlist.
- **view**: El usuario solicita ver su lista de reproducción.

2.2. Aprendizaje Profundo

El aprendizaje profundo o *Deep Learning* [4] forma parte del aprendizaje automático. Utiliza modelos con múltiples capas que procesan los datos paso a paso. Cada capa extrae algo nuevo y se comunica con la siguiente a través de una **red neuronal**. Así, el sistema va formando representaciones cada vez más complejas.

Las redes neuronales [5] se forman por un conjunto de unidades de procesamiento denominadas neuronas. Como se observa en la [Figura 2.1](#), estas reciben una señal de entrada que someten a una función generalmente no lineal, dando lugar a una señal de salida. Dentro de la red, las neuronas se conectan por medio de enlaces, los cuales tienen un valor de peso. Las señales se multiplican por este valor cuando pasan a través de ellos.

La imagen de la [Figura 2.2](#) muestra la principal diferencia entre aprendizaje automático y aprendizaje profundo. Mientras que en Machine Learning, la extracción de características y clasificación se realizan por separado, en Deep Learning una red neuronal profunda las realiza de manera conjunta.

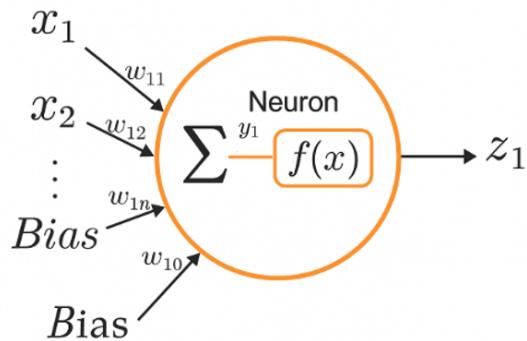


Figura 2.1: Diagrama de una neurona en la red neuronal

Este enfoque ha sido útil en tareas como el procesamiento del lenguaje. En lugar de programar reglas, los modelos aprenden por sí solos cómo representar palabras y frases a partir del uso real que se hace de ellas.

En este trabajo se han empleado modelos basados en arquitecturas profundas para la clasificación de textos (*LLMs*). Por eso, es importante entender cómo se entrena y ajustan estos modelos, además de los factores que afectan a su rendimiento.

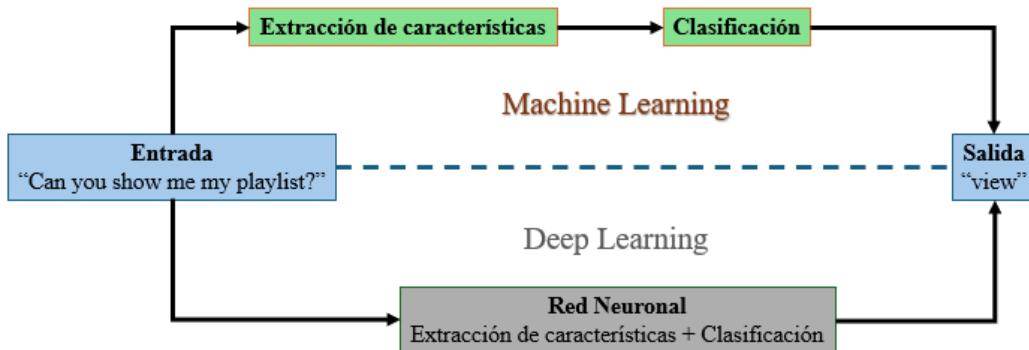


Figura 2.2: Diagrama de comparación entre Machine Learning y Deep Learning

En las siguientes secciones se explican los conceptos fundamentales relacionados con su proceso de entrenamiento.

2.2.1. Epochs y tamaño de lote

Una época o *epoch* en inglés sucede cuando se realiza una pasada completa por todo el conjunto de datos de entrenamiento durante el aprendizaje del modelo. En cada pasada, ajusta sus parámetros según los errores que va encontrando, por lo que el aprendizaje se basa en repasar y corregir, una y otra vez, sobre el mismo conjunto de ejemplos.

Por otra parte, se han introducido los bloques de fine tuning como un conjunto determinado de épocas. Esta estrategia permite analizar el rendimiento del modelo tras cada bloque, facilitando el control y la detección temprana de posibles problemas como el sobreajuste. Además, al encadenar varios bloques, se puede llegar a un número alto de épocas si las métricas lo piden. Todo el proceso se va adaptando según los resultados y el uso de recursos. En este proyecto, se han empleado bloques conformados por 3 épocas cada uno.

2.2.2. Tasa de aprendizaje, *weight decay* y *warm up ratio*

La tasa de aprendizaje o *learning rate* [6] es un hiperparámetro fundamental en el entrenamiento de modelos de aprendizaje automático. Se encarga de marcar el tamaño de los pasos que da el optimizador al ajustar los pesos de la red en cada iteración. Si su valor es alto, el modelo puede aprender más rápido, pero corre el riesgo de saltarse los mínimos de la función de pérdida y no converger. Si es bajo, el aprendizaje puede resultar excesivamente lento, llegando incluso a estancarse en mínimos locales.

Para regular esto, se emplea *weight decay* [7]. Se trata de una técnica de regularización que consiste en añadir un término adicional a la función de pérdida, el cual penaliza los pesos grandes del modelo, incentivando así que estos se mantengan lo más pequeños posible. En definitiva, *weight decay* ayuda a reducir el sobreajuste (**overfitting**) y mejorar la capacidad de generalización del modelo. Sin embargo, un valor alto de *weight decay* puede llegar a limitar la capacidad de aprendizaje, por lo que se debe buscar el nivel equilibrado de ajuste que permita mantener un buen *learning rate*. (ver Figura 2.3).

Por otra parte, también se ha empleado *warm_up_ratio* [8]. Se trata de un hiperparámetro que define una etapa inicial en la que la tasa de aprendizaje aumenta poco a poco, desde cero hasta el valor definido para el entrenamiento. El objetivo es evitar cambios bruscos cuando los pesos aún no están bien ajustados. Esta técnica ayuda a estabilizar la optimización en los primeros pasos.

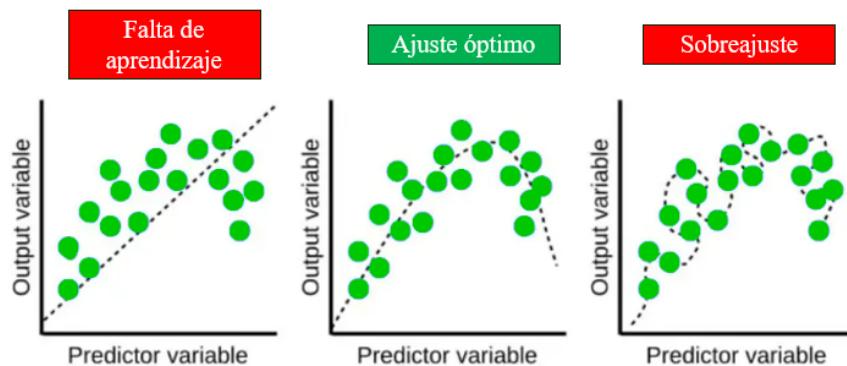


Figura 2.3: Aprendizaje del modelo en base al valor de *weight decay*

2.2.3. Función de pérdida

Para que un modelo sea capaz de comprender y predecir el lenguaje natural, se necesita una medida que muestre qué tan bien se desempeña el mismo. La función de pérdida [9] cuantifica la diferencia entre las predicciones del modelo y los valores esperados. El objetivo final del aprendizaje será reducir esta función de pérdida, de manera que el modelo sea lo más eficiente posible. Sin embargo, dependiendo del tipo de tarea, como regresión, clasificación o segmentación, se debe elegir una función de pérdida.

En este caso, nuestro objetivo de clasificación multiclas nos lleva a escoger la **entropía cruzada** o *cross-entropy loss*, al tratarse de la mejor opción para este tipo de problemas. Permite optimizar el modelo y, como consecuencia, facilitar su aprendizaje y convergencia. Para cada muestra, se mide la diferencia entre el valor real (1 cuando pertenece a esa clase, 0 cuando no) y la probabilidad predicha por el modelo para cada clase (un valor entre 0 y 1). Para obtenerla, se realiza el siguiente cálculo:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C y_{i,c} \cdot \log(\hat{y}_{i,c}) \quad (2.1)$$

Donde:

- N : número total de muestras.
- C : número de clases.
- $y_{i,c}$: valor real (1 si la muestra i pertenece a la clase c , 0 en caso contrario).
- $\hat{y}_{i,c}$: probabilidad predicha para la clase c en la muestra i .

Otra razón para elegir esta función es que está disponible de forma nativa en las principales librerías de aprendizaje profundo. Esto facilita su uso y aporta garantías sobre su funcionamiento.

2.2.4. Overfitting

El sobreajuste o *Overfitting* es un fenómeno del aprendizaje profundo donde un modelo aprende tan bien los patrones del conjunto de entrenamiento que llega a memorizarlos. Como resultado, el modelo presenta un gran rendimiento con el conjunto de entrenamiento, pero en el momento en el que se le presentan nuevos datos con patrones distintos, pierde mucha capacidad predictiva.

Los primeros indicios de sobreajuste se pueden detectar cuando la pérdida del entrenamiento va bajando mientras que la de validación comienza a aumentar (ver [Figura 2.4](#)).

2.2.5. Early stopping y validación

El early stopping es una técnica que ayuda a evitar el sobreentrenamiento. Se basa en vigilar una métrica de validación durante el entrenamiento, en este caso, la pérdida. Si la métrica deja de mejorar tras varias épocas seguidas, el proceso se detiene. Así se previene que el modelo se ajuste demasiado a los datos de entrenamiento y pierda capacidad para generalizar (ver [Figura 2.4](#)).

En este proyecto, se ha trabajado con `EarlyStoppingCallback` de la librería `transformers`, estableciendo un límite de tres épocas sin mejora antes de detener el entrenamiento.

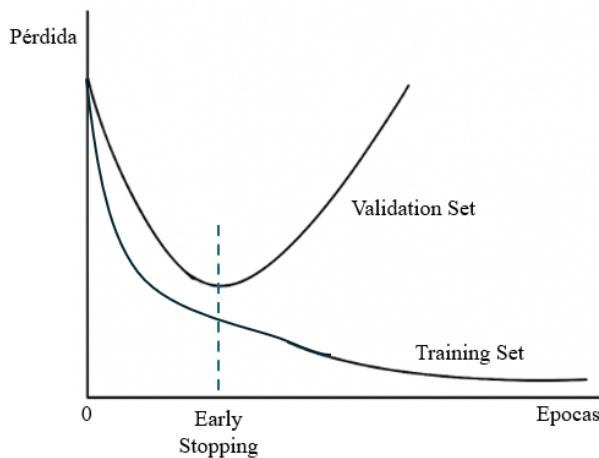


Figura 2.4: Gráfica del método Early stopping

2.2.6. Gradientes

Durante el entrenamiento del modelo, los gradientes indican cómo deben ajustarse los parámetros para reducir el error. En términos simples, muestran en qué dirección y con qué intensidad conviene modificar cada peso del modelo. Géron (2019) [10] describe el gradiente como una pendiente en una función multivariante. Su vector apunta hacia el mayor aumento posible. En aprendizaje automático, se usa para moverse en sentido contrario, buscando el punto donde el error es mínimo.

Sin embargo, los gradientes pueden volverse demasiado grandes y desestabilizar el entrenamiento. Para evitarlo, se aplica gradient clipping. Esta técnica impone un límite a su magnitud, controlado por un parámetro como `max_grad_norm`, mejorando así la estabilidad del proceso.

Por último, en entornos donde la memoria es limitada, puede usarse el parámetro `gradient_accumulation_steps`. Esto acumula los gradientes durante varios pasos, actualizando los pesos después de un número determinado de iteraciones.

Como argumentaban Goodfellow et al. [11], el conocimiento de estos parámetros resulta fundamental para mejorar el rendimiento de los modelos en tareas de aprendizaje profundo. Dan la posibilidad de explicar por qué un modelo ha funcionado mejor que otro, más allá de las métricas de evaluación.

2.3. Aprendizaje por Transferencia

El aprendizaje por transferencia es una técnica que busca aprovechar lo que un modelo ha aprendido en una tarea para aplicarlo en otra distinta pero parecida. En lugar de empezar desde cero cada vez, se toma lo aprendido antes como punto de partida. Esto hace que el entrenamiento en la nueva tarea pueda hacerse con menos datos, menos tiempo y, a veces, mejores resultados.

La idea parte de algo bastante intuitivo. Las personas hacemos esto todo el tiempo: lo que aprendemos en un contexto nos sirve en otro. El modelo, en este caso, entrena primero en una tarea de origen y luego reutiliza parte de ese conocimiento para enfrentarse a otra.

Lo habitual es trabajar con modelos que ya han sido entrenados con grandes cantidades de datos. Una vez que están bien ajustados, se pueden adaptar a tareas nuevas con muy pocas modificaciones. Es una forma eficiente de extender lo que ya funciona, sin tener que volver a construirlo todo desde el principio (ver Figura 2.5). [12].

Durante el desarrollo de este proyecto, se ha aplicado la técnica de aprendizaje por transferencia para aprovechar los modelos generativos preentrenados LLaMa y Falcon.

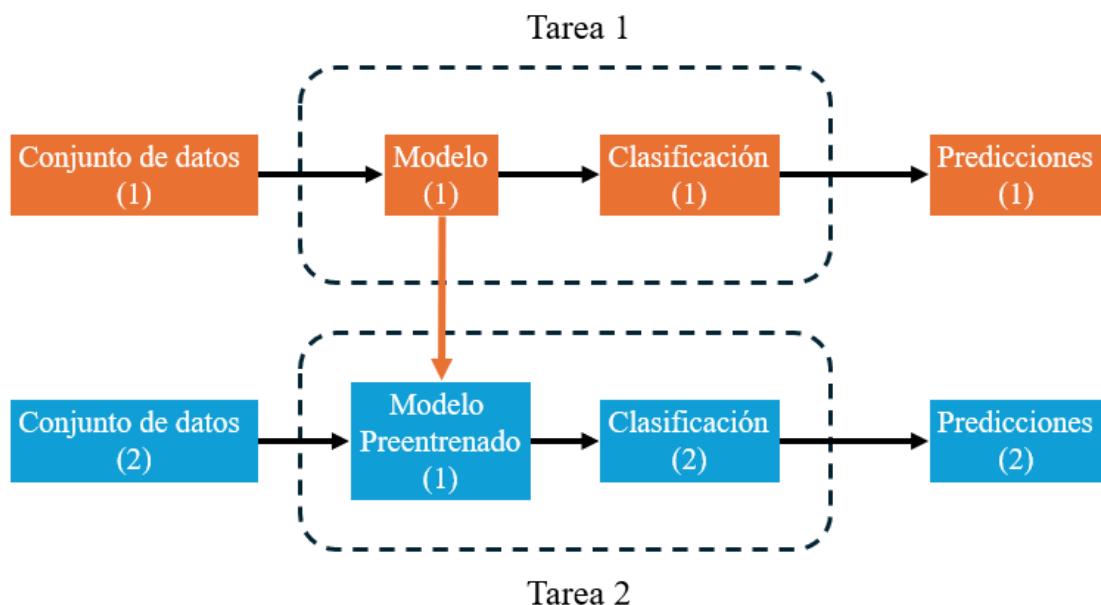


Figura 2.5: Diagrama de funcionamiento del Aprendizaje por Transferencia

2.4. Procesamiento del Lenguaje Natural

El Procesamiento del Lenguaje Natural (PLN) es un área de la inteligencia artificial que se ocupa de buscar cómo las máquinas pueden comprender y manipular texto en lenguaje natural para realizar tareas útiles [13]. En el contexto de este trabajo, el PLN se emplea para analizar el texto presente en solicitudes del usuario y detectar patrones que determinen cuál es la intención.

El PLN ha cambiado mucho en los últimos años. Parte de ese cambio se debe a los modelos de lenguaje basados en arquitecturas profundas, como los Transformers. Estos modelos permiten entender y representar el texto de una forma más rica gracias a las redes neuronales. Transforman las palabras en vectores que capturan tanto su significado como su función dentro de la frase, haciendo posible que los sistemas trabajen con el lenguaje de forma más precisa. [14]

2.4.1. Preprocesamiento del texto

El uso de redes neuronales para el procesamiento del lenguaje natural da lugar a la necesidad de buscar una representación numérica para los datos. Para entender esto, comenzaremos analizando la unidad a procesar: el **token**.

Los tokens pueden estar formados por caracteres, palabras o subpalabras. Llegados a este punto, se podría pensar que asociando una etiqueta numérica a cada token, estas unidades de procesamiento ya podrían ser tratadas por la red. Sin embargo, esto puede presentar un problema, ya que la red interpretaría estos valores cuantitativamente a nivel geográfico, pese a simplemente ser los identificadores de los tokens. Es decir, si hay un token con valor numérico '2' y otro con valor '8', la red neuronal podría deducir que un token con valor '4' está al doble de distancia del segundo respecto al primero. Para solucionarlo, se presenta 'one-hot encoding'.

Esta técnica de representación asigna un vector a cada token en lugar de una etiqueta numérica. Este vector tiene tantas posiciones como tokens se deseen tener en el vocabulario, y cada una de sus posiciones representará a cada token del mismo. La idea será marcar con un '1' la posición que indique el tipo de token que estamos representando, llenando con ceros el resto del vector. De esta forma, se creará un espacio de tantas dimensiones como tokens haya en el vocabulario, quedando todos a la misma distancia (ver [Figura 2.6](#)) [15].

A nivel práctico, se desaprovecha mucho espacio ya que el resto del vector queda vacío. La compactación de información es llevada a cabo por la red neuronal mediante el procesamiento de datos. Es por ello que trabajamos con el aprendizaje por transferencia para utilizar modelos ya preentrenados en transformar representaciones de one-hot encoding en vectores compactados. A este tipo de representaciones se les denomina **Word**

Embeddings[16].

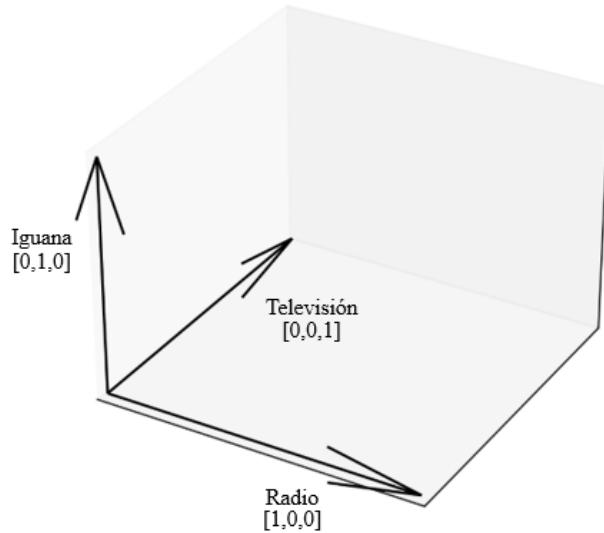


Figura 2.6: Representación espacial one-hot encoding para un vocabulario con tres tokens.

2.4.2. Representación del texto

Antes, los modelos representaban el texto usando métodos simples como bolsas de palabras o TF-IDF. Estas técnicas contaban qué palabras aparecían, pero no tenían en cuenta el orden ni el contexto. Ahora se usan representaciones densas, conocidas como *Word Embeddings*. Estos vectores permiten que los modelos capten mejor el significado de las palabras y sus relaciones dentro de una frase.

- **Word2Vec:** [17] Se trata de un algoritmo NLP publicado en 2013 que genera *Word Embeddings* mediante una red neuronal. Se basa en dos arquitecturas (ver Figura 2.7):
 - **Continuous Bag of Words:** El algoritmo predice la palabra en base a sus palabras vecinas, es decir, al contexto. Es eficiente cuando el conjunto de datos no es muy grande.
 - **Skip-gram:** Aquí el algoritmo hace justo lo contrario. Predice las palabras circundantes (el contexto) a partir de una única palabra de entrada. Esta arquitectura es más efectiva con conjuntos de datos extensos, ya que cada par palabra-contexto se procesa de forma independiente.
- **GloVe:** [18] Lanzado un año más tarde, GloVe es un modelo que aprende a generar *Word Embeddings* a partir de datos de coocurrencia y relaciones semánticas.

Parte de la matriz de coocurrencia, que recoge con qué frecuencia aparece una palabra junto a otra dentro de una ventana de texto. El tamaño de esa ventana se define al principio y se desliza por el corpus. Cada vez que una palabra aparece cerca de otra,

se suma al recuento. Sin embargo, no todas las posiciones cuentan igual. Las palabras más cercanas tienen más peso, reflejándose así en la forma en que se construye la matriz.

- **ELMo:** [19] Los *Embeddings from Language Models* o ELMo significaron un importante cambio frente a sus predecesores. A diferencia de Word2Vec o GloVe, ELMo utiliza un modelo basado en redes neuronales bidireccionales (BiLSTM). De esta manera y tomando tokens a nivel de caracteres, estos modelos son capaces de generar diferentes representaciones para una misma palabra según su uso en la oración.

Los modelos como Word2Vec, GloVe y, más recientemente, BERT y sus derivados, permiten obtener representaciones contextuales del texto. En este trabajo, se utilizan embeddings obtenidos mediante modelos como *DistilBERT* o *RoBERTa* para tareas de clasificación.

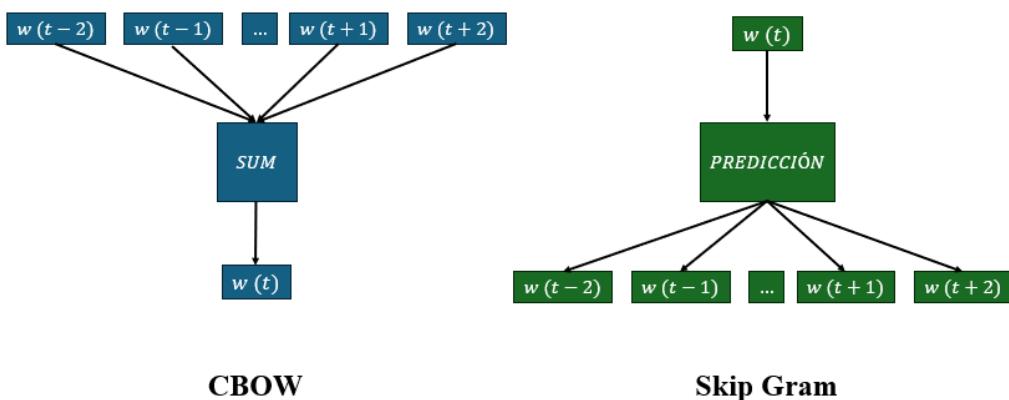


Figura 2.7: Diagrama de comparación entre arquitecturas CBOW y Skip Gram

2.4.3. Clasificación de texto en PLN

La clasificación de texto es una tarea fundamental en PLN. Puede ser binaria, multiclasa o multietiqueta, según cuántas categorías se quieran predecir y cómo se relacionen entre sí. En este trabajo se aplica para analizar solicitudes del usuario. La idea es comprender la intención que busca dicha petición y actuar en base a ello.

2.5. Transformers

2.5.1. Redes Neuronales Recurrentes (RNN)

Un procesamiento secuencial de los datos obliga a recordar resultados anteriores. Las **Redes Neuronales Recurrentes** [20] (*Recurrent Neural Networks*) son un tipo de red neu-

ronal y poseen ciclos que les permiten transmitir información hacia sí mismas, tomando en cuenta entradas previas. De esta forma, son capaces de mantener el contexto durante el tratamiento secuencial de datos (ver [Figura 2.8](#)).

Sin embargo, esta especie de memoria se ve afectada cuando la secuencia es potencialmente larga. Los estados que ocurrieron mucho antes en la secuencia dejan de contribuir al estado actual. En otras palabras, la red pierde la memoria de las entradas pasadas lejanas en la secuencia. Este problema es lo que se conoce como **desvanecimiento de gradiente**.

Para solucionar esto, surgen las redes **LSTM** [20] (*Long-Short Term Memory*), que emplean celdas con puertas (forget, input, output) para controlar qué información se recuerda o se olvida. Estas puertas gestionan el flujo de información a una celda de memoria, permitiendo que la red retenga dependencias a largo plazo.

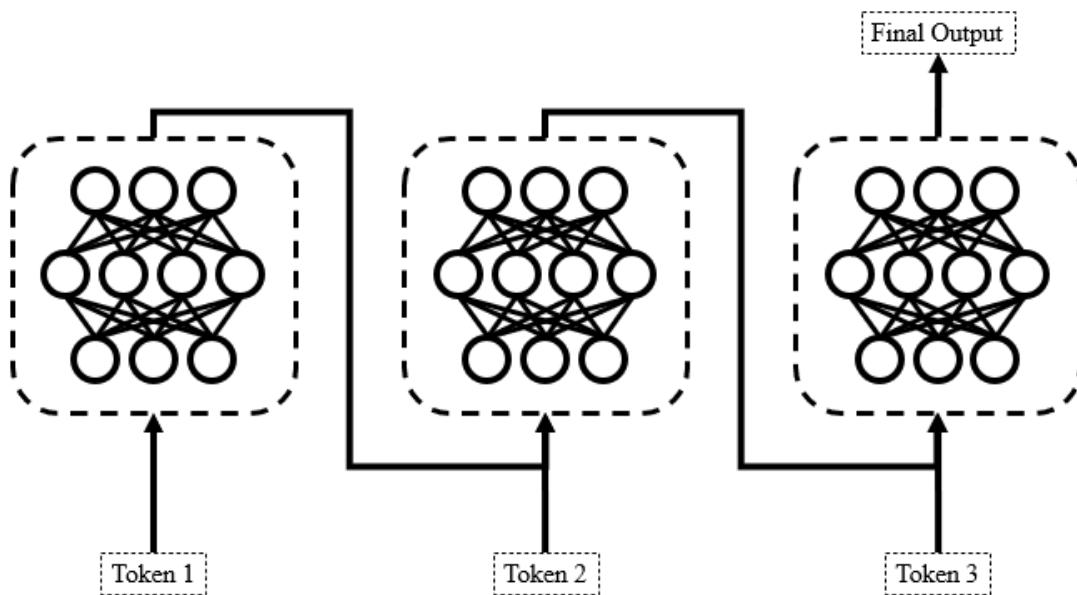


Figura 2.8: Arquitectura de una RNN

2.5.2. Fundamentos de los Transformer

Los Transformers surgieron de la mano de Vaswani et al. en 2017 bajo el lema 'Attention is all you need' [21]. Estas arquitecturas nacieron como alternativa a los modelos recurrentes tradicionales (RNN, LSTM). Mientras que estos modelos funcionan con el procesamiento secuencial de los datos, los Transformers se basan en los mecanismos de atención y en el procesamiento en paralelo.

Los **mecanismos de atención** [22] [23] permiten que el modelo analice varias partes de la secuencia de entrada al mismo tiempo. Esto le ayuda a captar relaciones que pueden estar lejos entre sí, tanto en el sentido semántico como en la estructura de la frase. La

idea central es calcular cuánta atención debe prestar cada palabra a las demás palabras de la secuencia. Algunas serán más relevantes que otras, y eso depende del contexto. Para hacerlo, cada token genera tres vectores: uno para definir lo que un token busca en otros tokens (**consulta** o *query*), otro para describir cómo un token se define a sí mismo (**clave** o *key*) y otro para indicar lo que un token aporta a otros tokens (**atención** o *attention*). El modelo compara las consultas con las claves usando un producto escalar, dando lugar a una puntuación. Después, esas puntuaciones se normalizan con softmax, y el resultado (vector atención) indica qué tanto debe atender una palabra a las demás (ver [Figura 2.9](#)).

Además, los Transformers se caracterizan por el **procesamiento en paralelo**. Al prescindir de las RNNs y su recurrencia, se pierde la posición de los tokens en el texto. Como solución, se implementa la **codificación posicional**. Se trata de un mecanismo que permite introducir los tokens de manera simultánea sin perder el orden inherente de la frase, ya que mediante unas combinaciones de funciones seno y coseno, asigna una posición única a cada token dentro de la secuencia.

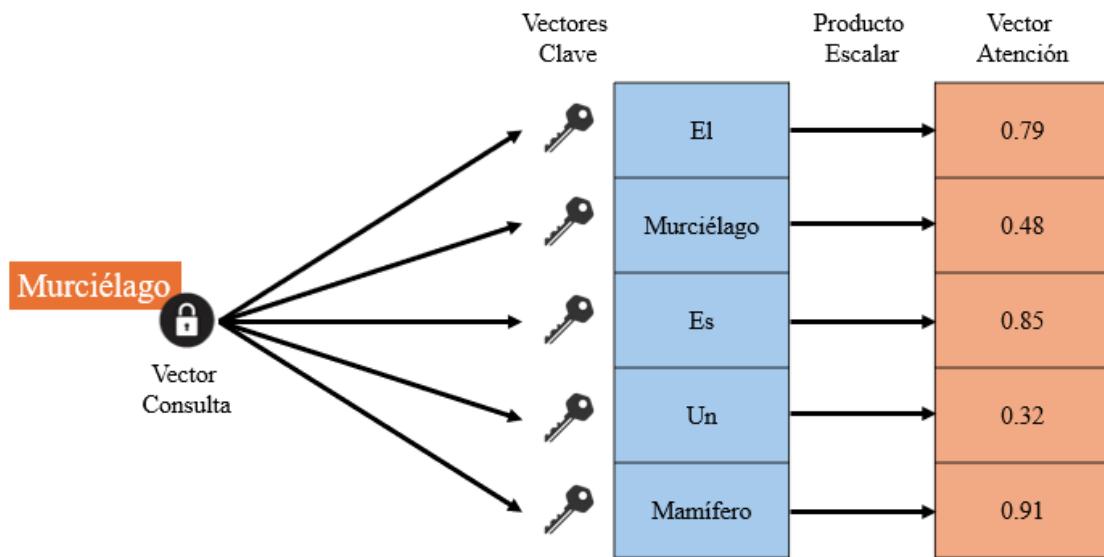


Figura 2.9: Procesamiento del vector atención del token murciélagos

2.5.3. Arquitectura de los Transformers

La arquitectura transformer [22] [21] típicamente se compone de dos bloques principales: el codificador (encoder) y el decodificador (decoder).

- **Codificador:** Esta etapa genera representaciones contextuales de cada elemento de la secuencia de entrada, elaborando una representación para cada elemento en función de los demás presentes en la secuencia, capturando relaciones sintácticas y semánticas. Está formado por varias capas que comparten la misma estructura.

- **Decodificador:** Esta etapa se encarga de generar nuevas secuencias de salida utilizando las representaciones contextualizadas que produce el codificador (en arquitecturas encoder-decoder) o basándose en la entrada y su propio estado interno (en arquitecturas decoder-only). Al igual que el decodificador, lo forman múltiples capas con la misma estructura.

Las arquitecturas que implementan ambas partes se conocen como encoder-decoder transformers o modelos sequence-to-sequence ([Figura 2.10](#)). Sin embargo, en los últimos años también se han desarrollado arquitecturas basadas únicamente en el codificador (encoder-only transformers) o en el decodificador (decoder-only transformers).

Los Transformers han dado lugar a una nueva generación de modelos de lenguaje pre-entrenados, entre ellos, los modelos de lenguaje generativos pertenientes a este trabajo. Estos modelos se entrena con grandes volúmenes de texto mediante tareas de aprendizaje no supervisado y luego se adaptan mediante *fine-tuning* a tareas específicas.

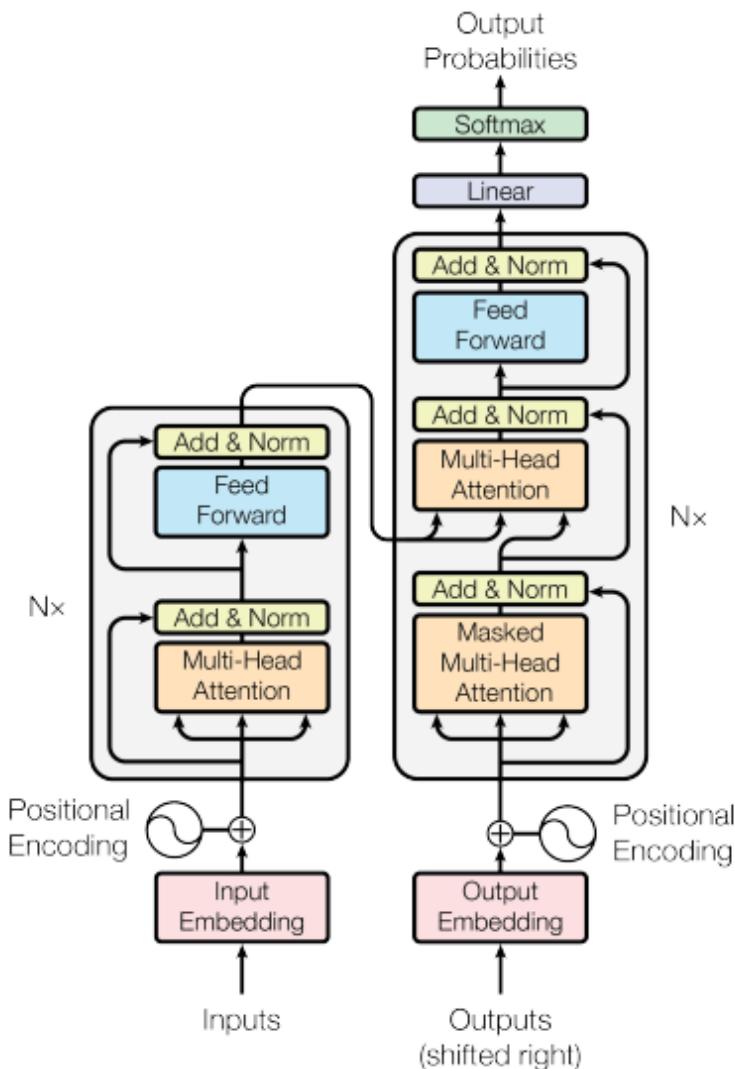


Figura 2.10: Arquitectura de un Transformer

2.6. Modelos Generativos de Lenguaje

Los Modelos Generativos de Lenguaje (LLMs, por sus siglas en inglés) [22] son una forma avanzada de inteligencia artificial diseñada para trabajar con texto natural. Se entrena utilizando grandes cantidades de texto sin etiquetas explícitas y emplean técnicas de *Deep Learning* para predecir la siguiente palabra en una secuencia. Son un subtipo de Transformer compuestos únicamente por el decodificador. Así pues, el proceso de generación de texto se realiza de manera **autorregresiva**, lo que significa que el modelo genera un token a la vez, basándose en el contexto previo.

Modelos como LLaMa o Falcon han demostrado ser herramientas poderosas en tareas como clasificación, resumen o traducción. En este proyecto, nos centraremos en su papel en la clasificación.

2.6.1. Zero-shot, Few-shot y Fine-tuning

La forma en que se les indica a los LLMs qué generar (a través de diferentes tipos de prompting) influye significativamente en la calidad y relevancia de sus resultados. Existen tres modos principales de uso:

- **Zero-Shot:** [24] En este escenario, el modelo únicamente recibe la instrucción, sin ejemplos adicionales, y se espera que genere una respuesta basada en su conocimiento general adquirido durante el entrenamiento. Por ejemplo:

¿El usuario busca añadir una canción, eliminar una canción, ver su lista de reproducción o eliminar todas sus canciones?

"Guarda 'Creep' de Radiohead en mi playlist". Responde: add, remove, view o clear.

- **Few-Shot:** [24] Aquí el modelo recibe algunos ejemplos de la tarea antes de la instrucción. Estos ejemplos sirven como una guía para que el modelo comprenda el formato, el tipo de respuesta esperada y el contexto específico de la tarea. Por ejemplo:

Ejemplo 1: Incluye 'Let It Be' de The Beatles → add

Ejemplo 2: Quita 'Always' de mi lista de reproducción → remove

Ejemplo 3: Enséñame mis canciones → view

Ejemplo 4: ¿Podrías limpiar mi playlist? → clear

Frase: Guarda 'Creep' de Radiohead en mi playlist →

- **Fine-tuning:** Este proceso trata de ajustar los pesos del modelo sometiéndolo previamente a un entrenamiento con un conjunto de datos específico. Esta técnica requiere más recursos pero genera los mejores resultados. Se analizará más en detalle en la próxima sección.

Durante el desarrollo de este proyecto, se ha trabajado con las tres metodologías, focalizando más en el *zero-shot* y el *fine-tuning*, con el fin de llevar a cabo una comparativa y determinar qué tanto modifican las respuestas del modelo el uso de una estrategia u otra.

2.6.2. Ventajas y limitaciones

Como ventajas, los modelos generativos presentan una rápida capacidad de adaptación a nuevas tareas, sin necesidad de datasets específicos.

Sin embargo, también presentan limitaciones. Los LLMs requieren una enorme cantidad de datos y recursos computacionales, lo que conlleva costos significativos. Además, no siempre son consistentes; pueden ser sensibles al ‘prompt’, específicamente en tareas como la clasificación multiclas de textos. [24]

2.7. Fine-tuning y optimización de hiperparámetros

El ajuste fino, o *fine-tuning*, [25] es una técnica común en aprendizaje profundo. Parte de un modelo que ya ha sido entrenado y se adapta a una tarea concreta usando un conjunto de datos más pequeño. Se basa en el aprendizaje por transferencia, es decir, se aprovecha lo que el modelo ya ha aprendido y se ajusta solo lo necesario para el nuevo problema (ver Figura 2.11).

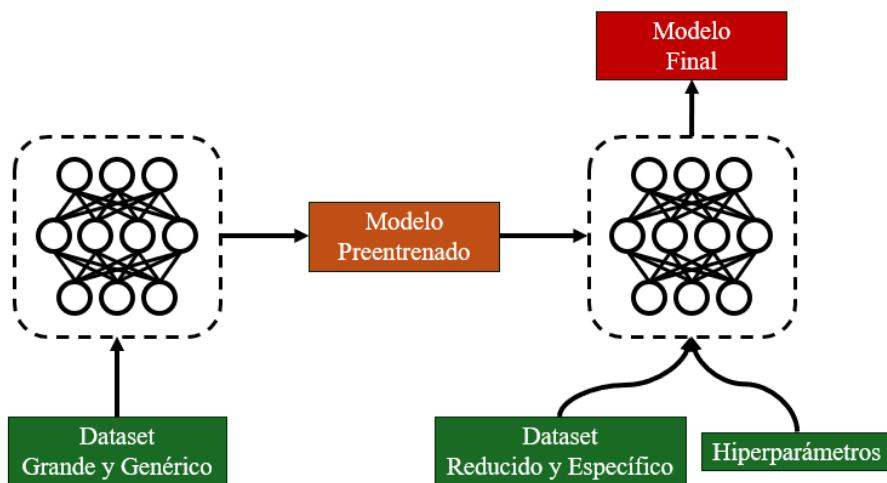


Figura 2.11: Diagrama de fine-tuning a un modelo

Entrenar modelos profundos desde cero suele requerir muchos datos y una cantidad considerable de recursos computacionales. En la práctica, eso no siempre está al alcance. Se convierte en algo mucho más eficiente el entrenar modelos grandes en tareas generales, para luego solo tener que concretar mediante ajuste fino. Gracias a esto, el tiempo de

entrenamiento se reduce de forma notable. Lo que antes podía tomar semanas o meses, ahora puede resolverse en cuestión de horas.

La optimización de hiperparámetros también resulta clave en la eficiencia de los recursos. Se tratan de configuraciones ajustables de un modelo de aprendizaje automático que se establecen antes de que comience el entrenamiento. Tienen un papel fundamental, afectando tanto al proceso como al resultado final. Elegir bien los valores (como la tasa de aprendizaje, el tamaño del lote, el número de épocas de entrenamiento, etc.) puede marcar una diferencia importante en el rendimiento del modelo.

2.8. Modelos de Aprendizaje Empleados

Para este proyecto, se han empleado dos modelos de lenguaje generativos que resolverán la tarea de clasificación multiclase de texto en la que el sistema debe categorizar cada solicitud de usuario como una de las siguientes intenciones: *add* (añadir), *remove* (eliminar), *view* (visualizar) o *clear* (vaciar).

Los modelos seleccionados han sido LLaMa y Falcon, ambos pertenecientes a la familia de modelos de lenguaje generativos basados en la arquitectura Transformer:

- **Falcon-7b:** Se trata de un modelo desarrollado por el *Technology Innovation Institute* de Abu Dhabi, y se encuentra disponible en la plataforma Hugging Face¹. Cuenta con unos 7 mil millones de parámetros, lo que le permite captar relaciones complejas en grandes conjuntos de texto y adaptarse a distintas tareas de procesamiento del lenguaje natural.

Una característica particular de este modelo es que se pre-entrenó en su mayoría con datos web del corpus REFINEDWEB [26]. Esto pone en cuestión la idea habitual de que, para lograr buen rendimiento y generalización, los modelos necesitan también datos curados como libros o artículos técnicos, además de datos de la web.

- **Llama-3.2-1B-Instruct:** LLaMa es una familia de modelos generativos desarrollada por Meta y también disponible en HuggingFace². En este trabajo se ha utilizado una variante de la serie, llamada Llama-3.2-1B-Instruct, que cuenta con unos 1.200 millones de parámetros. Su tamaño intermedio permite encontrar un buen equilibrio entre el rendimiento, el consumo de recursos y la facilidad para ajustarlo a tareas concretas.

Este modelo está orientado tanto a la generación de texto, como a tareas de comprensión y clasificación, y concretamente, la variante *Instruct* está optimizada para seguir instrucciones y sostener diálogos, lo que la hace especialmente útil en asistentes conversacionales.

¹<https://huggingface.co/tiiuae/falcon-7b>

²<https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct>

Ambos modelos, además, son de código abierto y cuentan con el respaldo de la comunidad científica. Esto facilita su uso y permite experimentar con ellos. Su capacidad para generar representaciones contextuales los hace especialmente útiles para tareas como la clasificación de intenciones en lenguaje natural, que es fundamental en este asistente musical.

2.9. Medidas de Evaluación

Con el objetivo de evaluar los modelos de clasificación desarrollados en el proyecto, se han sometido a una serie de métricas. De esta manera, se obtienen unas medidas con las que interpretar el rendimiento del sistema. En este caso, en la interpretación de solicitudes y clasificación multiclas (add, remove, view, clear).

2.9.1. Accuracy

Se trata de la proporción de predicciones correctas entre el total de predicciones realizadas. Es una medida generalizada sobre qué tan bien clasifica el modelo. Su fórmula es:

$$\text{Accuracy} = \frac{\text{Número de predicciones correctas}}{\text{Número total de muestras}} = \frac{\sum_{i=1}^n (y_i = \hat{y}_i)}{N} \quad (2.2)$$

Siendo:

- N : Número total de muestras.
- y_i : Etiqueta real.
- \hat{y}_i : Etiqueta predicha.

2.9.2. Precisión

De todas las predicciones positivas realizadas para una clase, la precisión mide cuántas fueron correctas. Al ser un problema multiclas, la medición será para 4 clases:

$$\text{Precision}_c = \frac{TP_c}{TP_c + FP_c} \quad (2.3)$$

Siendo:

- $Precision_c$: Precisión en la clase c .
- TP_c : Número de verdaderos positivos para la clase c (veces que el modelo predijo clase c correctamente).

- FP_c : Número de falsos positivos para la clase c (veces que el modelo predijo clase c pero era otra).

2.9.3. Exhaustividad (recall)

De todos los casos reales de una clase, la exhaustividad o *recall* mide cuántos fueron detectados correctamente. De la misma forma que con la precisión, la medición será para 4 clases:

$$\text{Recall}_c = \frac{TP_c}{TP_c + FN_c} \quad (2.4)$$

Siendo:

- $Precision_c$: Recall en la clase c .
- TP_c : Verdaderos positivos para la clase c (el modelo predijo correctamente esa clase).
- FN_c : Falsos negativos para la clase c (el modelo no predijo esa clase cuando debería haberlo hecho).

2.9.4. F1-Score

El F1-score no es más que la media armónica entre la precisión y el recall de una clase. Es decir, sirve para equilibrar qué tan bien predice esa clase cuando la elige (precisión) y qué tan bien la reconoce cuando realmente aparece (recall). Su fórmula, también aplicada a 4 clases en este caso, es:

$$F1_c = 2 \cdot \frac{Precision_c \cdot Recall_c}{Precision_c + Recall_c} \quad (2.5)$$

Siendo:

- $F1_c$: F1-Score en la clase c .
- $Precision_c$: Precisión en la clase c .
- $Recall_c$: Recall en la clase c .

2.9.5. Macro-promedio

El macro-promedio (macro-average) es la media aritmética simple de una métrica (precisión, recall y F1) calculada por clase. Esta medida hace que cada clase tenga el mismo

peso, por lo que resultará útil en este proyecto, ya que el número de muestras aportadas para cada clase es el mismo.

$$\text{Precision}_{macro} = \frac{1}{n} \sum_{c=1}^n \text{Precision}_c \quad (2.6)$$

$$\text{Recall}_{macro} = \frac{1}{n} \sum_{c=1}^n \text{Recall}_c \quad (2.7)$$

$$\text{F1}_{macro} = \frac{1}{n} \sum_{c=1}^n \text{F1}_c \quad (2.8)$$

2.9.6. Matriz de confusión

La matriz de confusión es una tabla que permite visualizar cuántas veces el modelo predijo correctamente cada clase y cuántas veces se equivocó al confundirla con otra.

Como se puede ver en la [Figura 2.12](#), cada fila representa la clase real y cada columna la clase predicha. El valor de cada celda indica cuántas veces ocurrió esa combinación.

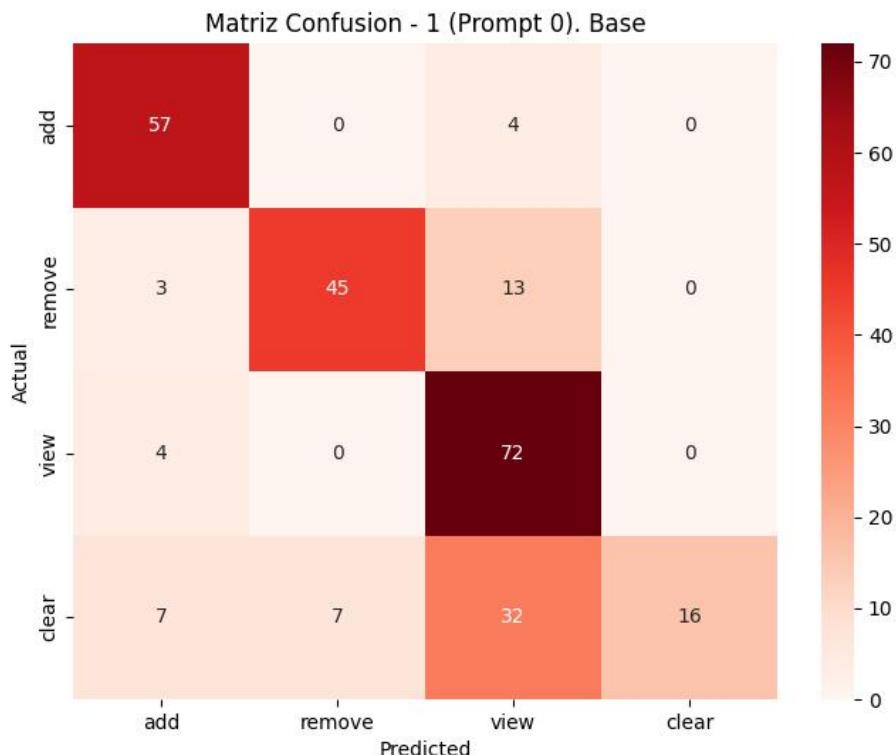


Figura 2.12: Ejemplo de una matriz de confusión

2.10. Trabajos relacionados

Existen varios trabajos previos que exploran el uso de sistemas de recomendación y asistentes virtuales en el ámbito musical, con el objetivo de ofrecer soluciones más personalizadas. Un ejemplo relevante es el estudio de Shadan Shameli (2023) [27], que aborda la creación de un chatbot musical diseñado para proporcionar una experiencia interactiva, en contraste con las interfaces tradicionales. Utiliza técnicas de Procesamiento de Lenguaje Natural (PLN) para interpretar la entrada de texto del usuario. Reformula el problema como una tarea de clasificación de intenciones, asociando a cada solicitud un artista específico, y aplicando y comparando diferentes métodos de aprendizaje automático supervisado, como Árbol de Decisión, Naive Bayes, KNN, Bosque Aleatorio y SVM.

En el trabajo de Ye Chang (2023) [28], se presentó un chatbot especializado en recomendaciones musicales basado en Dialogflow. El sistema fue capaz de interactuar con los usuarios a través de texto, proporcionando información sobre artistas, canciones y géneros musicales, similar a las funcionalidades de recomendación implementadas en este proyecto. Sin embargo, el enfoque de este trabajo se centró en la integración con Spotify API y no utilizó modelos generativos como LLaMA y Falcon para procesar el lenguaje natural.

El uso de modelos generativos está presente en trabajos como el de Víctor Chumillas Hernando (2024) [29], que explora la aplicación de tecnologías LLM, para desarrollar un chatbot especializado capaz de responder consultas sobre la Universidad Politécnica de Madrid (UPM). Este trabajo utiliza algoritmos de aprendizaje automático, arquitecturas de red neuronal como transformers, técnicas de generación de texto, y el uso de embeddings y bases de datos vectoriales para comprender el contexto de las preguntas y generar respuestas relevantes y personalizadas basadas en documentos específicos cargados por el usuario. No obstante, presenta un enfoque orientado a la generación de texto y no a la clasificación.

Este proyecto se distingue al integrar modelos generativos avanzados en un problema de clasificación multiclasa y sobre un chatbot conversacional. El agente musical actuará en base a las predicciones de LLaMA y Falcon, modelos que no están específicamente diseñados para una tarea de clasificación. Así pues, este trabajo busca contribuir a la investigación analizando cómo es viable el uso de estos modelos para esta tarea.

2.11. Tecnologías y Recursos Utilizados

A lo largo del proyecto, se ha trabajado con diferentes plataformas, herramientas y bibliotecas que han resultado ser fundamentales en su desarrollo. La [Figura 2.13](#) muestra los logos de los principales de estos recursos.



Figura 2.13: Logos de las herramientas utilizadas.

A continuación, se citan y describen estos elementos:

2.11.1. Python

Existen numerosos lenguajes de programación en los que poder trabajar con aprendizaje automático. En este caso, se ha elegido Python³.

Dispone de una gran cantidad de librerías y frameworks específicos del campo del aprendizaje automático, por no hablar de que, al ser tan famoso, es fácil encontrar documentación, ejemplos y soluciones a casi cualquier problema. Se ha utilizado para la creación de scripts de procesamiento, entrenamiento y evaluación de los modelos de lenguaje generativos.

2.11.2. Google Colab

El entorno de desarrollo elegido ha sido Google Colab.⁴

Se trata de un entorno de programación online proporcionado por Google. Gracias a las rápidas GPUs, permite la aceleración de la ejecución del código. Además, al ser una

³<https://www.python.org/>

⁴<https://colab.research.google.com/>

plataforma y subirse los archivos a la nube, su accesibilidad es mucho más flexible desde cualquier dispositivo.

2.11.3. Hugging Face

Hugging Face⁵ es una comunidad de desarrolladores que dispone de herramientas para trabajar con modelos de aprendizaje automático ya entrenados. A través de sus APIs, es posible descargar y utilizar estos modelos de forma sencilla, sin necesidad de configuraciones complejas; en este caso, LLaMA y Falcon.

2.11.4. GitHub

GitHub⁶ es una plataforma de desarrollo colaborativo que permite gestionar proyectos de software utilizando el sistema de control de versiones Git. En este proyecto, se ha utilizado GitHub para trabajar con bibliotecas como DialogueKit y para la integración de diferentes librerías utilizadas a lo largo del desarrollo del sistema.

2.11.5. LoRA

LoRA⁷ es una técnica pensada para ajustar modelos de lenguaje grandes sin necesidad de modificar todos sus parámetros. En vez de actualizar la red completa, se añaden matrices pequeñas y entrenables en ciertas capas. Esto reduce el uso de memoria y el costo computacional durante el entrenamiento.

En este TFG, se ha utilizado LoRA (de la librería peft) durante el *Fine Tuning* de LLaMa y Falcon, reduciendo significativamente el uso de memoria GPU durante la ejecución.

2.11.6. Optuna

Optuna⁸ es una herramienta de optimización utilizada para ajustar los valores de los hiperparámetros empleados durante el entrenamiento de los modelos, de manera que sean los más eficientes para lograr el objetivo.

⁵<https://huggingface.co/>

⁶<https://github.com/>

⁷https://huggingface.co/docs/peft/package_reference/lora

⁸<https://optuna.org/>

2.11.7. SQL

SQL (Structured Query Language)⁹ es un lenguaje estándar para trabajar con bases de datos relacionales. En este proyecto se ha utilizado SQLite como motor ligero para almacenar y consultar la información relacionada con canciones.

2.11.8. Flask / Dialoguekit

Durante el desarrollo del ChatBot se utilizaron dos frameworks principales: Flask¹⁰ y DialogueKit¹¹.

Flask es un microframework en Python que ha servido para conectar el programa principal con las diferentes partes del ChatBot conversacional. Por otro lado, DialogueKit es una biblioteca diseñada para facilitar la creación de agentes conversacionales. Fue empleada para el diseño del asistente musical y para la gestión de la comunicación cliente-servidor en tiempo real.

En resumen, este capítulo recoge los conceptos necesarios para entender el enfoque seguido en el trabajo. Estos conocimientos han sido clave para tomar las decisiones técnicas y metodológicas que se explican en los próximos capítulos.

⁹<https://www.sqlite.org/>

¹⁰<https://flask.palletsprojects.com/>

¹¹<https://github.com/iai-group/DialogueKit>

CAPÍTULO 3

Metodología, Experimentación y Resultados

En este capítulo, se llevará a cabo un análisis detallado sobre la metodología empleada para tratar el problema de clasificación de intenciones, y su consiguiente implementación en un chatbot conversacional. Desde la preparación del conjunto de datos y procesamiento de texto hasta el entrenamiento de los modelos de lenguaje generativos escogidos; pasando por la implementación del modelo NER para la identificación de entidades. Finalmente, se analizan los resultados obtenidos mediante distintas métricas de evaluación con el fin de valorar el rendimiento del sistema propuesto.

3.1. Descripción General de la Metodología

El marco de experimentación de este trabajo se ha estructurado en varias fases, las cuales abarcan desde el preprocesamiento de datos hasta la evaluación y análisis de los resultados obtenidos de los modelos una vez ajustados y entrenados. Finalmente, los modelos se implementan en el chatbot conversacional.

El proyecto comienza con el **preprocesamiento de los datos**. Aquí se prepara el conjunto de frases de entrada mediante normalización textual y tokenización, para adaptarlo al formato que esperan los modelos. Después se llevó a cabo la **selección de las arquitecturas base**. En este caso, se utilizan LLaMa y Falcon, los cuales son modelos de código abierto, están basados en Transformers y se pueden ajustar bien a tareas de clasificación de texto.

La siguiente fase consiste en el **ajuste de los hiperparámetros**. Se fijan el número de épocas, la tasa de aprendizaje, el tamaño del batch y técnicas como weight decay o early stopping. Sus valores han sido seleccionados tras un proceso de optimización de hiperparámetros, en el que se han comparado diferentes opciones mediante Optuna. El objetivo es mejorar la capacidad del modelo para generalizar.

Una vez definidos los hiperparámetros, se realiza el **entrenamiento de los modelos** mediante fine-tuning, usando el subconjunto de datos de entrenamiento y validación. El proceso se regula con el uso de LoRA, con el objetivo de reducir el uso de memoria GPU

durante la ejecución.

Con los modelos ya entrenados, se procede a la **evaluación y análisis de los resultados**. Se usan métricas como precisión, recall, F1 y matriz de confusión. Estas medidas permiten ver qué tan bien el modelo reconoce las intenciones en las frases de entrada del subconjunto de datos de test. También se realizará un **análisis de errores** para estipular sobre las posibles causas de los errores cometidos por los modelos.

Finalmente, se lleva a cabo la **implementación de los modelos en el ChatBot conversacional**. Para ello, se desarrolla un programa en un entorno local que gestiona el agente conversacional mediante el uso de Flask y DialogueKit. A través de una API y un túnel local, se conecta con los modelos para la comunicación de las predicciones, y una base de datos desarrollada simula la gestión de las canciones. El resultado final es un simulador conversacional entre el usuario y el ChatBot.

La [Figura 3.1](#) muestra el pipeline del marco de experimentación seguido en este trabajo.

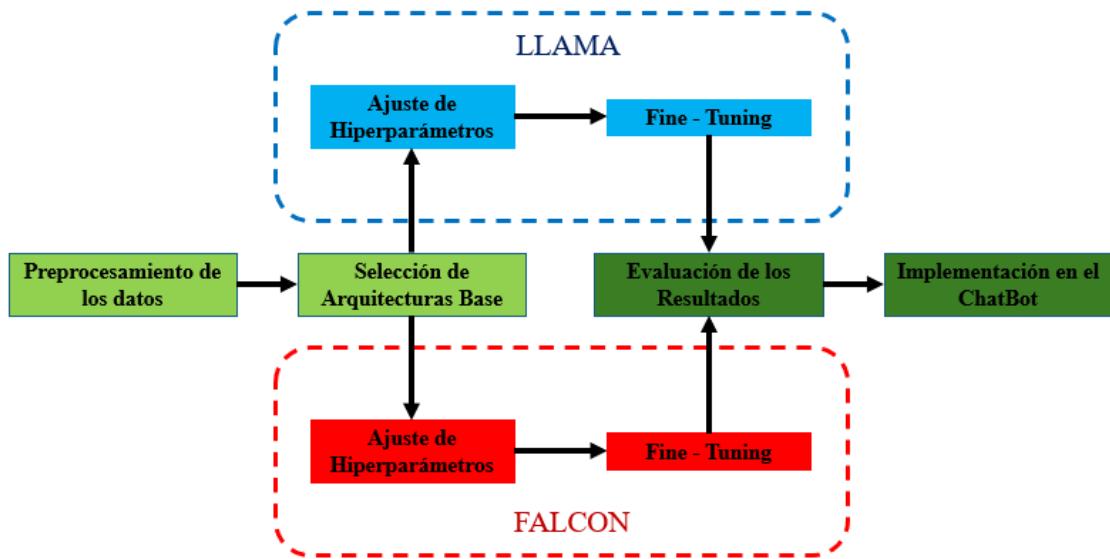


Figura 3.1: Pipeline general del marco de experimentación

3.2. Preparación de los Datos

A continuación, se detallan los datos utilizados para el entrenamiento y evaluación de los modelos. Desde su generación hasta su división en subconjuntos; pasando por su normalización y tokenización.

3.2.1. Descripción de los Datos

El dataset empleado en este proyecto ha sido generado artificialmente mediante el uso de Chat-GPT, con el objetivo de simular interacciones realistas entre usuarios y un asistente musical. Se trata de 1600 frases repartidas en grupos de 400 por cada etiqueta, las cuales pueden ser 'add', 'view', 'clear' o 'remove', dependiendo de la intención que sugiere la petición:

- **add**: el usuario desea añadir una canción a su lista de reproducción.
- **remove**: el objetivo es retirar una canción de la playlist.
- **clear**: se busca limpiar la playlist, retirando todas las canciones.
- **view**: el usuario quiere ver las canciones presentes en su lista de reproducción.

Esta estrategia permitió construir un corpus amplio y balanceado. Se ajusta bien a la tarea y evita el uso de datos etiquetados a mano. Por último, el conjunto fue almacenado en formato CSV para facilitar el procesamiento posterior.

3.2.2. División de Datos

El conjunto de datos original ha sido dividido en subconjuntos de entrenamiento, validación y test. En primer lugar, el 20 % del conjunto fue catalogado para test. El 15 % del 80 % restante fue agrupado como el subconjunto de validación, y el resto de ese 80 % como el subconjunto de entrenamiento (ver [Figura 3.2](#)).

Una distribución desigual de las clases en los subconjuntos de datos puede afectar negativamente al rendimiento y evaluación del modelo, especializándose más en aquellas clases que estén más presentes. Por ello, se ha garantizado el equilibrio de clases mediante muestreo estratificado.

3.2.3. Preparación para su uso

Una vez separados los subconjuntos de frases, se les añade una cabecera informativa necesaria para que el modelo sea capaz de interpretar la función de clasificación que debe realizar:

- **Subconjuntos de entrenamiento y validación**: aparte de la cabecera, a estos subconjuntos también se les añadirá al final la etiqueta a la que pertenece cada frase. De esta manera, el modelo podrá ir aprendiendo y entrenándose.
- **Subconjunto de test**: en este caso, solo se añadirá el prompt informativo antes de la frase. No se añade la etiqueta ya que el modelo deberá ser capaz de predecirlo. En

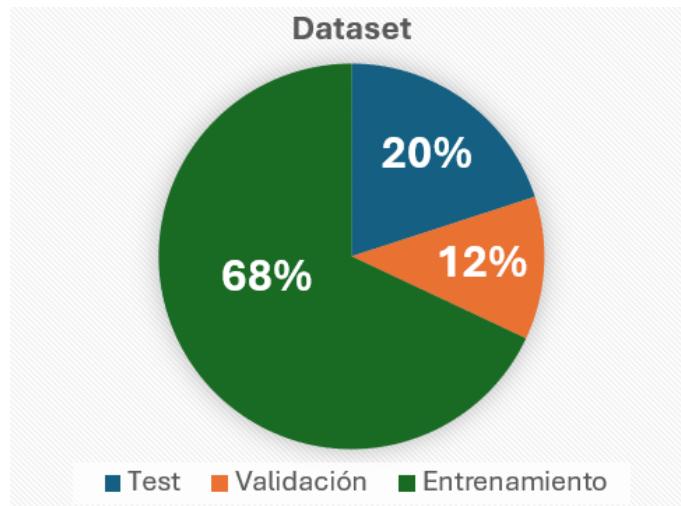
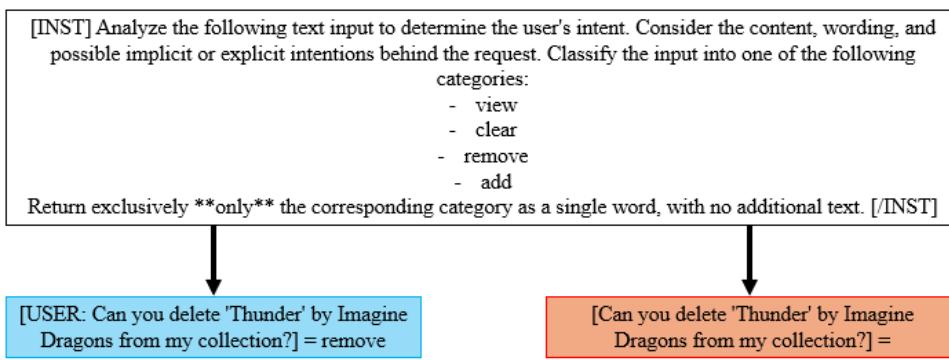


Figura 3.2: División del conjunto original de datos

lugar de eso, se almacenan por separado en otro array.

La Figura 3.3 muestra el flujo de trabajo llevado a cabo durante esta preparación de los datos para su uso.

Finalmente, estos conjuntos son convertidos a datasets compatibles con Hugging Face. Este cambio de estructura permitirá tratarlos directamente por los modelos exportados de esta comunidad de desarrolladores.



Entrenamiento y Validación

Test

Figura 3.3: Preparación de datos según su uso

3.2.4. Tokenización y Codificación

Con el objetivo de preparar el texto de entrada antes de ser procesado, se han utilizado los tokenizadores correspondientes a cada modelo (LLaMa y Falcon), cargados de Hugging Face. Todo el proceso ha sido gestionado con la clase `AutoTokenizer`, la cual permite adaptar automáticamente el tokenizador al modelo.

Así pues, para el texto:

```
'Add the song to the playlist'
```

El tokenizador de ambos modelos lo transformará en los tokens:

```
[‘_Add’, ‘_the’, ‘_song’, ‘_to’, ‘_the’, ‘_playlist’]
```

Sin embargo, su codificación numérica para **Falcon** será:

```
[1132, 345, 1045, 217, 345, 9045]
```

Mientras que para **LLaMA**:

```
[29892, 464, 6023, 373, 464, 14529]
```

Pese a que la estructura de sus tokens es similar, difieren en su codificación, ya que cada modelo cuenta con vocabulario propio.

Además, se ha configurado el uso del token especial `<eos>` (final de frase o *end of sentence*), al final de cada entrada. El `pad_token` se ha fijado también con el mismo valor, garantizando así que las secuencias estén bien formateadas cuando se aplica `padding`.

El padding permite que todas las entradas tengan la misma longitud. Cuando una frase es más corta que la longitud máxima establecida, se le añaden tokens especiales de relleno hasta alcanzar esta medida.

3.3. Selección de los Modelos

Al comienzo del proyecto se estudiaron varias arquitecturas Transformer ya preentrenadas. El objetivo era encontrar un modelo adecuado para clasificar intenciones en lenguaje natural. La decisión debía tener en cuenta tres aspectos: rendimiento en tareas similares, capacidad para el ajuste fino (o *fine tuning*) y viabilidad de uso con los recursos disponibles.

En primera instancia se consideraron modelos como `bert-base-uncased`¹ y `roberta-base`². Son modelos conocidos y se usan a menudo en tareas de clasificación. Sin embargo, en este trabajo se buscaba una estructura más orientada a modelos de lenguaje generativos, capaces de manejar texto de forma flexible.

También se evaluaron opciones más grandes como `mistral-7b-instruct`³ o variantes de LLaMA 2 con 13B parámetros⁴. Fueron descartadas por su alto consumo de memoria y el costo computacional asociado.

¹<https://huggingface.co/google-bert/bert-base-uncased>

²<https://huggingface.co/FacebookAI/roberta-base>

³<https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2>

⁴<https://huggingface.co/meta-llama/Llama-2-13b>

La elección final incluyó dos modelos generativos de código abierto. Ambos usan arquitectura autoregresiva. Se seleccionaron [meta-llama/Llama-3.2-1B-Instruct⁵](https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct) y [tiiuae/falcon-7b⁶](https://huggingface.co/tiiuae/falcon-7b).

LLaMA 3.2-1B fue elegida por ser una versión compacta. Está afinada para seguir instrucciones y puede entrenarse en entornos con GPU limitadas. Falcon-7B se incluyó como modelo complementario. Su arquitectura es más densa y grande. El objetivo era comparar su rendimiento con el de un modelo más ligero y analizar cómo influye en el entrenamiento y resultados finales.

3.4. Ajuste de Hiperparámetros

Con el objetivo de mejorar el rendimiento de los modelos, se llevó a cabo una búsqueda de los valores más eficientes para los hiperparámetros. A lo largo de esta sección, se detallarán los pasos seguidos para el ajuste de estos hiperparámetros empleados en el entrenamiento de los modelos.

3.4.1. Métodos de Búsqueda

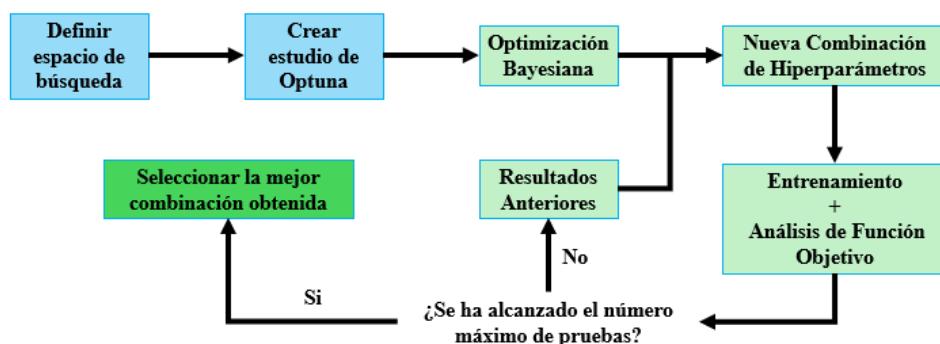


Figura 3.4: Diagrama de funcionamiento de Optuna

Para la optimización de los hiperparámetros se optó por el uso de una estrategia de búsqueda aleatoria avanzada con la herramienta **Optuna** ([Subsección 2.11.6](#)).

Optuna permite explorar el espacio de hiperparámetros usando optimización bayesiana y búsqueda guiada por resultados anteriores, reduciendo considerablemente el número de combinaciones a evaluar en comparación con métodos como la búsqueda exhaustiva, donde se evalúan sistemáticamente todas las combinaciones posibles.

En la [Figura 3.4](#) se aprecia el flujo de trabajo llevado a cabo por Optuna.

⁵<https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct>

⁶<https://huggingface.co/tiiuae/falcon-7b>

3.4.2. Hiperparámetros Evaluados

Para el proceso de ajuste, se han evaluado diferentes hiperparámetros clave que afectan al rendimiento del modelo. Se ha definido un espacio de búsqueda para cada uno, el cual define los límites de la búsqueda de combinaciones óptimas llevada a cabo por Optuna.

| Hiperparámetro | Tipos de valores | Rango de búsqueda |
|--|------------------|-------------------|
| <code>learning_rate</code> | Reales | [1e-6, 5e-6] |
| <code>max_grad_norm</code> | Reales | [0.4, 5.0] |
| <code>weight_decay</code> | Reales | [0.0, 0.3] |
| <code>warmup_ratio</code> | Reales | [0.0, 0.3] |
| <code>per_device_train_batch_size</code> | Enteros | [4, 8] |
| <code>gradient_accumulation_steps</code> | Enteros | [1, 4] |

Tabla 3.1: Espacio de búsqueda de Optuna para los hiperparámetros

La [Tabla 3.1](#) muestra estos hiperparámetros, así como los rangos de búsqueda definidos para cada uno de ellos. Fueron seleccionados debido a su impacto directo en la capacidad de generalización y precisión durante el entrenamiento.

El hiperparámetro `learning_rate`, por ejemplo, es crucial para controlar la magnitud de los ajustes realizados en cada iteración del entrenamiento. El `max_grad_norm` regula el valor máximo del gradiente, mientras que `weight_decay` contribuye a la regularización del modelo, evitando el sobreajuste.

El `warmup_ratio` es un hiperparámetro que determina el número de pasos durante los cuales el learning rate aumenta de manera gradual al comienzo del entrenamiento, permitiendo que el modelo se estabilice antes de aplicar el learning rate completo. El `per_device_train_batch_size` define el número de ejemplos procesados por cada dispositivo (en este caso, la GPU) en una sola iteración. Ajustar este parámetro correctamente ayuda a optimizar la utilización de la GPU sin provocar errores por falta de memoria. Finalmente, el `gradient_accumulation_steps` permite usar un tamaño de lote mayor de manera efectiva, acumulando gradientes durante varios pasos y actualizando los parámetros solo después de haber procesado múltiples lotes.

3.4.3. Tamaño del Conjunto de Datos Utilizado

Para ajustar los hiperparámetros se utilizó todo el conjunto de entrenamiento. De esta manera, los modelos se entran en condiciones similares a las del escenario final, evitando posibles desviaciones por usar muestras reducidas. Sin embargo, pese a que el dataset resulta ser de un tamaño manejable, el costo computacional podría resultar excesivo.

Para solucionar esto, cada combinación de hiperparámetros sugerida por el optimizador entrena al modelo durante 3 épocas completas. Así pues, esto permite observar su

comportamiento con suficiente detalle a la vez que se evita un alto costo computacional.

3.4.4. Criterios de Selección

La selección de la mejor combinación de hiperparámetros se basó en la métrica `eval_loss` o **función de pérdida**, que mide la pérdida del modelo al final del entrenamiento sobre los datos de validación. El fundamento de priorizar esta métrica no es más que su alineación directa con el objetivo de minimizar el error durante el ajuste fino o *fine-tuning*.

3.5. Entrenamiento de los Modelos

A lo largo de esta sección, se detallarán los pasos seguidos para el entrenamiento de los modelos, destacando sus configuraciones, baselines y seguimiento de todo el procedimiento.

3.5.1. Baselines

Como punto de partida para comparar posteriormente con los modelos finales, se usaron las versiones preentrenadas de `meta-llama/Llama-3.2-1B-Instruct` y `tiiuae/falcon-7b` en modo *zero-shot* ([Subsección 2.6.1](#)). Así pues, no se le aplicó ningún tipo de *fine-tuning*, sólo el modelo base ante el dataset de test con una breve descripción de la función de clasificación a realizar.

Los modelos generaron predicciones a partir del prompt, sin haber visto ejemplos del problema. Sin embargo, al tratarse de modelos generativos, las respuestas no siempre se limitaban a la intención esperada. A menudo incluían frases completas o explicaciones adicionales. Por eso se aplicó un proceso de limpieza posterior, cuyo objetivo era extraer solo la palabra que indicaba la intención: *add*, *remove*, *view* o *clear*. Esto permitió comparar las salidas del modelo con las etiquetas reales de forma coherente y medir la capacidad general de los modelos para interpretar instrucciones y clasificar intenciones.

En este escenario inicial, **Llama** logró alcanzar una precisión del **75.9 %**, mientras que el modelo **Falcon** obtuvo un **69.7 %**. Estos resultados sirvieron como referencia para evaluar el impacto de las mejoras introducidas tras el ajuste de hiperparámetros y *fine-tuning*.

3.5.2. Configuración de Entrenamiento

Tras el proceso de optimización realizado con `Optuna`, se identificaron las combinaciones de hiperparámetros que ofrecían el mejor rendimiento sobre el conjunto de validación para cada uno de los modelos ajustados.

En ambos modelos, se llevaron a cabo 18 combinaciones de hiperparámetros diferentes ([Figura 3.5](#)) y se escogieron finalmente aquellas que minimizan la función objetivo (función de pérdida).



Figura 3.5: Resultados de Optuna durante la búsqueda de hiperparámetros

En el caso de Llama, la mejor combinación obtenida fue la del trial 11, mientras que para Falcon se eligió el trial 16.

| Hiperparámetro | LLaMA | Falcon |
|-----------------------------|-----------|-----------|
| Learning rate | 4.902e-06 | 3.573e-06 |
| Max grad norm | 2.828 | 3.25076 |
| Weight decay | 0.1108 | 0.1206 |
| Warmup ratio | 0.1155 | 0.1276 |
| Batch size | 7 | 4 |
| Gradient accumulation steps | 1 | 1 |

Tabla 3.2: Mejores combinaciones de hiperparámetros obtenidas para cada modelo

El entrenamiento de los modelos se llevó a cabo utilizando la clase `SFTTrainer`, una

extensión de `Trainer` de la librería `transformers` de Hugging Face, la cual está pensada para ajustar modelos generativos mediante fine-tuning. Los parámetros y reglas del entrenamiento se definieron con el objeto `TrainingArguments`. Este objeto recoge los hiperparámetros clave y controla aspectos como la evaluación, el guardado de modelos y el registro del proceso.

Ambos modelos fueron entrenados en bloques de 3 épocas y con la configuración de hiperparámetros visible en la [Tabla 3.2](#). El optimizador elegido fue `paged_adamw_32bit`. Además, se estableció un número límite de 60 épocas como medida de control para asegurar un entrenamiento completo sin exceder los recursos disponibles.

Se activó el uso de precisión mixta con `fp16`. Las secuencias se limitaron a una longitud máxima de 256 tokens. También se habilitó `group_by_length` para agrupar ejemplos de longitud similar y así mejorar la eficiencia del entrenamiento.

Además de esto, durante todo el entrenamiento se estableció un almacenamiento automático del modelo cada dos bloques completados, pudiendo así retomar el proceso en caso de interrupción. Una vez finalizado, se almacena el modelo con menor `eval loss` (`load best model at end`). Se aplica un `EarlyStoppingCallBack` ([Subsección 2.2.5](#)) con una paciencia de 3 épocas.

Tanto estos almacenamientos periódicos como el final serán utilizando el formato LoRA ([Subsección 2.11.5](#)), con el fin de evitar una saturación de almacenamiento.

3.5.3. Monitorización del Entrenamiento

Para el seguimiento del entrenamiento, se empleó la integración de `TrainingArguments` con la herramienta `TensorBoard`. De esta manera, se consiguió monitorizar en tiempo real la evolución de la pérdida tanto en el conjunto de entrenamiento como en el de validación.

Como se aprecia en la [Figura 3.6](#), el **entrenamiento de Llama** comenzó con mejoras significativas de la función de pérdida. A medida que las épocas fueron sucediendo, estas mejoras fueron reduciéndose progresivamente hasta estabilizarse en valores muy similares entre épocas. Finalmente, se detuvo el entrenamiento al llegar al límite establecido de **60 épocas** (20 bloques) con una pérdida de validación inferior a **0.23**, lo que indica una buena convergencia del modelo.

De igual manera, el **entrenamiento de Falcon** visible en la [Figura 3.7](#) logró mejoras notables al principio. Estas fueron disminuyendo con el paso de las épocas hasta que no se produjeron mayores mejoras en la función de pérdida del conjunto de validación. Así pues, al no haber mejoras durante 3 épocas consecutivas se activó el `EarlyStopping` y se detuvo el entrenamiento, evitando un posible sobreajuste ([Subsección 2.2.4](#)). El resultado final fue el modelo entrenado durante **36 épocas** (12 bloques) dando lugar a una pérdida de validación inferior a **0.20**.

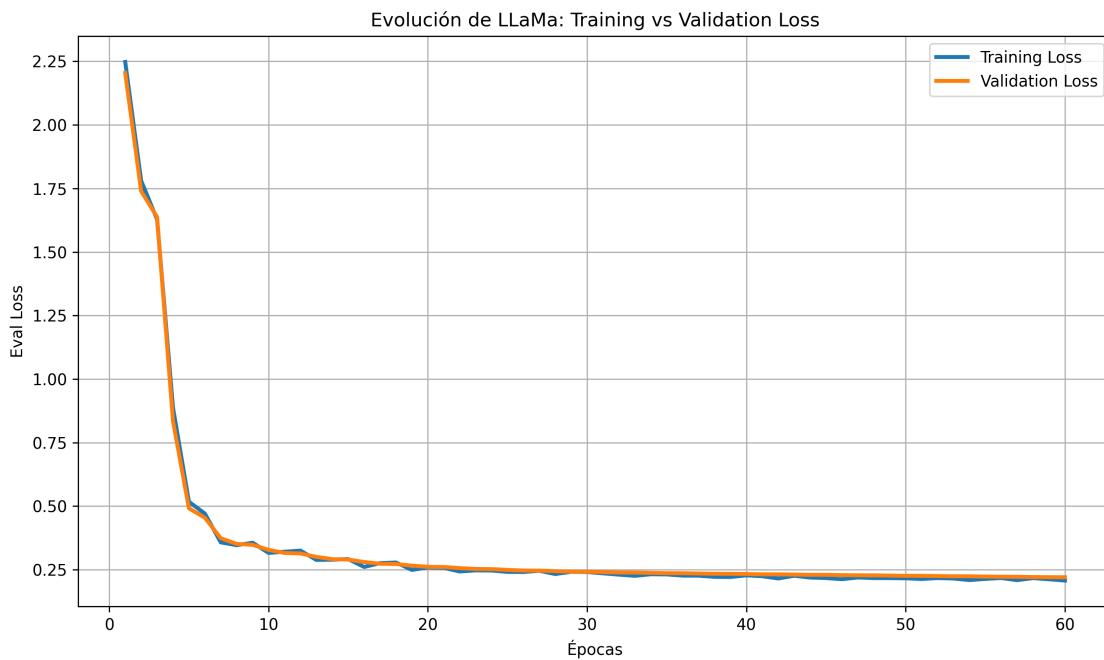


Figura 3.6: Evolución del entrenamiento de Llama monitorizada con TensorBoard.

Ambas gráficas fueron generadas mediante la herramienta `matplotlib`⁷, una biblioteca de visualización en Python.

3.6. Evaluación y Análisis de Resultados

En esta sección, se exponen los resultados obtenidos por los modelos entrenados tras su evaluación sobre el conjunto de test. Para ello, se han empleado métricas estándar como la exactitud global o accuracy, la precisión, el recall y el F1-score por cada clase, además de la media macro (Sección 2.9).

En la Tabla 3.3 se presentan los resultados macro promedios obtenidos por los modelos Llama y Falcon antes (baselines, Subsección 3.5.1) y después del entrenamiento. Estos muestran una mejora significativa. En cuanto a la **precision** y **recall**, los modelos entrenados alcanzan valores superiores, con el Falcon-7B entrenado logrando la mayor mejora (de 0.80 a 0.89 en precision y de 0.70 a 0.86 en recall). El modelo LLaMA 3.2-1B también muestra una mejora, aunque más moderada, con un incremento de 0.80 a 0.87 en precision y de 0.76 a 0.86 en recall.

En términos de **f1-score**, tanto el Falcon-7B entrenado como el LLaMA 3.2-1B entrenado alcanzan un valor del 86 %, lo que refleja un desempeño sobresaliente en la tarea de clasificación de intenciones. Comparado con los modelos baseline, donde el Falcon-7B

⁷<https://matplotlib.org/>

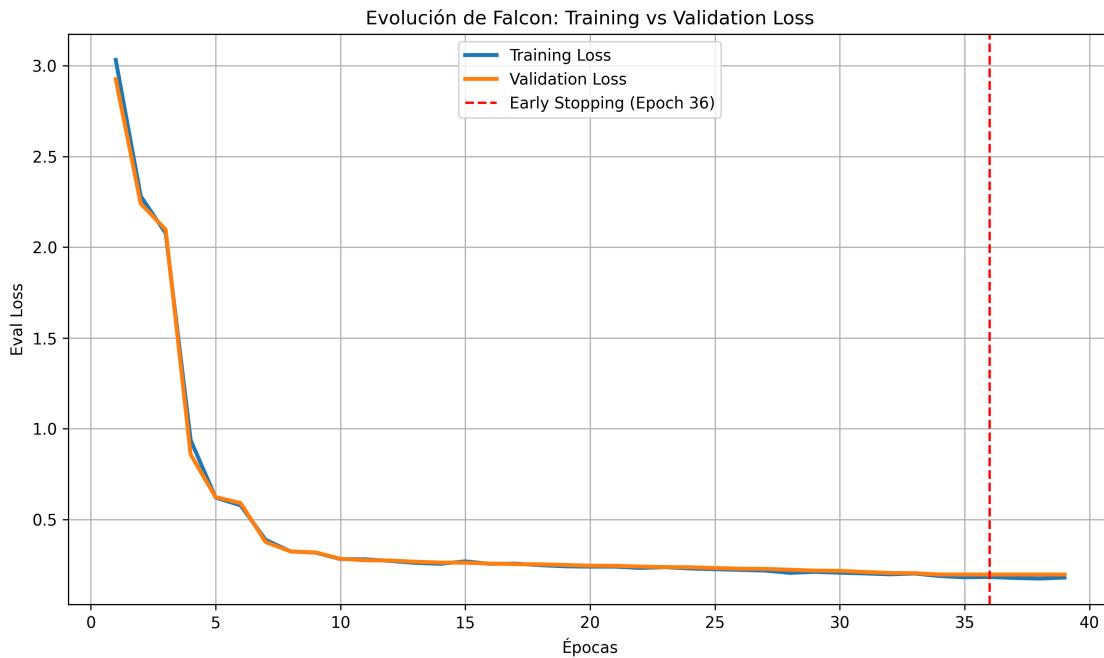


Figura 3.7: Evolución del entrenamiento de Falcon monitorizada con TensorBoard.

obtuvo un accuracy de 0.697 y el LLaMA 3.2-1B de 0.759, las mejoras son notables, lo que demuestra la efectividad del fine-tuning y el ajuste de hiperparámetros.

| Modelo | Precision | Recall | F1-score | Accuracy |
|------------------------|-------------|-------------|-------------|--------------|
| Falcon-7B Baseline | 0.80 | 0.70 | 0.67 | 0.697 |
| Falcon-7B Entrenado | 0.89 | 0.86 | 0.86 | 0.863 |
| LLaMA 3.2-1B Baseline | 0.80 | 0.76 | 0.76 | 0.759 |
| LLaMA 3.2-1B Entrenado | 0.87 | 0.86 | 0.86 | 0.863 |

Tabla 3.3: Resultados de evaluación sobre el conjunto de test.

Más concretamente, en la [Tabla 3.4](#), se muestran los resultados obtenidos para cada clase específica (add, remove, view, clear) de los modelos en el conjunto de test. Para el modelo Falcon-7B, se observan excelentes resultados en las clases *add* y *view*, con un recall impecable en ambas, aunque con un valor más bajo en la clase *remove* (0.74) y en *clear* (0.62). Aunque el modelo tiene un buen desempeño en tareas generales, sigue teniendo ciertas dificultades para reconocer con alta precisión estas dos clases.

Por otro lado, LLaMA 3.2-1B muestra un rendimiento más equilibrado entre las clases, con un alto recall en todas ellas, especialmente en *view* (100 %). La precisión también es alta para todas las clases, sobre todo en *add* (1.00). En general, este modelo tiene un rendimiento robusto, pero su F1-score solo consigue superar a Falcon en el reconocimiento de la clase *clear* (0.84 frente a 0.77).

Esto sugiere que Falcon tiene una mayor capacidad para generalizar, mientras que

LLaMA, aunque con un buen desempeño, necesita mayor refinamiento en ciertas clases.

| Modelo | Clase | Precision | Recall | F1-score |
|--------------|--------|-----------|--------|-------------|
| Falcon-7B | add | 0.83 | 1.00 | 0.91 |
| | remove | 0.74 | 1.00 | 0.85 |
| | view | 1.00 | 0.82 | 0.90 |
| | clear | 1.00 | 0.62 | 0.77 |
| LLaMA 3.2-1B | add | 1.00 | 0.80 | 0.89 |
| | remove | 0.84 | 0.80 | 0.82 |
| | view | 0.81 | 1.00 | 0.89 |
| | clear | 0.84 | 0.85 | 0.84 |

Tabla 3.4: Resultados por clase para los modelos evaluados (conjunto de test).

Matrices de Confusión

En la [Figura 3.8](#) se muestra la matriz de confusión generada por el modelo LLaMA 3.2-1B tras su entrenamiento. Se puede observar que el modelo clasifica correctamente las instancias de la intención *view*. Sin embargo, existe cierta confusión al interpretar 15 frases de intención *add* como *view*. Algo parecido ocurre con *remove* y *clear*, donde la confusión ocurre en ambos sentidos. A pesar de esto, el modelo mantiene un rendimiento general equilibrado entre clases.

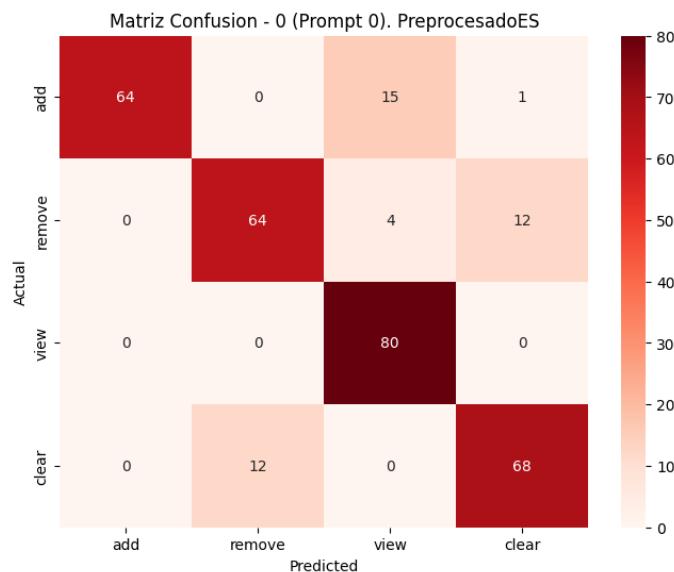


Figura 3.8: Matriz de confusión para el modelo LLaMA.

Por otra parte, la matriz de confusión de Falcon-7B presente en la [Figura 3.9](#) tiene un comportamiento diferente. Se puede apreciar que el modelo clasifica correctamente las clases *add* y *remove*. No obstante, muestra ciertas dificultades para clasificar correctamente

clear y *view*. En particular, se confunden 14 veces instancias de *view* como *add*, y 28 ejemplos de *clear* fueron mal clasificados como *remove*.

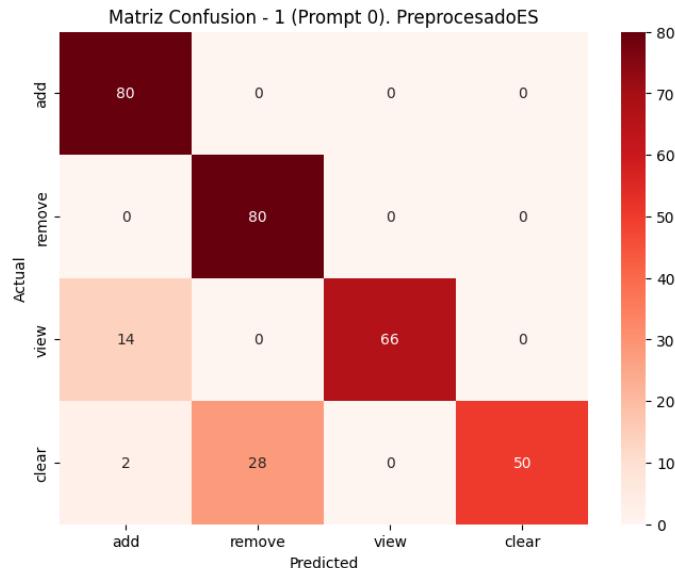


Figura 3.9: Matriz de confusión para el modelo Falcon.

3.7. Análisis de Errores

Tras observar los resultados, se observa que los modelos han cometido ciertos errores. El objetivo de esta sección es identificar los patrones o causas que expliquen sus fallos más frecuentes, con el fin de orientar mejoras futuras.

Errores más frecuentes

Una revisión manual de los errores cometidos por los modelos reveló que tanto el modelo Llama como el Falcon tienden a confundir la intención *clear* con *remove*, y en menor medida *view* con *add*. Esto podría deberse a similitudes semánticas entre las frases o a una formulación ambigua.

Identificación de Errores Comunes

Caso 1:

Texto: “Erase every song I have in the playlist.”

Etiqueta real: clear Predicción: remove

Este mensaje fue clasificado erróneamente como *remove*, entendiendo la intención de eliminar un único elemento de la lista. La ambigüedad en el alcance de la acción (uno vs todos), así como la posible falta de entrenamiento sobre ejemplos específicos, podrían ser las causas.

Caso 2:

Texto: "Show me the songs I added"

Etiqueta real: view **Predicción:** add

El modelo no detectó la intención de ver la playlist y posiblemente se centró en el uso del verbo *add*. Al igual que antes, es probable que se deba a la ausencia de casos más concretos tratados durante el entrenamiento.

Reflexiones y posibles mejoras

Como se ha podido observar, los modelos son sensibles a las estructuras gramaticales atípicas y a los casos ambiguos. Una posible solución sería emplear técnicas de *data augmentation* para introducir una mayor variedad de reformulaciones naturales y ambigüedades, mejorando así la generalización en las predicciones.

3.8. Implementación del ChatBot

Finalmente, tras el tratamiento de los modelos, se procede al diseño e implementación del asistente conversacional que hará uso de ellos. Así pues, a lo largo de esta sección se expondrán los elementos y pasos seguidos en su diseño, desde la creación de la base de datos y framework hasta los procesos de comunicación entre las partes que lo componen.

3.8.1. Base de Datos

Con el objetivo de que el asistente pueda trabajar con información útil, se construyó una base de datos propia. Está compuesta por más de un millón de canciones extraídas de un archivo CSV y se almacena en formato SQLite, permitiendo consultar y modificar los datos desde el programa principal usando instrucciones SQL ([Subsección 2.11.7](#)).

En cuanto a su estructura, la base de datos se compone de dos tablas ([Figura 3.10](#)):

- **Songs:** tabla principal que almacena la información general del conjunto de canciones: título, artista, álbum y año de lanzamiento. Esta tabla solo recibe **operaciones de consulta**.

- **Playlist_songs:** tabla auxiliar que registra las canciones guardadas por el usuario durante la conversación. Sobre esta se permite **añadir y eliminar canciones**, además de **ver y vaciar la lista**.

Los datos se cargaron usando Python y la librería `sqlite3`⁸. Antes de insertarlos, se aplicó una normalización básica del texto: se eliminaron mayúsculas, tildes y caracteres especiales para facilitar las búsquedas.

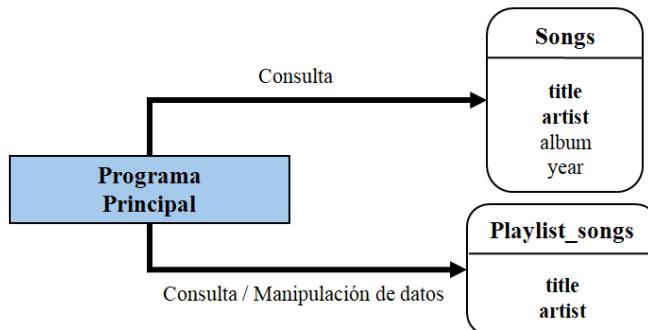


Figura 3.10: Estructura y comunicación con la base de datos.

3.8.2. Creación e interacción en el Servidor Web

El sistema cuenta con una interfaz web accesible desde el navegador. Está diseñada para simular una conversación fluida entre el usuario y el asistente musical. Esta interfaz actúa como cliente dentro de una arquitectura cliente-servidor. Muestra el historial de mensajes e interpreta las respuestas del agente en tiempo real.

Estructura del frontend

Para la interfaz se ha tomado como base el `ChatWidget`⁹ desarrollado por *Krisztian Balog*, profesor de la Universidad de Stavanger. El proyecto se ha desarrollado mediante el uso de React como soporte para TypeScript y sigue una estructura modular organizada en varias carpetas:

- `src/components/` contiene los elementos visuales reutilizables. Incluye los bloques que muestran los mensajes del usuario y del asistente.
- `src/context/` gestiona el estado global del chat. Controla la lista de mensajes, la conexión con el servidor y el estado general del sistema.

⁸<https://docs.python.org/3/library/sqlite3.html>

⁹<https://github.com/iai-group/ChatWidget>

- `src/tests/` agrupa los scripts de prueba para comprobar que los componentes funcionan como se espera.
- `src/App.tsx` es el punto de entrada principal. Coordina el flujo de la conversación y organiza la estructura de la interfaz.
- `src/index.tsx` monta el componente principal en el contenedor HTML definido en la carpeta `public/`.

Dentro de `public/`, el archivo `index.html` actúa como base del frontend. También contiene referencias estáticas, como el favicon, y configuraciones específicas para el despliegue web. El resultado final se muestra en la [Figura 3.11](#).

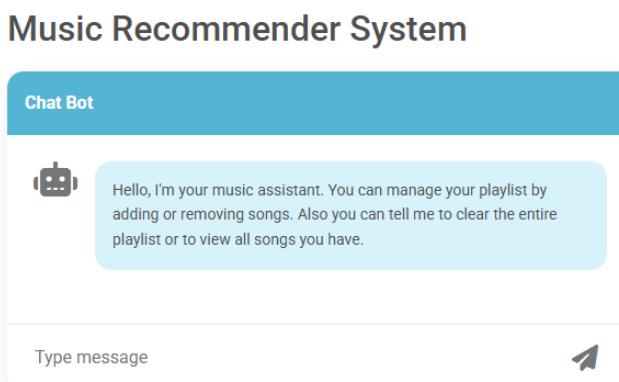


Figura 3.11: Interfaz del ChatBot

Conexión con el servidor mediante DialogueKit

La comunicación entre la interfaz y el servidor se gestiona con **DialogueKit** ([Subsección 2.11.8](#)), usando su clase `FlaskSocketPlatform`¹⁰. Este componente conecta el frontend con el backend mediante WebSockets, dando lugar a un protocolo de comunicación que permite la comunicación bidireccional en tiempo real entre cliente y servidor.

Cuando el usuario envía un mensaje, el flujo es el siguiente:

1. DialogueKit registra el mensaje como una instancia de `Utterance`.
2. El mensaje se pasa al programa principal en el entorno local, que aplica la lógica conversacional correspondiente.
3. La respuesta generada se devuelve al navegador y se muestra automáticamente.

¹⁰https://github.com/iai-group/DialogueKit/blob/main/dialoguekit/platforms/flask_socket_platform.py

3.8.3. Conexión Remota entre el Chatbot y los Modelos

Dado el tamaño de los modelos, las predicciones de los mensajes se ejecutan en remoto, concretamente en Google Colab. De esta manera, se evita sobrecargar la memoria del entorno local y se mantiene la fluidez del sistema.

Para conectar los modelos con el programa principal, se desarrolló una **API** sencilla con Flask, que recibe peticiones POST con el texto del usuario ya convertido en *prompt*. Técnicamente, la API **carga el modelo entrenado**, ya fusionado sin LoRA ni capas PEFT, usando la librería **transformers**. Además, **define la función predict**, que aplica el modelo al prompt y devuelve la intención en formato JSON.

La comunicación con el entorno local es posible gracias al uso de un **túnel local**¹¹, que crea un espacio seguro desde Google Colab hacia una URL pública. Cada vez que se lanza el servidor Flask, se abre también el túnel usando el puerto 5000 o 5001 (según si es LLaMA o Falcon), y se comunica la URL resultante al programa principal.

Así pues, cuando se necesite usar los modelos, el programa principal arma el prompt, lo envía a la URL pública, espera la respuesta y extrae la intención predicha ([Figura 3.12](#)).



Figura 3.12: Comunicación remota entre el entorno local y los modelos.

3.8.4. Arquitectura del Sistema

Cuando el usuario envía una petición a través de la interfaz, el mensaje es recibido por el servidor Flask, el cual lo pasa a DialogueKit para su procesamiento. De esta manera, el mensaje llega al programa principal, donde se ejecutan los siguientes pasos:

1. **Predicción de la intención:** Llegado este punto, el programa realiza una solicitud a la API del modelo entrenado, ya sea LLaMA o Falcon, para predecir la intención del usuario. El modelo responde en base a su función `predict`.
2. **Interacción con la base de datos:** Despues de obtener la intención del usuario, el programa principal actúa, en base a esta, sobre la base de datos para realizar las operaciones necesarias.

¹¹<https://github.com/localtunnel/localtunnel>

3. Respuesta al usuario: Finalmente, el programa envía la respuesta procesada al servidor Flask, el cual la comunica al usuario a través de la interfaz del ChatBot.

Todo el procedimiento queda resumido en el diagrama de la [Figura 3.13](#).

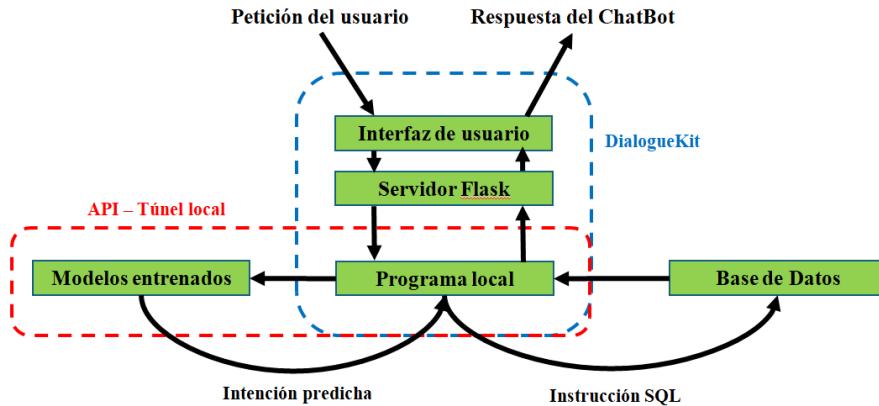


Figura 3.13: Esquema de funcionamiento del ChatBot

3.8.5. Resultado final

El sistema conversacional final integra correctamente el modelo de predicción, una interfaz web y una lógica backend que gestiona operaciones sobre la base de datos musical.

Limitaciones

Sin embargo, el sistema actual utiliza un patrón fijo para extraer el título de la canción y el nombre del artista. Este enfoque impone ciertas restricciones al formato del mensaje.

Para que la extracción de información funcione, el título debe ir entre comillas simples. Además, el nombre del artista debe aparecer justo después de la palabra *by*, que actúa como delimitador.

Por ejemplo, mensajes como *please add Bohemian Rhapsody by Queen* o *delete 'Hysteria' Muse* no se interpretan correctamente. En cambio, la estructura válida sería: *delete 'Hysteria' by Muse*.

3.8.6. Posible mejora

Una mejora posible sería integrar un sistema de reconocimiento de entidades nombradas (NER) entrenado para identificar títulos de canciones y nombres de artistas sin necesidad de una sintaxis estricta.

Ejemplos de prueba

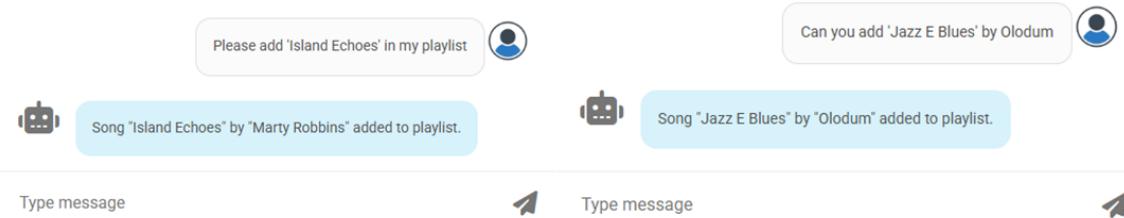


Figura 3.14: Ejemplos para añadir canciones en el ChatBot

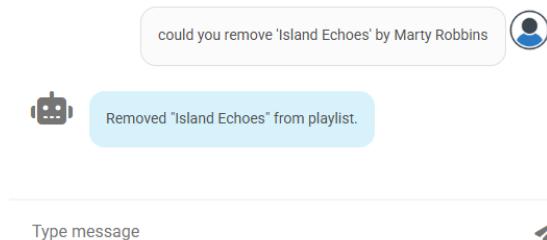


Figura 3.15: Ejemplo para eliminar una canción en el ChatBot

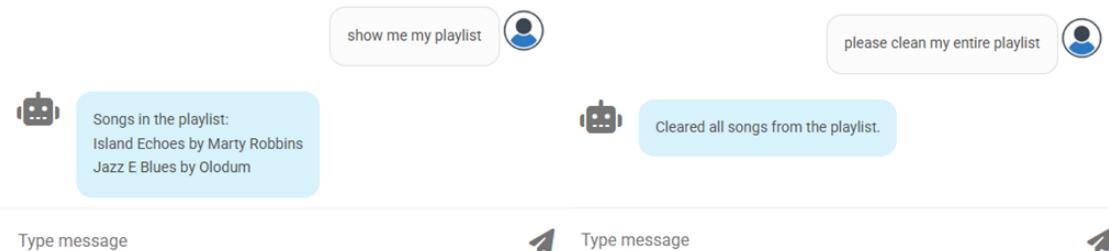


Figura 3.16: Ejemplo para ver/limpiar la lista de canciones en el ChatBot

CAPÍTULO 4

Conclusiones y Trabajo Futuro

Durante este capítulo se ofrecerá una síntesis clara del trabajo realizado. Desde un análisis general de los resultados obtenidos hasta posibles futuras investigaciones a desarrollar. Finalmente, se muestra una tabla informativa del tiempo invertido en las distintas partes del trabajo.

4.1. Conclusiones

Consecución de Objetivos

Los objetivos definidos al inicio del trabajo han sido alcanzados satisfactoriamente:

- Se llevó a cabo una revisión exhaustiva de los enfoques más relevantes dentro del aprendizaje automático y procesamiento de lenguaje natural (PLN) que pueden aplicarse a la comprensión de textos. Esto incluyó modelos preentrenados, con especial énfasis en las arquitecturas de tipo Transformer que se ajustan bien a tareas de clasificación de texto.
- Se trabajó con LLaMA y Falcon, modelos generativos basados en la arquitectura Transformer. Fueron ajustados específicamente mediante el uso de técnicas de fine-tuning para adaptarlos a las tareas del chatbot, lo que permitió personalizarlos para las necesidades de clasificación de intenciones musicales.
- Con el fin de mejorar el rendimiento de los modelos, se utilizó Optuna para el ajuste de los hiperparámetros, lo que contribuyó a obtener un modelo más preciso y eficiente.
- Métricas como precisión, recall, F1-score y la matriz de confusión fueron aplicadas para evaluar la eficiencia de los modelos sobre la clasificación del conjunto de test, permitiendo medir su rendimiento y observar áreas de mejora.
- El chatbot se diseñó utilizando el framework DialogueKit, que permitió gestionar la interacción entre el usuario y el sistema. Se integró una base de datos de canciones mediante SQL y se comunicó el entorno local con los modelos ya entrenados mediante una API.

- Tras haber aprendido, LaTeX fue empleado para estructurar y redactar toda la documentación científica de forma rigurosa. De igual manera, Google Colab fue utilizado como entorno principal para la ejecución de los modelos, lo que permitió trabajar de forma más eficiente, evitando la saturación de recursos en el equipo local.

Resumen de los Hallazgos

Los resultados obtenidos en la fase de evaluación indican que ambos modelos ofrecen un rendimiento sólido tras el entrenamiento. LLaMA logró una mejora del 10% en F1-score respecto al baseline, mientras que Falcon aumentó un 19%. Ambos acabaron con un 86% de acierto sobre el conjunto de test. Además, se ha comprobado que, con un entrenamiento adecuado y el uso de técnicas como LoRA, estos modelos pueden ser finamente ajustados sin necesidad de grandes recursos.

Por otra parte, el análisis de errores ha revelado que los fallos más frecuentes se relacionan con el uso de estructuras gramaticales atípicas (cierta falta de generalización) y casos ambiguos.

Implicaciones

Los resultados muestran que un modelo generativo puede actuar como clasificador dentro de un sistema conversacional. Esto permite pensar en su uso en interfaces más amplias, donde se requiere una comprensión semántica flexible.

También se ha comprobado que es posible desplegar estos modelos en remoto, a través de APIs, y conectarlos con sistemas locales. De este modo, se mantiene la capacidad de procesamiento sin depender del equipo local.

Limitaciones

La extracción de entidades como el título y el artista de la canción presenta ciertas limitaciones. Para que funcione, el mensaje debe seguir una estructura concreta, como el uso de comillas simples para el título y el conector *by* para el artista.

Además, la falta de ejemplos ambiguos o con estructuras gramaticales más variadas dificulta ocasionalmente la detección de la intención.

Contribuciones

Este trabajo aporta un enfoque completo que combina técnicas de PLN, modelos de lenguaje generativos y el desarrollo de un sistema conversacional operativo. Se ha abordado desde el procesamiento de datos hasta la puesta en marcha del ChatBot, que interactúa en tiempo real con el usuario. El resultado muestra cómo el aprendizaje automático puede aplicarse en sistemas conversacionales adaptados al usuario.

Consideraciones Éticas

Es importante considerar las implicaciones éticas del uso de modelos generativos en sistemas conversacionales, sobre todo cuando se utilizan para interpretar instrucciones humanas.

La clasificación de intenciones debe entenderse como un apoyo, no como una respuesta definitiva. Las decisiones basadas en ella deberían acompañarse de revisión humana. Todo el proyecto ha seguido un enfoque académico y experimental.

4.2. Trabajo Futuro

Extensiones del Estudio

Una posible ampliación sería aplicar este enfoque a otros dominios similares, como sistemas de recomendación cinematográficos. Así pues, podría extenderse a la gestión de tareas o agendas. El ChatBot se adaptaría para que ayude a organizar citas, recordatorios o listas de la compra mediante lenguaje natural.

Investigaciones Adicionales

Se podrían explorar variantes del sistema basadas en modelos multilingües o entrenados con datos más cercanos al lenguaje conversacional. Esto permitiría mejorar la comprensión de frases informales, abreviadas o con ambigüedad. También sería interesante experimentar con modelos que ajusten su respuesta al estilo de habla del usuario, dando lugar a una interacción más natural y personalizada.

Aplicaciones Prácticas

El sistema puede ampliarse a interfaces de voz o chat dentro de plataformas de streaming. Esto permitiría gestionar listas con peticiones habladas o escritas, sin depender de menús. También podría integrarse en dispositivos inteligentes del hogar, permitiendo al usuario controlar su música sin necesidad de una pantalla.

Mejoras Metodológicas

Una mejora posible sería aplicar técnicas de aprendizaje semi-supervisado o self-training. Esto permitiría ampliar el conjunto de entrenamiento con datos no etiquetados.

También podría añadirse un módulo de reconocimiento de entidades (NER) para extraer automáticamente el título de la canción y el artista. De este modo, el sistema no dependería de estructuras fijas en los mensajes del usuario.

4.3. Planificación Temporal del Trabajo Realizado

En esta sección se expone el tiempo dedicado al estudio y resolución de las diferentes partes de este TFG, detallando las tareas más complejas y las horas globales empleadas en la [Tabla 4.1](#).

Tabla 4.1: Planificación temporal del trabajo realizado.

| Tarea | Horas |
|---|------------|
| Aprendizaje de los conceptos necesarios para abordar el TFG: | 70 |
| - Aprendizaje automático, Transformers, modelos generativos - Clasificación de texto, Fine Tuning, PLN, métricas de evaluación | |
| Aprendizaje de la tecnología necesaria: | 80 |
| - PyTorch, HuggingFace, Google Colab, LoRA, Optuna, Latex - Flask, APIs, DialogueKit, SQLite, JavaScript/React | |
| Desarrollo del sistema de entrenamiento de modelos: | 100 |
| - Tratamiento de datos, optimización de hiperparámetros, fine-tuning - Análisis de resultados, evaluación y gráficas | |
| Desarrollo del chatbot conversacional: | 60 |
| - Implementación del agente, base de datos e integración de los modelos - Comunicaciones via API y despliegue remoto del ChatBot | |
| Redacción y revisión de la memoria | 70 |
| Preparación de figuras y diagramas | 30 |
| Total | 410 |

La tabla resume el número de horas dedicadas a cada una de las tareas principales del

proyecto. La parte más extensa ha sido el desarrollo del sistema de entrenamiento, con 100 horas.

Aproximadamente 70 horas se dedicaron a la formación teórica. El trabajo se centró en el estudio de la teoría y los conceptos clave para abordar la tarea y alcanzar los objetivos. Al tratarse ciertos contenidos de ser poco abordados en el plan de estudios habitual, fue necesario consultar bibliografía especializada y documentación técnica.

El aprendizaje de las herramientas tecnológicas empleadas en el proyecto requirió unas 80 horas. Además, la implementación del chatbot necesario para simular la conversación con el usuario en tiempo real sumó 60 horas adicionales.

Una vez realizada la fase de experimentación, se dedicaron 100 horas a la elaboración de la memoria, 30 de las cuales fueron dirigidas a la preparación de figuras y esquemas.

Referencias

- [1] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [2] L.J. Sandoval Serrano. “Algoritmos de aprendizaje automático para análisis y predicción de datos”. En: *Revista Tecnológica*; no. 11. (2018).
- [3] SAP. ¿Qué es machine learning? Consultado el 10 de abril de 2025. s.f. URL: <https://www.sap.com/spain/products/artificial-intelligence/what-is-machine-learning.html>.
- [4] Yann LeCun, Yoshua Bengio y Geoffrey Hinton. “Deep learning”. En: *Nature* 521.7553 (2015), págs. 436-444.
- [5] Sergios Theodoridis. *Machine learning: a Bayesian and optimization perspective*. Academic press, 2015.
- [6] K. You et al. “How does learning rate decay help modern neural networks?” En: *arXiv preprint arXiv:1908.01878*. (2019).
- [7] Andy Lo. *Weight Decay and Its Peculiar Effects*. Consultado el 24 de abril de 2025. 2021. URL: <https://towardsdatascience.com/weight-decay-and-its-peculiar-effects-66e0aee3e7b8/>.
- [8] P. Goyal et al. “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”. En: *arXiv preprint arXiv:1706.02677* (2017).
- [9] GeeksforGeeks. *What Is Cross-Entropy Loss Function?* Consultado el 25 de abril de 2025. 2024. URL: <https://www.geeksforgeeks.org/what-is-cross-entropy-loss-function/>.
- [10] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (2nd ed.) O'Reilly Media, 2019.
- [11] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: <https://www.deeplearningbook.org>.
- [12] José Manuel Suárez. *Estudio de métodos y técnicas de aprendizaje por transferencia en el contexto de aprendizaje automático*. Universidad Nacional de La Plata, 2021.
- [13] K. Chowdhary y K. R Chowdhary. *Natural language processing. Fundamentals of artificial intelligence*, 603-649. Springer, 2020.
- [14] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. En: *arXiv preprint arXiv:1810.04805* (2018).
- [15] Carlos Santana. *INTRO al Natural Language Processing (NLP) 1 - ¡De PALABRAS a VECTORES!* Consultado el 14 de abril de 2025. 2020. URL: https://youtu.be/Tg1MjMIVArc?si=rn3aqZk_iP83Z3_-.
- [16] Carlos Santana. *INTRO al Natural Language Processing (NLP) 2 - ¿Qué es un EMBEDDING?* Consultado el 14 de abril de 2025. 2020. URL: https://youtu.be/RkYuH_K7Fx4?si=kDCRJ59xuVLFZcy4.

- [17] Seyhmus Yilmaz y Sinan Toklu. “A deep learning analysis on question classification task using Word2vec representations”. En: *Neural Computing and Applications*, 32(7), 2909–2928. (2020).
- [18] Jeffrey Pennington, Richard Socher y Christopher D. Manning. “Glove: Global vectors for word representation.” En: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1532-1543) (2014).
- [19] Wikipedia contributors. “ELMo”. En: *Wikipedia* (2025). Consultado el 15 de abril de 2025. URL: <https://en.wikipedia.org/wiki/ELMo>.
- [20] Robin M. Schmidt. “Recurrent neural networks (rnns): A gentle introduction and overview”. En: *arXiv preprint arXiv:1912.05911* (2019).
- [21] Ashish Vaswani, Noam Shazeer, Niki Parmar et al. “Attention is all you need”. En: *Advances in Neural Information Processing Systems*. Vol. 30. 2017.
- [22] Natallia Tsuranava. *Estudio de la arquitectura neuronal transformer y comparación entre el mecanismo de atención multicabezal frente al multiconsulta*. Universidad de Alicante, 2024.
- [23] Carlos Santana. *¿Qué es un TRANSFORMER? La Red Neuronal que lo cambió TODO!* Consultado el 17 de abril de 2025. 2021. URL: <https://youtu.be/aL-EmKuB078?si=9QzfvMzZ7USoj0wn>.
- [24] F. Barreto et al. *Generative artificial intelligence: Opportunities and challenges of large language models*. In *International conference on intelligent computing and networking* (pp. 545-553). Singapore: Springer Nature, 2023.
- [25] Kenneth Ward Church, Zeyu Chen y Yanjun Ma. “Emerging trends: A gentle introduction to fine-tuning.” En: *Natural Language Engineering*, 27(6), 763-778 (2021).
- [26] G. Penedo et al. “The RefinedWeb dataset for Falcon LLM: outperforming curated corpora with web data, and web data only”. En: *arXiv preprint arXiv:2306.01116* (2023).
- [27] S. Shameli Derakhshan. “A Music Virtual Assistant Based on Machine Learning”. En: *University of Victoria* (2023).
- [28] Chang Ye. “Chatbot de recomendación musical basado en Dialogflow”. En: *Universitat de Barcelona* (2023).
- [29] V. Chumillas Hernando. “Aplicación de chatbot especializado”. En: *Universidad Politécnica de Madrid* (2024).

APÉNDICE A

Anexos

A.1. Código y programa del ChatBot en Google Drive

Se han habilitado dos carpetas de Google Drive con todos los códigos y archivos necesarios para replicar este proyecto.

A.1.1. Tratamiento de los modelos generativos

La carpeta **Colab Notebooks** contiene el código que se ejecuta en Google Colab y está relacionado con el entrenamiento, ajuste y evaluación de los modelos. Presenta las siguientes partes:

- **DataSplit.ipynb**: Este cuaderno contiene el código necesario para dividir el conjunto de datos original en tres subconjuntos: entrenamiento, validación y test.
- **FinetuningFalcon.ipynb**: En este cuaderno se lleva a cabo el proceso de ajuste de hiperparámetros y el entrenamiento del modelo Falcon. Durante el entrenamiento, los modelos se guardan periódicamente, y una vez finalizado el proceso, el modelo se guarda con su configuración LoRA y se registran las medidas de evaluación.
- **FinetuningLlama.ipynb**: Similar al cuaderno anterior, pero para el modelo LLaMA. Aquí también se lleva a cabo el ajuste de hiperparámetros, el entrenamiento y el almacenamiento de los modelos entrenados junto con sus evaluaciones.
- **EvalLossPlot.ipynb**: Este cuaderno genera una gráfica a partir de los datos de `eval_loss` proporcionados durante el entrenamiento mediante TensorBoard. La gráfica muestra la evolución de la pérdida de validación y entrenamiento a lo largo de las épocas.
- **MergeModels.ipynb**: Este cuaderno carga y guarda los modelos entrenados en su forma completa (sin la configuración LoRA de PEFT), lo que permite la utilización de los modelos para la implementación del chatbot.
- **ft_falcon_model/** y **ft_llama_model/**: Estas subcarpetas contienen los resultados de las métricas de evaluación de los modelos ya entrenados, junto con los archivos de los modelos.

- **final_model/**: Esta subcarpeta contiene el modelo final entrenado con la configuración LoRA y el completo (**falocn_merged** y **llama_merged**), la gráfica generada de la evolución de eval_loss, y los archivos apiFalcon.ipynb y apiLlama.ipynb. Estos archivos permiten cargar los modelos en su forma completa (sin LoRA) y subirlos a la API. Además, se crea un túnel local, y la URL generada permite la comunicación con el entorno local.

La carpeta es accesible a través del siguiente enlace:

[https://drive.google.com/drive/folders/1FTygwtiM0_Xo2kDHgAyG-3rMxpAzXWQT?
usp=drive_link](https://drive.google.com/drive/folders/1FTygwtiM0_Xo2kDHgAyG-3rMxpAzXWQT?usp=drive_link)

A.1.2. Implementación del ChatBot

Esta carpeta contiene el código para implementar el chatbot. Bajo la directiva de la biblioteca DialogueKit, el archivo **music_recommender.py** controla todo el proceso. Se encarga de recibir el mensaje, obtener la predicción de las intenciones del usuario al comunicarse con los modelos entrenados y la consiguiente actuación sobre la base de datos musical. La estructura se corresponde con la descrita en la [Subsección 3.8.2](#).

Dicha carpeta puede verse a través del siguiente enlace:

[https://drive.google.com/drive/folders/1Bx20mMFr8YlYJ36seqakmUUcuAq9vtpd?
usp=drive_link](https://drive.google.com/drive/folders/1Bx20mMFr8YlYJ36seqakmUUcuAq9vtpd?usp=drive_link)

A.1.3. Ejecución del sistema

Para inicializar el chatbot y poner en funcionamiento el sistema, se siguen estos pasos:

1. **Sustituir la URL**: En la función **predict_intent** presente en el código del archivo **music_recommender.py**, sustituye la URL establecida por la URL proporcionada tras ejecutar el archivo apiFalcon.ipynb o apiLlama.ipynb desde Google Colab. De esta manera, se creará y conectará el tunel local entre el programa y la API del modelo seleccionado.
2. **Iniciar el servidor web**: Abre una terminal en la carpeta del programa y ejecuta el siguiente comando: **npm start**. Esto iniciará el servidor que sirve de interfaz del chatbot en el navegador, abriendo una nueva pestaña en el navegador.
3. **Ejecutar el programa principal**: En otra terminal dentro de la carpeta principal, ejecuta: **start ./music_recommender.py**. Esto inicializará el chatbot en la interfaz. Ahora se podrá interactuar con el asistente musical, utilizando los modelos entrenados en Google Colab.