TRABAJO FIN DE GRADO

INGENIERÍA INFORMÁTICA

Modelos de NLP en el ámbito médico

Codificación automática de enfermedades y procedimientos

Autor

Víctor Manuel Oliveros Villena

Directores

Pedro Angel Castillo Valdivieso

Carlos Rodríguez Abellán

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, curso 2023 - 2024

Modelos de NLP en el ámbito médico: Codificación automática de enfermedades y procedimientos

Víctor Manuel Oliveros Villena

Palabras clave: NLP, LLM, NER, GLiNER, GPT, Prompt Engineering, Hugging Face, Aprendizaje Profundo.

Resumen

Desde la aparición de la inteligencia articifial, hemos sido testigos del gran avance que ha supuesto en todos los aspectos de nuestra vida y diferentes sectores profesionales. De entre todos los campos que engloba, uno de los más interesantes e importantes es el procesamiento del lenguaje natural (NLP), marcado en los últimos años por los modelos de lenguaje de gran tamaño (LLM). El objetivo final de estos es la compresión y generación de lenguaje de una manera lo más parecida a la humana, empleando algoritmos de aprendizaje profundo entrenados con una gran cantidad de datos de texto, corpus. Uno de los ejemplos más conocidos son la serie de modelos GPT de OpenAI, usados en ChatGPT, o Gemini, de Google.

A nivel profesional, encontramos una enorme variedad de aplicaciones. Desde análisis de sentimientos en redes sociales, servicio al cliente mediante asistentes o chatbots, traducción automática, reconocimiento de voz como Siri o Alexa, análisis de datos, etc. De entre todos ellos, este proyecto se centrará en el ámbito médico en donde estos modelos son cada vez más utilizados.

Por tanto, se propone el empleo y estudio de diversos modelos NLP de cara a la clasificación automática de enfermedades y tratamientos. Para ello, nos enfocaremos en una de las ramas fundamentales del NLP, NER (Named Entity Recognition). Mediante ella, seremos capaces de identificar palabras como entidades y clasificarlas en categorías a partir de diversos casos clínicos, proporcionados dentro de un corpus, de una manera lo más precisa posible. Además, se discutirán los desafíos y beneficios potenciales que suponen. Se explorarán varios modelos de Hugging Face, entre los cuales destacamos GLiNER, uno de los más recientes, junto con zero-shot learning. Finalmente, se realizará un análisis y comparación de los resultados obtenidos mediante este proceso con aquellos generados por modelos de lenguaje generativos y otros modelos NER más tradicionales.

NLP models in the medical field: Automatic coding of diseases and procedures

Víctor Manuel Oliveros Villena

Keywords: NLP, LLM, NER, GLiNER, GPT, Prompt Engineering, Hugging Face, Deep Learning.

Abstract

Since the appearance of artificial intelligence, we have witnessed the great advance it has brought in all aspects of our lives and different professional sectors. Of all the fields it encompasses, one of the most interesting and important is natural language processing (NLP), marked in recent years by large language models (LLM). The ultimate goal of these is the compression and generation of language in a way that is as close to human as possible, using deep learning algorithms trained with a large amount of text data, corpus. One of the best-known examples is the series of GPT models from OpenAI, used in ChatGPT, or Gemini, from Google.

On a professional level, we find a huge variety of applications. From sentiment analysis on social networks, customer service through assistants or chatbots, automatic translation, voice recognition like Siri or Alexa, data analysis, etc. Among all of them, this project will focus on the medical field where these models are increasingly used.

Therefore, the use and study of various NLP models is proposed for the automatic classification of diseases and treatments. To do this, we will focus on one of the fundamental branches of NLP, NER (Named Entity Recognition). Through it we will be able to identify and classify entities into categories from various clinical cases, provided within a corpus, in the most precise way possible. In addition, the challenges and potential benefits they pose will be discussed. Several Hugging Face models will be explored, among which we highlight GLiNER, one of the most recent, along with zero-shot learning. Finally, an analysis and comparison of the results obtained through this process will be carried out with those generated by generative language models and other more traditional NER models.

D. **Pedro Ángel Castillo Valdivieso**, Profesor del Área de Arquitectura y Tecnología de Computadores de la Universidad de Granada.

D. **Carlos Rodríguez Abellán**, investigador en Fujitsu.

Informan:

Que el presente trabajo, titulado ***Modelos de NLP en el ámbito médico, codificación automática de enfermedades y procedimientos***, ha sido realizado bajo su supervisión por **Víctor Manuel Oliveros Villena**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 11 de junio de 2024.

Los directores:

Pedro Ángel Castillo Valdivieso **Carlos Rodríguez Abellán**

Agradecimientos

En primer lugar, a mis tutores Pedro y Carlos, por su apoyo, consejos y predisposición desde el inicio de este proyecto.

También a mi familia y amigos, por haber sido y ser un gran apoyo constante y ayudarme a seguir adelante.

Índice general

1. Introducción **1**
1.1. Motivación 2
1.2. Objetivos 3
1.3. Justificación de la necesidad del sistema 4
1.4. Estado del arte 4
1.4.1. Dominio del problema a resolver 5
1.4.2. Historia del NLP 6
1.4.3. NLP en la actualidad 7
1.4.4. Trabajos relacionados 10
1.5. Estructura de la memoria 11

2. Fundamentos teóricos **13**
2.1. Aprendizaje automático 13
2.1.1. Aprendizaje profundo 16
2.2. Procesamiento del lenguaje natural 18
2.2.1. Reconocimiento de entidades nombradas 18
2.2.2. Transformers 23

3. Tecnologías y herramientas empleadas en el desarrollo del sistema **29**
3.1. Google Colaboratory 29
3.1.1. Python 30
3.2. Hugging Face 31
3.2.1. GLiNER 32
3.2.2. BERT 35
3.3. OpenAI 36
3.3.1. GPT-3.5 Turbo 37
3.4. Codalab 38
3.5. Dataset 38
3.5.1. Descripción 38
3.5.2. Estructura de los archivos 39

4. Especificación de requisitos **43**
4.1. Requisitos funcionales 43
4.2. Requisitos no funcionales 44
4.3. Requisitos de documentación 44

5. Planificación y presupuesto del proyecto **47**
5.1. Planificación temporal y económica 47
5.1.1. Planteamiento del proyecto 47
5.1.2. Reparto de tareas 48
5.1.3. Investigación 48
5.1.4. Búsqueda del software necesario 48
5.1.5. Aprendizaje 48
5.1.6. Implementación 49
5.1.7. Evaluación 49
5.1.8. Documentación 49
5.2. Presupuesto 50
5.3. Impacto ambiental de modelos Transformers 54

6. Planteamiento **57**
6.1. Descripción del problema 57
6.2. Métricas a emplear 58

7. Implementación **59**
7.1. Carga y exploración de datos 59
7.2. Prototipado del sistema 70
7.2.1. Prototipado de GPT-3.5 70
7.2.2. Prototipado de RoBERTa 75
7.2.3. Prototipado de GLiNER 89

8. Experimentación **101**
8.1. Evaluación de GPT-3.5 101
8.2. Evaluación de RoBERTa 107
8.3. Evaluación de GLiNER 115
8.4. Análisis comparativo de los modelos 131

9. Conclusiones **133**
9.1. Conclusiones 133
9.2. Trabajos futuros 134

Bibliografía **139**

Índice de figuras

1.1. Desarrollo moderno del NLP 10

2.1. Ejemplo de funciones calculadas con algoritmos de AA para clasificación, mediante perceptrón, y regresión, mediante regresión lineal. 14
2.2. Similitud entre redes neuronales biológicas y artificiales. Formadas por una estructura de capas: capa de entrada, salida y capas intermedias. 17
2.3. Recall vs Precisión 22
2.4. Arquitectura básica de un Transformer 24
2.5. Árbol de evolución de los Transformers 28

3.1. Arquitectura del modelo GLiNER 33
3.2. GLiNER Multilingüe 33
3.3. GLiNER Zero-shot y GLiNER Supervisado, respectivamente 34

5.1. Diagrama de Gantt de la cronología del proyecto. 50
5.2. Verificación QR de curso: Intro to Generative AI 52
5.3. Verificación QR de curso: Intro to Hugging Face 53
5.4. Verificación QR de curso: AI Transformers 53
5.5. Verificación QR de curso: Finetuning Transformer Models . . 53
5.6. Emisiones CO_2 de modelos de AA (azul) en comparación con contextos cotidianos (morado). 55
5.7. Emisiones CO_2 de LLaMA 2 para distintos tamaños. 55
5.8. Diagrama de cajas de huellas de carbono según la tarea de inferencia de modelos Transformers. 56

7.1. Gráfica de frecuencias 62
7.2. Gráfica de frecuencias para entidades únicas por tipo 64
7.3. Nube de palabras 65
7.4. Gráfica de distribución de número de entidades por caso clínico 66
7.5. Gráfica de distribución de longitudes de entidades 68
7.6. Gráfica de correlación entre tipos de entidades 69

8.1. Matriz de Confusión de GPT-3.5 103
8.2. Gráfica de frecuencias de entidades de GPT-3.5 104
8.3. Gráfica de frecuencias de entidades predichas correctamente de GPT-3.5 105
8.4. Gráfica de distribución de longitudes de GPT-3.5 106
8.5. Curvas de aprendizaje para RoBERTa usando *dev_text* como test. 109
8.6. Curvas de aprendizaje para RoBERTa usando *dev_text* como validación. 109
8.7. Evolución de métricas de RoBERTa usando *dev_text* como test. 110
8.8. Evolución de métricas de RoBERTa usando *dev_text* como valid. 110
8.9. Frecuencias de tipos de entidades detectadas por ambas implementaciones de RoBERTa. 112
8.10. Matriz de confusión de RoBERTa usando *dev_text* como test. 112
8.11. Matriz de confusión de RoBERTa usando *dev_text* como validación. 113
8.12. Gráfica de distribución de longitudes de RoBERTa. 114
8.13. Comparación de rendimiento de distintos modelos BERT. . . 115
8.14. Curvas de aprendizaje para GLiNER Small usando *dev_text* como test. 117
8.15. Curvas de aprendizaje para GLiNER Medium usando *dev_text* como test. 117
8.16. Evolución de métricas del modelo GLiNER Small con *dev_text* para test. 118
8.17. Evolución de métricas del modelo GLiNER Medium con *dev_text* para test. 118
8.18. Frecuencias de tipos de entidades detectadas por ambas versiones de GLiNER. 120
8.19. Matriz de confusión de GLiNER Small 120
8.20. Matriz de confusión de GLiNER Medium usando *dev_text* como test. 121
8.21. Gráfica de distribución de longitudes de GLiNER Medium Dev Test. 122
8.22. Curvas de aprendizaje para GLiNER Medium usando *dev_text* como validación. 124
8.23. Curvas de aprendizaje para GLiNER Medium usando Trainer.py. 125
8.24. Evolución de métricas del modelo GLiNER Medium con *dev_text* para validación. 126
8.25. Evolución de métricas del modelo GLiNER Medium con Trainer.py. 126
8.26. Frecuencias de tipos de entidades detectadas por ambas versiones de GLiNER Medium. 127

8.27. Matriz de confusión de GLiNER Medium usando *dev_text* como validación. 128
8.28. Matriz de confusión de GLiNER Medium usando Trainer.py. 129
8.29. Gráfica de distribución de longitudes de GLiNER Medium con Trainer.py. 129
8.30. Rendimiento de GLiNER, ChatGPT y UniNER sobre un conjunto de 20 datasets para NER. 131

Índice de cuadros

3.1. Estructura de archivos *train_text.tsv*, *dev_text.tsv* y *test_text.tsv* 41
3.2. Estructura de archivos *text_annotation.tsv*, *dev_annotation.tsv* y *test_annotation.tsv* 41

5.1. Coste del desarrollo del proyecto 51

7.1. Frecuencias concretas por tipo de entidad 63
7.2. Frecuencias concretas por entidades únicas por tipo 64

8.1. Métricas de evaluación test de GPT-3.5 103
8.2. Frecuencias concretas de entidades predichas por GPT-3.5 . . 104
8.3. Evolución del aprendizaje del modelo RoBERTa con *dev_text* para test. 108
8.4. Evolución del aprendizaje del modelo RoBERTa con *dev_text* para validación. 108
8.5. Métricas de evaluación test de RoBERTa 111
8.6. Evolución del aprendizaje del modelo GLiNER Small con *dev_text* para test. 116
8.7. Evolución del aprendizaje del modelo GLiNER Medium con *dev_text* para test. 116
8.8. Métricas de evaluación test de GLiNER 119
8.9. Evolución del aprendizaje del modelo GLiNER Medium con *dev_text* para validación. 123
8.10. Evolución del aprendizaje del modelo GLiNER Medium con Trainer.py. 124
8.11. Métricas de evaluación test de ambos versiones de GLiNER Medium 127
8.12. Rendimiento de los diversos modelos GLiNER implementados 130
8.13. Rendimiento de GPT, RoBERTa y GLiNER. 131

Capítulo 1

Introducción

Con el avance de la inteligencia artificial durante los últimos años, su empleo en todo tipo de dispositivos es prácticamente universal. Esto ha generado que esta disciplina haya aumentado en gran medida su influencia, aceptación e impacto en una enorme cantidad de campos [1].

Este concepto abarca una gran cantidad de paradigmas, entre los más destacados encontramos el aprendizaje automático, la visión por computador, robótica, análisis de datos o los sistemas expertos, entre muchos otros. Sin embargo, en este proyecto nos centraremos en el Procesamiento de Lenguaje Natural (NLP) mediante LLMs (Large Language Models), cuyo objetivo se centra en la interacción entre las computadores y el lenguaje humano, realizando tareas como su compresión o generación, siendo capaz de extraer un significado representativo a partir de un texto [2]. Para ello emplearemos arquitecturas de redes neuronales profundas.

La aparición del modelo GPT (Generative Pre-trained Transformer), integrado en aplicaciones populares como ChatGPT, ha supuesto un gran impacto en la utilización de modelos NLP debido a su escalabilidad y gran tamaño en comparación con sus modelos precedentes. Como consecuencia, se considera uno de los mayores descubrimientos recientes de la inteligencia artificial. Así, ha conseguido ser uno de los modelos más reconocidos por el público en general [3].

Mientras que los modelos GPT están enfocados en la generación de texto, también existen aquellos cuya finalidad se centra en otros aspectos como el reconocimiento de entidades nombradas (NER). Más concretamente, NER es el problema de localizar y categorizar palabras relevantes (a las cuales nos referiremos como entidades) dentro de un texto [4]. Exploraremos el funcionamiento de varios de ellos, de entre los cuales destacamos GLiNER, uno de los más recientes que destaca por obtener un rendimiento sólido que es capaz de superar tanto a ChatGPT como modelos fine-tuned LLMs

mediante el empleo de zero-shot evaluation [5].

De este modo, realizaremos un estudio y pondremos en práctica diversos modelos sobre un corpus de casos clínicos españoles, de modo que evaluaremos y compararemos sus desempeños en función de la capacidad que posean a la hora de proporcionar diagnósticos y tratamientos. Asimismo, se tocarán diversos aspectos como sus beneficios y desafíos de cara al futuro y el impacto que tienen en la actualidad.

1.1. Motivación

Principalmente este proyecto aborda una de las muchas aplicaciones que tiene la inteligencia artificial en la vida diaria. En concreto, en el ámbito médico, sector el cual puede ser considerado realmente relevante debido a la importancia de la salud y el bienestar de las personas.

De todas las aplicaciones que podría proporcionar la inteligencia artificial en la medicina, se ha escogido la disciplina correspondiente al procesamiento del lenguaje natural debido a su gran avance en los últimos años con la aparición de las arquitecturas Transformers en 2017 [6], las cuales sirvieron como base de muchos modelos de aprendizaje profundo (LLM), como el ya mencionado previamente, GPT (Generative Pre-trained Transformer) por OpenAI, y al poco tiempo BERT (Bidirectional Encoder Representations from Transformers) de Google AI Language.

En los últimos años han cobrado especial relevancia debido a su fácil accesibilidad y empleo. Las tecnologías NLP son capaces de trabajar con textos y realizar diversas funciones como el reconocimiento de entidades, generación de textos, clasificación... En nuestro caso planteamos emplear varios de estos aspectos sobre diversos casos clínicos de cara a facilitar, o incluso conseguir, la obtención de diagnósticos y tratamientos. Los beneficios que supondría son los siguientes:

- **Mejorar la precisión de diagnósticos:** Un buen modelo, entrenado con un corpus adecuado (amplitud, variedad de textos, representativo...), puede facilitar enormemente la obtención diagnósticos y tratamientos de manera más precisa y rápida.
- **Optimización de la gestión de registros médicos:** La automatización de la codificación de enfermedades proporcionaría una agilización a la hora de gestionar registros médicos.
- **Reducción de errores humanos:** Suponiendo que los casos clínicos de cada paciente sean correctos, estos modelos podrían reducir la cantidad de errores humanos provocados por diversas situaciones como la

entrada manual de datos o el cansancio. Proporcionando más calidad y fiabilidad.

- **Aumento de la eficiencia en el sector:** Si se consigue una manera de obtener diagnósticos más precisos y a más velocidad, evitando aspectos como la codificación manual por parte del personal médico, se aumentaría la productividad en entornos clínicos.
- **Adaptación a los avances tecnológicos:** El mundo cada vez se encuentra más digitalizado, por lo que la aplicación de estos modelos podría ser una excelente manera de mantenerse actualizado y competitivo.

En resumen, la motivación detrás de la aplicación de modelos NLP para la codificación automática de enfermedades radica estudiar como la aplicación de muchos de sus algoritmos, los cuales están en constante evolución, son capaces de mejorar la capacidad del sector médico de proporcionar diagnósticos lo más fiables, precisos y de calidad posibles para los pacientes. Así, se conseguiría cubrir las necesidades de las personas, al mismo tiempo que se proporcionaría más seguridad, comodidad, control y eficiencia dentro de este sector sin olvidar el aprovechamiento de las últimas tecnologías.

1.2. Objetivos

El objetivo principal de este proyecto es estudiar el desempeño de algoritmos recientes para reconocimiento de entidades nombradas (NER) en el campo de la medicina a la hora de facilitar la obtención de diagnósticos y tratamientos. Para ello, se proponen los siguientes subojetivos:

- Realizar un estudio de los modelos NLP, y sobre todo NER, profundizando en su papel en la medicina y cuáles son los modelos más reconocidos.
- Seleccionar y emplear las herramientas existentes en el mercado para desarrollar un estudio de aplicación real de modelos NER sobre un corpus de datos biomédicos reales y revisado previamente por expertos.
- Estudiar, entrenar y evaluar el rendimiento de diversos modelos NER recientes con otros precedentes pero destacados, sobre un conjunto de datos biomédicos y empleando métricas específicas dentro del NLP. Profundizando en cuáles son sus desventajas y ventajas.
- Comparar el desempeño de los modelos NER seleccionados con modelos generativos, evaluando su precisión en la identificación de entidades médicas.

1.3. Justificación de la necesidad del sistema

Este proyecto encuentra su justificación en la necesidad de abordar una serie de desafíos cruciales relacionados con la interpretación y análisis eficientes de importantes volúmenes de datos textuales en el ámbito médico. En un mundo cada vez más impulsado por la información y comunicación, la capacidad de emplear técnicas de inteligencia artificial para procesar y comprender el lenguaje humano de manera precisa y rápida puede mejorar significativamente la eficiencia en la toma de decisiones y facilitar la compresión profunda de datos. Esta eficiencia se traduce en ahorro de tiempo y recursos, permitiendo una utilización más óptima de los recursos disponibles y un aumento en la productividad y la innovación del sector médico.

Todas estas ventajas no solo facilitarían la realización de mejores diagnósticos y tratamientos por parte de los profesionales médicos, sino que también aumentaría el bienestar y la satisfacción de las personas en sus entornos diarios, aspectos claves para promover una vida de calidad. En este sentido, el empleo de estas herramientas permitiría adaptar la atención médica a las necesidades de cada paciente, incrementando su comodidad. Colateralmente, la seguridad y la precisión de los diagnósticos se verían reforzados debido al análisis de grandes volúmenes de datos médicos, de modo que el sistema podría detectar patrones y tendencias que podrían pasar desapercibidas para los humanos, ayudando a prevenir errores diagnósticos y a optimizar los planes de tratamiento. Esta capacidad tendría como consecuencia un aumento de la confianza tanto por parte de los pacientes como los profesionales de la salud.

Además, la aplicación de estos modelos promueve optimizar los recursos y reducir los tiempos de espera de los pacientes, debido a la automatización de tareas administrativas y de análisis de datos.

En resumen, el empleo del NLP en el sector médico es capaz de mejorar la eficiencia, precisión y personalización de la atención médica cuyos beneficiados serían tanto los profesionales del sector como los propios pacientes.

1.4. Estado del arte

El procesamiento del lenguaje natural (NLP) ha ganado mucha atención recientemente para la representación y análisis del lenguaje humano en el ámbito computacional. Sus aplicaciones son muy diversas, destacando campos como la traducción, detección de spam, extracción de información, resumen, preguntas y respuestas y un largo etcétera [7].

Si lo enfocamos en el sector que nos incumbe, el médico, podemos afirmar que es crucial para la mejora del bienestar de las personas debido a que es

un medio muy adecuado a la hora de transformar información relevante localizada en un texto en datos estructurados que puedan ser usados por procesos computacionales con el objetivo de mejorar la salud de los pacientes y avanzar en esta disciplina [8].

1.4.1. Dominio del problema a resolver

Durante los últimos años la cantidad de información relativa al campo de la salud y bienestar, incluyendo publicaciones, registros electrónicos e información web ha aumentado enormemente. Sobre todo en la lengua inglesa. Aprovechar esa información, la cual se encuentra en su mayoría en formato textual, es un aspecto crítico en todos los campos de la salud. De este modo, se puede lograr impulsar la innovación en la investigación destinada a mejorar esta disciplina, así como mejorar la calidad y reducir los costos. El empleo del procesamiento del lenguaje natural es esencial ya que permite y facilita la obtención de datos estructurados a partir de textos [8].

Una de las áreas clave donde el empleo de modelos NLP ha tenido un impacto es en el análisis de textos clínicos para la extracción de información médica. Estos modelos pueden identificar y clasificar información que pueda ser empleada para la generación de diagnósticos, tratamientos, resultados de pruebas y otros eventos clínicos. Con ello, se consigue un rápido acceso a información relevante en grandes conjuntos de datos.

Además, estos modelos también han sido usados para mejorar la interoperabilidad de los registros médicos electrónicos (EMR) al extraer y estructurar automáticamente información clínica de documentos no estandarizados e incluso de imágenes [9]. Facilitando la integración de datos entre sistemas de registros médicos y mejorando la eficiencia en la recuperación de información.

En cuanto a campos prometedores encontramos el análisis de sentimientos, en donde estos modelos podrían ser capaces de analizar el tono emocional y la actitud de los pacientes en sus interacciones con el sistema de atención médica. Esto puede ser útil de cara a identificar pacientes en riesgo de abandono del tratamiento o para detectar signos de depresión u otras condicionales de salud mental, entre otros muchos usos. Además, facilitaría la toma de decisiones a la hora de realizar diagnósticos [10]. Del mismo modo, en los últimos años también han comenzado a surgir modelos NLP empleados para la minería de texto biomédico (BioNLP) con el objetivo de extraer relaciones entre entidades como genes, proteínas, enfermedades y tratamientos de la literatura científica. Dando como resultado una aceleración de la investigación biomédica lo cual podría llevar al descubrimiento de nuevos tratamientos, biomarcadores y mecanismos de enfermedades [11]. Otro aspecto que también está presentando avances es la personalización de

la atención médica, siendo capaces de analizar el lenguaje y las preferencias de los pacientes para adaptar las intervenciones clínicas y recomendaciones de tratamientos a las necesidades individuales de cada uno [12].

Por último, merece la pena destacar BioMistral (2024), un modelo de lenguaje a gran escala (LLM) de código abierto diseñado específicamente para el dominio biomédico. Este modelo ha sido pre-entrenado a partir de un repositorio con gran cantidad de artículos científicos médicos, PubMed Central. Así, este modelo ha sido capaz de obtener resultados que muestran un rendimiento superior en comparación con otros modelos médicos, tanto de código abierto como propietarios. Además, se realiza una evaluación multilingüe de dicho rendimiento en otras 7 lenguas, lo que constituye la primera evaluación a gran escala de modelos de lenguaje médico en varios idiomas [13].

Así, la inteligencia artificial y en concreto la NLP en el ámbito médico está en constante evolución, con una gran variedad de aplicaciones que cada vez es mayor. Conforme más refinamiento y desarrollo experimenten estos modelos, mayor impacto tanto a nivel práctico como en investigación habrá. Sin embargo, estos avances deben abordar los desafíos éticos y de privacidad asociados a la privacidad de los datos de los pacientes, la equidad del acceso y la interpretación de los resultados, y la transparencia en el desarrollo e implementación de estos modelos [14].

1.4.2. Historia del NLP

Las primeras apariciones de la NLP [7] surgieron a finales de 1940, su finalidad principal era la traducción mediante computador (MT, Machine Translation). Sin embargo, el término NLP como tal aún no se había establecido. El ruso y el inglés eran los lenguajes dominantes para MT [15]. Sin embargo, más adelante en 1966 dicha investigación casi desaparece, de acuerdo al ALPAC (Automatic Language Processing Advisory Committee), se concluyó que la MT no tenía un gran futuro. Aún así, la producción de estos sistemas continuó y tiempo más adelante algunos de ellos fueron capaces de proveer respuestas aceptables a sus clientes [16]. Durante esos años el uso de computadores para estudios lingüísticos y literarios también comenzó. Así, a comienzos de 1960, y debido a la influencia de la IA (campo cuyo surgimiento era muy reciente), se produjeron importantes trabajos, en este caso relacionados con la respuesta a preguntas, BASEBALL QA [17]. Al poco tiempo surgieron sucesores de estos sistemas, más sofisticados y con mayor capacidad lingüística y de procesamiento de tareas, como LUNAR [18] o Winograd SHRDLU. En este punto, a principios de 1980, la teoría de la gramática computacional se volvió un área de investigación muy activa, la cual relacionaba la lógica con la capacidad del sistema de entender

el significado y conocimiento proporcionado por el usuario de cara a deducir sus intenciones.

Al final de la década aparecieron potentes procesadores de oraciones de propósito general, como SRI's Core Language Engine [19]. Al mismo tiempo, la Teoría de la Representación del Discurso [20] ofreció un medio para abordar un discurso más extendido dentro del marco gramático-lógico. Debido a este gran crecimiento, aumentó mucho la cantidad de recursos prácticos, gramáticas, herramientas y analizadores.

Durante los siguientes años, hasta los 2000, hubo una gran cantidad de trabajos a destacar. Cohen et al. (2002) [21] hizo una primera aproximación a una teoría compositiva para la interpretación de melodías basada en supuestos fonológicos. Asimismo, McKeown (1985) [22] demostró que los esquemas retóricos (párrafos que nos sirven para aprender a escribir) podrían usarse para producir textos lingüísticamente coherentes y eficaces. Además, la investigación en NLP comenzó a discutir nuevos temas como redes probabilísticas o la desambiguación de las palabras. Durante la década de los 90, se hizo foco sobre el procesamiento estadístico del lenguaje [23] y sobre la extracción de información y resumen automático [24].

1.4.3. NLP en la actualidad

Los objetivos principales del NLP incluyen la interpretación, análisis y manipulación del lenguaje natural de los datos para un propósito específico mediante el uso de varios algoritmos, herramientos y métodos. Sin embargo, hay una gran cantidad de desafíos involucrados que dependen de dichos datos, y los que en muchas ocasiones dificultan alcanzar todos los objetivos mediante un único enfoque. Por tanto, el desarollo de distintas herramientas y métodos entre el campo de la NLP y otras áreas relevantes han sido motivo de estudio para muchas investigaciones en los últimos años.

A comienzos de los 2000 destacó lo conocido como modelado del lenguaje neuronal en donde en base a las n palabras que precedían a otra, se determinaba una probabilidad para cual sería la siguiente. Con relación a esto surgieron los conceptos de *forward neural network* y *lookup table* [25]. Por otro lado, también se propuso la aplicación del aprendizaje multitarea en el campo del NLP [26], donde dos modelos convolucionales con max pooling fueron usados para reconocimiento y etiquetado de entidades nombradas. A su vez, surgieron otras propuestas como el text embedding [27], donde se añadía un vector numérico en representación del texto. Gracias a esto último se facilitó enormemente la aplicación de redes neuronales en NLP, en donde se empezaron a emplear codificadores y decodificadores de cara al mapeado de una secuencia de texto a vector y viceversa. A partir de 2013, las redes neuronales en NLP comenzaron a tener un rol muy importante debido a la

gran cantidad de estudios realizados hasta ese año. Primero se comenzaron a emplear redes neuronales convolucionales (CNN), que beneficiaron sobre todo al campo de la clasificación de imágenes y análisis visual. Más adelante, estos modelos fueron capaces de abordar más campos como clasificación de oraciones, análisis de sentimientos, clasificación de texto, realización de resúmenes, traducción y muchos más.

Otras redes neuronales que también cobraron bastante protagonismo en el NLP fueron las Redes Neuronales Recurrentes (RNNs), las cuales han sido ideales de cara a trabajar con datos secuenciados como texto, el habla, audio, vídeo y muchos otros [28]. También se implementó una versión modificada de estas redes, conocidas como Long Short-Term Memory (LSTM) que han sido muy usadas en aquellos casos donde se desea extraer información importante y retenerla durante más tiempo, dejando de lado la información irrelevante [29]. Futuros desarrollos de este modelo, más simples, mostraron proporcionar mejores resultados que el modelo estándar en diversas tareas, por ejemplo, Gated Recurrent Unit (GRU) [30]. A continuación, surgieron los primeros mecanismos de atención [31][6] que sugerían una red que aprendiera a que prestar atención en base a una parte oculta de los datos y sus respectivas anotaciones. Este mecanismo es el empleado por los Transformers, los cuales han tenido un desarrollo significante en el campo del NLP desde su aparición en 2017 (*Attention is all you need* [6]) debido a su capacidad de aprender dependencias a largo plazo. Aunque se encuentran limitados por una longitud fija de palabras. Así, se propuso una nueva estructura neuronal, los Transformers-XL [32], que pueden aprender más allá de dicha limitación. Recientemente también se debe destacar la introducción del aprendizaje profundo en este campo [33].

A partir de este punto, se han publicado muchos trabajos sobre NLP, facilitando la creación de nuevas herramientas y sistemas. Así, a día de hoy podemos destacar Analizadores de Sentimientos, Parts of Speech (PoS) Taggers, Chunking, Named Entity Recognition (NER), Etiquetado por Rol Semántico, IA generativa, etc. En el análisis de sentimientos se utilizan recursos lingüísticos como un léxico de sentimientos y una base de datos de patrones de sentimientos para analizar documentos. Encontramos ciertas limitaciones ya que la mayoría de datos etiquetados se encuentran en inglés y están diseñados para lenguas indoeuropeas, por lo que se deja un poco de lado las asiáticas y de Medio Oriente. En cuanto a Parts of Speech, se han hecho varios trabajos en donde se logra una clasificación y etiquetado eficiente en categorías gramaticales de palabras de un texto empleando técnicas basadas en reglas e incluso para lenguas como el árabe [34], sánscrito [35] e indio [36]. Para el árabe también se han empleado y obtenido buenos resultados mediante el empleo de aprendizaje automático supervisado usando algoritmos de tipo Support Vector Machine (SVMs).

Con respecto al Chunking, proceso del cual obtenemos oraciones a partir de texto no estructurado, es frecuentemente usado junto con PoS y evaluado con el dataset CoNLL 2000. Encontramos gran variedad de trabajos que obtienen buenos resultados. [37] [38] [39]

En cuanto a Named Entity Recognition (NER), es el enfoque que vamos a tratar en este proyecto, es una técnica cuya finalidad es reconocer y separar las distintas entidades (palabras) de un texto y categorizarlas en clases. Uno de los más interesantes es Ritter (2011) [40], el cual propuso el empleo de NER sobre tweets, dado que el lenguaje usado en ellos, por lo general, es una jerga del inglés tradicional y estándar. Por tanto, no puede ser procesado por herramientas convencionales. Así, para obtener mejores resultados optó por realizar un pipeline en donde se comenzaba haciendo PoS, luego un chunking para adaptar el formato y finalmente, NER. El rendimiento se vio mejorado en comparación con las herramientas estándar. Por otro lado, también destaca GLiNER, un modelo muy reciente (2023) el cual ha presentado ventajas en comparación con LLMs de generación secuencial de tokens como ChatGPT y otros fine-tuned LLMs , siendo más robusto a nivel de comprensión [5].

En la disciplina del etiquetado de rol semántico (SRL), la cual da un rol semántico a las oraciones, encontramos formalismos como PropBank [41], en donde se asignan roles a palabras que son argumentos de un verbo en la oración. Los argumentos precisos dependen del marco del verbo y si existen múltiples verbos en la sentencia, por lo que podría haber varias etiquetas. En la actualidad, estos sistemas constan de varias etapas: crear árboles de análisis sintáctico, identificar qué nodos del árbol representan los argumentos de un verbo dado y finalmente, clasificar estos nodos para calcular las etiquetas SRL correspondientes.

Por último, la IA generativa, la cual envuelve todos aquellos sistemas capaces de generar contenido nuevo y original, como texto, imágenes o música. Uno de los modelos más populares es Generative Pre-trained Transformer (GPT), desarrollado por OpenAI y lanzado en 2018. A partir de un enfoque de pre-entrenamiento y fine-tuning son capaces de aprender grandes cantidades de texto sin etiquetas. Sus características más relevantes han sido su gran capacidad de generación de texto de calidad, siendo coherente y relevante, su transferencia de conocimiento, su flexibilidad a la hora de realizar tareas (traducción, Q-A, generación de texto...) y sobre todo su escalabilidad, GPT-2 tiene 1.5 billones de parámetros, GPT-3 tiene 175 y GPT-4 no son conocidos en la actualidad [42].

En los siguientes apartados se detallarán aspectos más concretos y enfocados en las disciplinas que se tratarán.

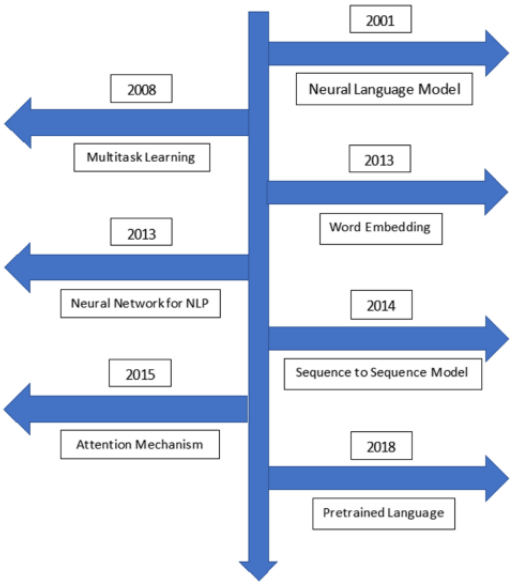

Figura 1.1: Desarrollo moderno del NLP

1.4.4. Trabajos relacionados

Existen varios trabajos y estudios que tratan como modelos NLP pueden ser aplicados a la medicina. A continuación, mencionaremos algunos de ellos y destacaremos sus similitudes:

- **Comparación de implementaciones BERT para NLP en documentos médicos** [43]: En la documentación sanitaria existen una gran cantidad de conceptos complicados, los cuales se ven reflejados en la toma de decisiones, en donde los modelos basados en transformers no han sido investigados tan profundamente. Así, este trabajo estudia el rendimiento de tres modelos BERT sobre este tipo de conceptos médicos, BERT estándar, BioBERT y ClinicalBERT.

- **NER Biomédico usando BERT en el marco de comprensión de lectura automática** [44]: Este trabajo enfoca NER en el ámbito médico (comúnmente llamado BioNER) en vez desde una tarea de etiquetado de secuencias, como un problema de comprensión lectora por computador (MRC). Esta formulación busca introducir más conocimiento previo utilizando consultas bien diseñadas y evitando tediosos procesos de decodificación.

- **Extracción de conceptos clínicos mediante Transformers** [45]: El estudio explora el rendimiento de los modelos basados en transformers en el ámbito clínico. Así, emplean los modelos BERT, RoBERTa,

ALBERT y ELECTRA para compararlos entre ellos.

- **Mejora de la extracción de conceptos clínicos mediante *contextual embedding*** [46]: Se explora como pueden ser usados modelos recientes (ELMo, BERT...) para la extracción de conceptos clínicos y se comparan con los tradicionales métodos de *word embedding* como word2vec, GloVe o fastText.
- **LLMs para extracción de conceptos clínicos y sus relaciones - Model tuning o Prompt tuning** [47]: Este trabajo desarrolla una arquitectura de aprendizaje basada en prompting suave y la compara con cuatro estrategias: fine-tuning sin prompts, hard-prompting con LLMs sin capas congeladas, soft-prompting con LLMs sin capas congeladas y soft-prompting con LLMs con capas congeladas. El LLM empleado es GatorTron, compuesto por 8.9 billones de parámetros y empleado sobre dos benchmarks de datasets relacionados con el ámbito médico.

1.5. Estructura de la memoria

A continuación, se explicará brevemente cómo se estructura esta memoria:

- **Introducción**: Este apartado proporciona una visión general del contenido de la memoria, presentando la motivación y justificación detrás del desarollo del sistema, los objetivos y cual es el contexto actual del tema a tratar, sin dejar de lado cuál ha sido su evolución.
- **Fundamentos teóricos**: Ya que vamos a tratar con conceptos relativamente complejos relacionados con inteligencia artificial, presentaremos las definiciones mínimas y más importantes para su comprensión e indagaremos en el funcionamiento de algoritmos relevantes que se emplearán a lo largo del proyecto.
- **Tecnologías y herramientas empleadas en el desarrollo del proyecto**: Una vez tengamos claros los principios en los que se basa el proyecto, presentaremos que herramientas se han empleado para su desarrollo y el porqué de dicha elección. Incluyendo lenguajes de programación, corpus y frameworks, entre otros.
- **Especificación de requisitos**: Definiremos las necesidades, expectativas y restricciones de nuestro sistema a partir de requisitos funcionales y no funcionales. Es importante destacar que este proyecto está más enfocado en la investigación que en la creación de un software como tal, por lo que se realizará desde ese punto de vista.

- **Planificación y presupuesto del proyecto**: Presentaremos las distintas etapas que se han desarrollado para hacer este proyecto posible, así como hablaremos del presupuesto necesario para implementar las tecnologías mostradas en un contexto real. También se abordarán temas de interés global como la repercusión ambiental que tienen los modelos con los que tratamos.
- **Planteamiento**: En este apartado hablaremos detalladamente del problema que vamos a estudiar, así como qué modelos vamos a emplear y qué sistema de evaluación usar.
- **Implementación**: Esta sección profundiza en cómo se ha desarrollado el sistema. Se tocarán temas como la carga y exploración de los datos, métodos concretos que se han empleado para cada modelo, como se han realizado los entrenamientos o la elección de hiperparámetros en nuestros algoritmos, entre otros aspectos.
- **Experimentación**: Una vez hayamos obtenido unos modelos que se adaptan a nuestros datos, realizaremos una evaluación de su rendimiento, comentaremos sus resultados y hablaremos de ventajas y desventajas que puedan presentar.
- **Conclusiones**: Finalmente, realizaremos una explicación general de todo el proyecto y plantearemos posibles expansiones del mismo.

Capítulo 2

Fundamentos teóricos

Es crucial establecer una base teórica sólida para comprender los principios y métodos que subyacen en el uso y desarrollo de las tecnologías empleadas en este proyecto. Por ello, en este apartado abordaremos cuáles son los conceptos básicos necesarios para facilitar su entendimiento.

Así, partiremos introduciendo a la inteligencia artificial, rama de la informática cuyo propósito es en esencia la búsqueda de la simulación de la inteligencia humana por parte de las máquinas. Dicho de otra manera, es la disciplina que trata de crear sistemas informáticos capaces de aprender y razonar como un ser humano [48]. A su vez, esta está conformada por otras muchas ramas, de entre las cuales nos centraremos en dos, el aprendizaje automático y el procesamiento del lenguaje natural debido a la sinergia que aparece al emplearlas simultáneamente.

2.1. Aprendizaje automático

El aprendizaje automático (AA) es una rama de la inteligencia artificial que busca conseguir que las computadores sean capaces de reconocer patrones en los datos y realizar predicciones basadas en estos. Dicho de otra manera, es la rama que da a las computadores la capacidad de aprender sin ser programadas explícitamente para ello. Debido a la naturaleza, los modelos o técnicas de AA aprenden enormemente de los datos con los que trabajan, y son susceptibles a los cambios que haya en ellos.

Este campo por sí mismo representa un vasto campo de investigación, en el que dependiendo de factores como las necesidades del problema, la naturaleza de los datos a utilizar o el objetivo a alcanzar, podemos encontrar distintos tipos de algoritmos de aprendizaje. En general, podemos dividir los distintos modelos en tres tipos de aprendizaje: aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo.

- **Aprendizaje supervisado**: Este tipo de algoritmos generan un modelo predictivo a partir de unos datos de entrada y sus respectivos datos de salida. Se denomina supervisado porque en el contexto en el que aprenden los algoritmos se tiene de antemano conocimiento total de los datos. Esto quiere decir que para cada entrada disponemos de su respectiva salida. Su funcionamiento se basa en buscar la relación que asocia los datos de entrada con los de salida, aproximando así una función que consiga definir dicha relación. Todo ello midiendo la precisión obtenida a través de una función de pérdida, ajustándola hasta que el error se haya minimizado lo suficiente. Así, disponiendo de una base de datos adecuada, se puede entrenar un modelo de AA. Además, encontramos dos tipos de problemas a la hora de extraer datos: clasificación y regresión.

  La clasificación utiliza un algoritmo para asignar con precisión los datos de prueba en categorías específicas. Reconoce entidades específicas dentro del conjunto de datos e intenta sacar algunas conclusiones sobre como deben etiquetarse o definirse. Entre los algoritmos de clasificación más comunes encontramos clasificadores lineales, máquinas de vectores de soporte (SVM), árboles de decisión, k vecinos más cercanos y Random Forest. Cabe destacar el perceptrón, el algoritmo más simple para clasificación binaria en donde las clases estén claramente diferenciadas.

  La regresión, por otro lado, se usa para entender la relación entre variables dependientes e independientes. Se utiliza habitualmente para hacer proyecciones, como los ingresos por ventas de una empresa determinada. En este caso, destacamos algoritmos como la regresión lineal, regresión logística y la regresión polinómica.

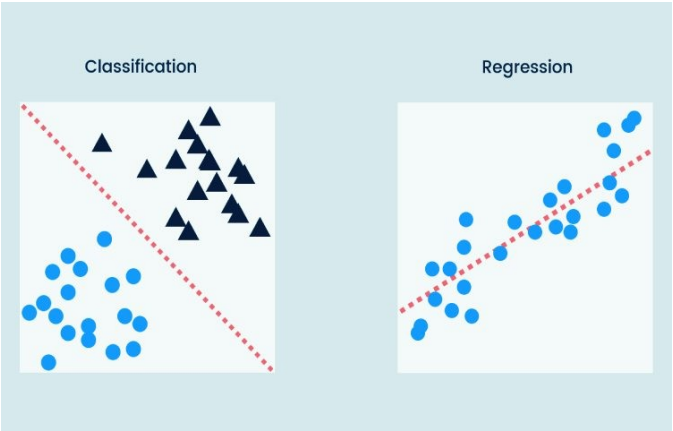

Figura 2.1: Ejemplo de funciones calculadas con algoritmos de AA para clasificación, mediante perceptrón, y regresión, mediante regresión lineal.

- **Aprendizaje no supervisado**: Este paradigma es similar al anterior, con la diferencia de que solamente disponemos de las variables de entrada como información. Por ello, no se disponen de datos con los que comparar sus resultados, y comprobar de esta manera si su estimación es buena o no. Así, son capaces de descubrir patrones ocultos sin necesidad de ninguna intervención humana, convirtiéndolos en una de las mejores soluciones para problemas de análisis de datos exploratorio, estrategias de venta cruzada, segmentación de clientes o reconocimiento de imágenes. Estos algoritmos se distinguen en tres enfoques: agrupación por clústeres, asociación y reducción de dimensionalidad.

  La agrupación en clústeres es una técnica de minería de datos que agrupa datos no etiquetados en función de sus similitudes o diferencias. La asociación es un método basado en reglas para detectar relaciones entre variables en un conjunto de datos determinado. Por último, la reducción de dimensionalidad es una técnica utilizada cuando el número de características, o dimensiones, de un determinado conjunto de datos es demasiado elevado. Así, reduce el número de entradas de datos a un tamaño gestionable, además de preservar la integridad del conjunto de datos lo máximo posible.

- **Aprendizaje por refuerzo**: Este enfoque se centra en algoritmos que aprenden siguiendo una filosofía muy humana, un modelo de acción recompensa. El modelo es sometido a un ambiente en el que irá tomando una serie de decisiones, y para cada una de estas se le asignará una recompensa en función de lo acertada que haya sido. De esta manera, el modelo irá aprendiendo a través de la experiencia y será capaz de reconocer que comportamiento se considera correcto para resolver el problema.

Otra manera de clasificar los algoritmos de aprendizaje automático es a partir de la naturaleza del valor que se busca estimar. De este modo, cuando hablamos de valores continuos, estaremos ante problemas de regresión. En cambio, cuando el objetivo es clasificar los elementos que tengan características comunes, estaremos ante problemas de *clustering* o clasificación, donde se busca crear grupos cuyos miembros sean lo más similares posible entre ellos. Así, el valor a estimar en este caso es discreto y se conoce como etiqueta.

Por otro lado, cabe resaltar que existen un sinfín de algoritmos de AA, muchos de ellos ampliamente reconocidos. Sin embargo, ninguno es mejor que otro sino que cada uno se va a adecuar mejor a un problema concreto dependiendo de la naturaleza de los datos con los que se va a tratar.

En cuanto a los datos empleados en los diversos algoritmos de aprendizaje, lo más probable es que no presenten unas condiciones ideales para su uso directo. Es decir, pueden haber ejemplos con poca exactitud y consistencia, afectando a la calidad de los datos. O podría darse el caso de que no haya cantidad suficiente, poca relevancia o poco representativos entre otros muchos aspectos. Así, es muy importante realizar un preprocesado de datos antes de emplearlos, lo que además de intentar disminuir el impacto de los problemas mencionados, implica técnicas como la normalización, limpieza y transformaciones necesarias para que puedan ser usados por el algoritmo en cuestión. Tras esto, es bastante común dividir el conjunto de datos en tres subconjuntos: entrenamiento, validación y test. Con el objetivo de entrenar el algoritmo, ajustar los hiperparámetros del modelo y realizar una evaluación intermedia durante el entrenamiento, y evaluar el rendimiento final del modelo, respectivamente.

De cara a este proyecto, emplearemos algoritmos de AA basados en aprendizaje supervisado y, concretamente, para clasificación. Más adelante veremos el porqué . De ahí que hayamos profundizado más en este apartado.

2.1.1. Aprendizaje profundo

Al aprendizaje profundo, o *deep learning* (DL), es una de las ramas del aprendizaje automático, y se compone de un subconjunto de algoritmos propios de este área del conocimiento. Estos se componen por una serie de unidades de procesamiento organizadas en capas comunicadas entre sí, formando lo que se conoce como una red neuronal (debido a la similitud con los sistemas neuronales biológicos del cerebro) [Figura 2.2]. Es por este motivo que se consigue la asimilación y entendimiento de los datos por parte del modelo. Para cumplir dicho objetivo, en cada capa se efectúa una transformación del modelo de representación de los datos a uno de más alto nivel. Para ello, este proceso hace uso de un conjunto de funciones no lineales simples que se aplican en cada capa con el objetivo de extraer características de los datos. Una vez extraídas, se consigue una nueva representación de los mismos. Gracias a la composición de varias de estas funciones simples, un algoritmo basado en DL es capaz de aprender funciones mucho más complejas y, por tanto, extraer información de estructuras de datos de grandes magnitudes y/o complejidad.

La denominación de profundo proviene de la cantidad de capas intermedias que pueden existir entre la entrada y la salida del modelo, considerándose más profundo conforme más capas posea. Esta profundidad determina la capacidad de procesamiento del modelo, pudiendo conseguir mejores resultados en problemas de mucha complejidad. Debido a ello, en la actualidad esta área se encuentra en auge ya que los avances tecnológicos han permiti-

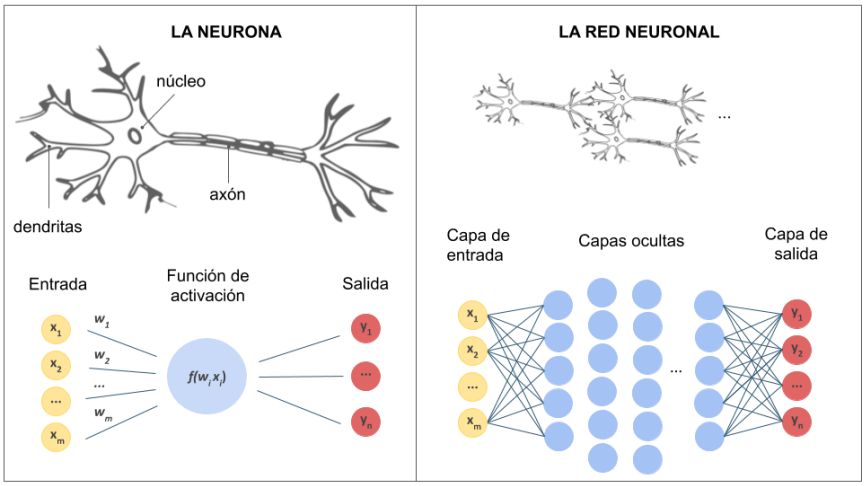

Figura 2.2: Similitud entre redes neuronales biológicas y artificiales. Formadas por una estructura de capas: capa de entrada, salida y capas intermedias.

do crear modelos mucho más costosos computacionalmente, los cuáles han conseguido unos buenos resultados en diversos problemas.

La naturaleza de dichos algoritmos, que procesan los datos y obtienen nuevos modelos de representación para los mismos, los hace muy útiles a la hora de obtener características de los datos con los que se trabaja. Por ello, muchos de los algoritmos de DL han sido ampliamente utilizados para problemas como el reconocimiento de imágenes, reconocimiento del habla, audio, procesamiento de textos... Además, encontramos diversas arquitecturas como las redes neuronales profundas, redes neuronales profundas convolucionales o redes de creencia profunda. De entre ellas, las redes neuronales convolucionales han conseguido muy buenos resultados en la actualidad, sobre todo en problemas de reconocimiento de imágenes y, en menor medida, en extracción de características textuales o clasificación de textos.

Por último, destacar una técnica particularmente relevante en *deep learning*, el fine-tuning. El fine-tuning es una técnica que cobra especial importancia cuando empleamos modelos que requieren un gran poder de cómputo y grandes conjuntos de datos para su preentrenamiento. Consiste en tomar un modelo previamente entrenado y ajustar sus parámetros para una tarea específica junto con un conjunto de datos más pequeño o diferente al que se utilizó durante el entrenamiento inicial. Así, permitiremos al modelo aprovechar el conocimiento adquirido previamente y adaptarlo a una nueva tarea, mejorando su rendimiento con menos datos y tiempo de entrenamiento. Opcionalmente, se pueden añadir nuevas capas o ajustar las existentes para adaptarse mejor a un cometido.

2.2. Procesamiento del lenguaje natural

El procesamiento del lenguaje natural, o *natural language processing* (NLP), es otra de las ramas de la inteligencia artificial que trataremos en profundidad en este proyecto. Previamente ya hemos realizado una amplia introducción a esta disciplina, por lo que estos apartados nos centraremos en aspectos concretos de esta que aplicaremos al proyecto.

El procesamiento del lenguaje natural es un campo que se enfoca en la interacción entre las computadores y los lenguajes humanos. Su importancia a día de hoy es crucial ya que permite entender, interpretar y generar lenguaje humano de una manera que es tanto útil como natual para los usuarios. Como ya vimos, sus aplicaciones son vastas y variadas.

De este modo, cuando trabajemos con modelos NLP, tendremos que buscar un modo de representar el lenguaje humano, como modelos de lenguajes, vectores de palabras (o *word embeddings*), gráficos semánticos, entre otros. Son fundamentales de cara al procesamiento y entendimiento del lenguaje por la máquina. Es por este motivo que enfoques como el análisis morfosintáctico cobra un papel vital, debido a la necesidad de descomponer el texto en sus componentes básicos, como palabras, frases y oraciones; permitiendo la comprensión de su estructura gramatical y las relaciones sintácticas entre ellas. Este punto cobra incluso más relevancia si estamos tratando tareas como el PoS tagging o reconocimiento de entidades nombradas. Si, además, estamos ante modelos cuya finalidad se centra en la generación de lenguaje, aspectos como la semántica, coherencia y relevancia del texto cobran mayor importancia aún. Estos modelos emplean algoritmos que aprenden a predecir la probabilidad de ocurrencia de una secuencia de palabras. Dicho aprendizaje se hace a partir de una gran cantidad de información estructurada, como textos, a los cuales se les suelen conocer como corpus debido a su representatividad, estandarización y referencias académicas.

De cara a la evaluación del rendimiento de los sistemas NLP es realmente importante medir su efectividad. Para ello, son ampliamente usadas métricas como la precisión, F1-score, recall, BLEU (para traducción automática) entre otras. A pesar de todo, aún encontramos limitaciones debido a las complejas estructuras que pueden presentar ciertos lenguajes, la resolución de la ambigüedad semántica y la adaptación a diferentes variedades lingüísticas y estilos de escritura.

2.2.1. Reconocimiento de entidades nombradas

El reconocimiento de entidades nombradas, o *named entity recognition* (NER), es una de las ramas del NLP, la cual podríamos incluso considerar la principal pues su uso dentro de otras ramas es bastante común (traducción,

resumen, QAs...) y mejora considerablemente sus rendimientos. Su tarea principal consiste en, a partir de un problema de extracción de información en el que estén implicados documentos, ya sean estructurados o no, identificar expresiones que se refieran a personas, lugares, organizaciones o compañías. Estos son los casos generales pero ya veremos más adelante que podemos extenderlo a todo tipo de ámbitos.

Funcionamiento y tipos de NER

Por tanto, podemos distinguir dos tareas, la primera consiste en la identificación de palabras en un texto, a las cuales nos referiremos como entidades, y la segunda la clasificación de estas en un conjunto de categorías predefinidas previamente y que sean de nuestro interés. Para los humanos, identificar entidades como nombres propios es bastante sencillo ya que la mayoría de ellos comienzan por una letra capital, en mayúscula. Sin embargo, este trabajo no es tan sencillo para una computadora. Incluso, podríamos llegar a pensar que a la hora de clasificarlas podríamos usar estructuras aparentemente simples como diccionarios, lo cual a pesar de que sería posible, realmente no es eficiente debido al enorme tamaño que deberían tener y el hecho de que tendrían que ir actualizándose con nuevas palabras cada vez que aparezcan en la literatura. Esto no quiere decir que el empleo de diccionarios no sea útil, sino que no debemos delegar en ellos hasta tal nivel. Además, hay aspectos que no se podrían tener en cuenta, como la ambigüedad o el contexto en el que una palabra es usada, que puede determinar el tipo de categorización para dicha entidad.

En base a esto, se han desarrollado distintos métodos, sobre todo en los últimos años. Podemos destacar tres tipos de enfoques para NER: basado en reglas, basado en aprendizaje automático o híbrido. Cuando hablamos de NER basado en reglas, nos referimos a extraer entidades a partir de un conjunto de características ortográficas, sintácticas (por ejemplo, precedencia de palabras) y gramáticas, las cuales generalmente han sido establecidas por un humano de manera que sigan unos patrones en particular. Dentro de este punto una técnica común es la aplicación de expresiones regulares [49] de cara a identificar entidades, de modo que el primer paso consiste en reconocer las oraciones, continuaría identificando los patrones y terminaría con la categorización de estos. Otro con menos popularidad sería el uso de *gazetteers* [50], es decir, emplear un conjunto de diccionarios de un modo similar al que mencionamos en apartados anteriores. Por último, también existen modelos más sofisticados que dependen en su totalidad de reglas previamente establecidas manualmente [51]. El problema de este tipo de modelos es que deben de ser usados para dominios muy concretos, aunque son capaces de obtener muy buenos resultados e incluso, a veces, mejores que modelos que emplean AA. Sin embargo, presentan otras desventajas como la dificultad

en su portabilidad y poca robustez.

En cambio, NER basada en aprendizaje automático se enfocan en tratar el problema de extracción de entidades como un problema de clasificación tratado desde un punto estadístico. De este modo, el sistema busca patrones y relaciones dentro de un texto empleando modelos estadísticos y algoritmos de aprendizaje automático. Siendo capaz de detectar y categorizar las respectivas entidades. Dado que estamos ante un problema de clasificación, podemos emplear dos tipos de aprendizaje, supervisado o no supervisado. Si nos decantamos por el supervisado necesitaremos emplear algoritmos que permitan al programa aprender a clasificar un conjunto de ejemplos etiquetados para un número de características (en nuestro caso las categorías) determinadas. Esto implica que nuestro conjunto de entrenamiento esté previamente etiquetado y sea lo suficientemente grande. En cuanto a modelos estadísticos que suelen usarse, destacamos aquellos basados en el modelo oculto de Markov, modelos basados en entropía máxima o modelos basados en árboles mediante SVM (Support Vector Machine).

Con lo que respecta al aprendizaje no supervisado, como ya vimos, no hay etiquetas por lo que el modelo no recibe ningún feedback. De este modo, el objetivo reside en que el programa sea capaz de construir representaciones a partir de datos. Sin embargo, este enfoque no es tan popular ya que realmente los sistemas que lo emplean no pueden considerarse no supervisados en su totalidad debido a que dependen de cierta información conocida con anterioridad. Así, aunque pueden no requerir de etiquetas explícitas para los datos, sí que es común el empleo de diccionarios con recursos externos para guiar el proceso de identificación de entidades, o el empleo de conocimiento lingüístico previo con información sobre estructuras semánticas y gramaticales, entre otras técnicas. Aún así, los modelos NER basados en aprendizaje automático, a diferencia de los basados en reglas, pueden ser fácilmente aplicados a diferentes dominios e idiomas.

Por último, NER híbrido es una combinación de ambos, creando nuevos métodos empleando los puntos fuertes de cada enfoque. Sin embargo, las desventajas del empleo de reglas siguen siendo vigentes, de modo que este tipo de modelos no pueden extenderse a una gran variedad de dominios. Aunque son capaces de obtener buenos resultados en aquellos en los que sí son soportados.

Métricas y evaluación

A continuación, vamos a introducir cuales son las métricas de evaluación al emplear NER. Como sabemos, NER se encarga de identificar y categorizar entidades; para ello, asigna identificadores únicos dependiendo del tipo de expresión que estemos tratando. Los tipos básicos son NUMEX para

cantidades, TIMEX para fechas u horas y ENAMEX para organizaciones, personas o lugares. Actualmente existen muchas más. Así, primeramente identifica todas las instancias de las expresiones correspondientes para subcategorizarlas a posteriori. El sistema no se basa en un sistema por etapas, *pipeline*, en donde NER se maneja completamente como un preprocesamiento del análisis de oraciones y discursos, va más allá. Así, lo que se busca es producir una única salida clara y no ambigua para cualquier cadena relevante de texto, reconociendo lo que la cadena representa y no solo su apariencia superficial. Para casos como expresiones NUMEX es simple, pero en otros casos la respuesta no puede ser tan evidente y requiere de técnicas que utilicen información de un contexto más amplio, por ejemplo, si la palabra comienza por mayúscula.

Para afrontar estos problemas, existe un modelo de evaluación en donde se mide tanto la precisión (P) como el recall (R) [Figura 2.3] de cara a cuantificar el funcionamiento de nuestro modelo. La precisión es una medida de exactitud de las predicciones positivas de un modelo. Una alta precisión significa que el modelo tiene pocos falsos positivos, es decir, cuando el modelo predice una entidad como positiva, es muy probable que sea correcta.

$$P = \frac{Verdaderos\ positivos}{Verdaderos\ positivos\ +\ Falsos\ positivos}$$

El recall o sensibilidad, por otro lado, es una medida de la capacidad del modelo para identificar todas las instancias positivas. Se define como la proporción de verdaderos positivos sobre el total de instancias reales positivas. Un alto recall implica que el modelo tiene pocos falsos negativos, es decir, es capaz de identificar la mayoría de instancias positivas presentes en los datos.

$$R = \frac{Verdaderos\ positivos}{Verdaderos\ positivos\ +\ Falsos\ negativos}$$

Es importante entender la diferencia entre ambas, pues serán usadas durante el desarrollo de este proyecto. Asimismo, nos servirán para introducir otra nueva métrica que también cobrará bastante relevancia.

Del mismo modo, cuando hablamos de verdaderos positivos (TP), nos referimos a instancias, o ejemplos, positivos que nuestro modelo clasifica como tal; falsos positivos (FP), son instancias negativas que el modelo clasifica como positivas; y falsos negativos (FN) son instancias positivas que el modelo clasifica como negativas.

Con esta información, podemos establecer una nueva métrica a partir de la relación inversa entre precisión y recall, donde mejorar uno puede llevar a una disminución en el otro. Por tanto, para evaluar el rendimiento general

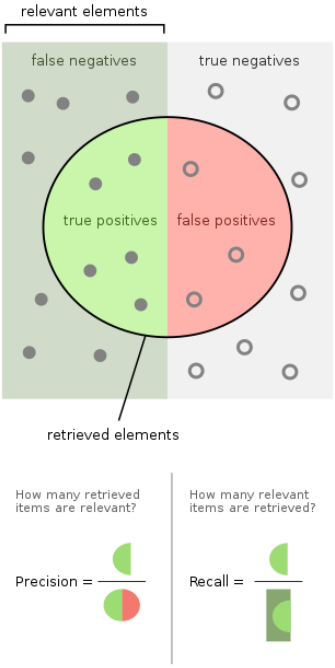

Figura 2.3: Recall vs Precisión

de un modelo, en nuestro caso NER, a menudo se utiliza esta relación, la cual se conoce como F1-score y es la media armónica de precisión y recall.

$$F1 - score = 2 \cdot \frac{Precisión \cdot Recall}{Precisión + Recall}$$

El rango de valores de esta métrica (y de las anteriores) varía entre 0 y 1, donde 1 indica el mejor rendimiento posible (alta precisión y alto recall) y 0 el peor. Así, nos proporciona un equilibrio entre ambas y es más robusto que ellas por sí solas, especialmente en situaciones donde hay un trade-off entre ambas.

Zero-shot learning

Por último, vamos a hablar del aprendizaje zero-shot aplicado a NER. Como tal, es un concepto perteneciente al campo del aprendizaje automático que se ocupa del escenario en el que un modelo se entrena en un conjunto de clases y se prueba en otro conjunto de clases distinto sin recibir supervisión para este segundo. Por tanto, si hacemos una analogía con NER, nos referiremos a la capacidad de un modelo de reconocimiento de entidades nombradas para identificar y clasificar entidades que no han sido vistas durante el entrenamiento del modelo. En otras palabras, un modelo zero-shot

NER puede reconocer y asignar etiquetas a entidades que no están presentes en su conjunto de datos de entrenamiento.

Esto es posible gracias a la capacidad que tenga el modelo para comprender y generalizar patrones lingüísticos y semánticos, lo que le permite identificar nuevas entidades basadas en su contexto y estructura. Estos modelos suelen basarse en representaciones de lenguaje contextual pre-entrenadas, como BERT o GPT (de los cuales hablaremos en apartados posteriores más en profundidad), que han sido entrenadas en grandes corpus de texto y capturan una comprensión profunda del lenguaje natural.

Gracias a esto, algunas aplicaciones incluyen la identificación de entidades específicas de dominio o nuevas entidades emergentes que no estaban presentes en los datos de entrenamiento originales del modelo. Esto proporciona una flexibilidad y capacidad de adaptación significativas en escenarios donde el conjunto de entidades objetivo puede cambiar con el tiempo o donde se necesita identificar entidades en un dominio específico para el cual no se dispone de datos de entrenamiento etiquetados.

2.2.2. Transformers

Para comprender cómo funcionan los Transformers, es importante conocer que son los grandes modelos de lenguaje, o *Large Language Models* (LLMs). Estos son redes neuronales capaces de trabajar con textos (lectura, traducción, resumir...) pudiendo así crear frases y predecir palabras del mismo modo que lo haría un humano. Para ello, previamente han debido ser entrenadas con vastas cantidades de textos para ser capaces de reconocer patrones y aprender.

Arquitectura Transformer

Los Transformers son un tipo de LLM considerados a día de hoy como uno de los modelos más destacados en el *deep learning*. Además, actualmente muchos son agnósticos a los datos, es decir, pueden trabajar con texto, imágenes, vídeo, audio o incluso secuencias de proteínas, por lo que han sido adoptados en una gran cantidad de campos como en visión por computador, procesamiento del habla y, el que nos incumbe en este proyecto, procesamiento de lenguaje natural. Actualmente existen muchas variantes, propuestas debido a su gran éxito en los últimos años, pero la arquitectura Transformer básica [Figura 2.4] es un modelo secuencia a secuencia, *seq2seq*, formada por un codificador y un decodificador, cada uno compuesto por una serie de bloques idénticos. Estos bloques contienen principalmente un módulo de autoatención *multi-head* y una red de alimentación hacia adelante (FFN) aplicada a nivel de posición. El primer módulo mencionado es clave en este

tipo de arquitecturas.

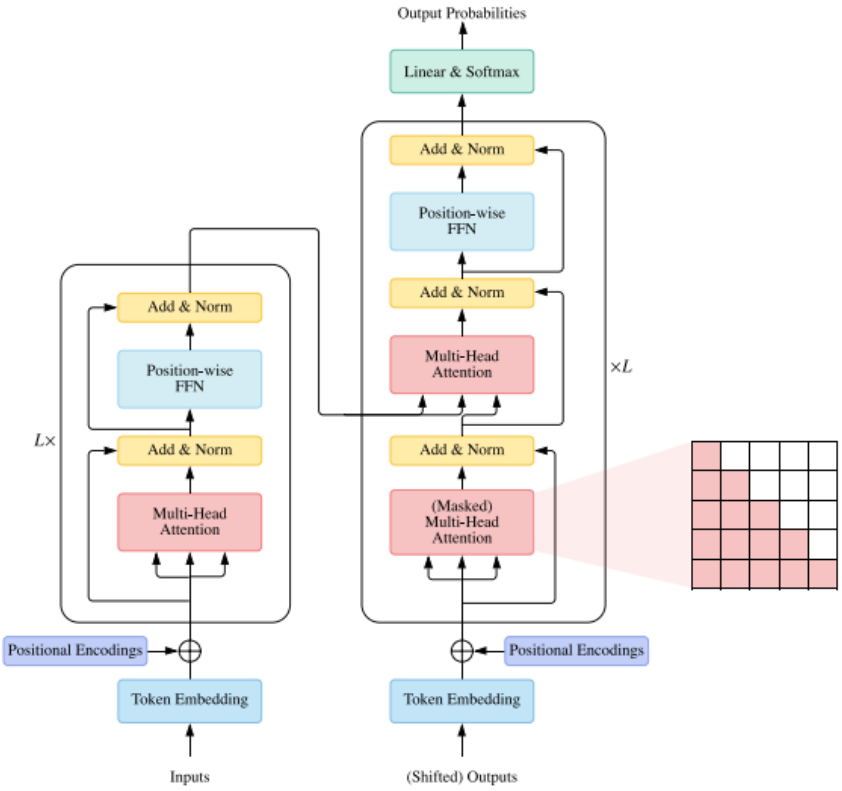

Figura 2.4: Arquitectura básica de un Transformer

A continuación, desglosaremos con más detalle las partes y el funcionamiento de dicha arquitectura. Siguiendo el proceso natural, partimos de una secuencia de entrada, la cual pasa al codificador, *encoder*. Este se encarga de representarla internamente con el objetivo de capturar el contexto y las relaciones entre los elementos de dicha secuencia. Consta de varias capas, embedding, autoatención, *feed-forward* y normalización y conexiones residuales.

La capa de embedding se encarga de convertir cada palabra en uno o varios tokens, es decir, en un valor numérico asociado, ya que las redes neuronales no pueden procesar cadenas de texto. A continuación, el token pasa a ser un vector de características de 512 dimensiones, propuesto en el paper donde fueron presentados los Transformers [6]. Es importante destacar que este vector emplea lo que se conoce como *positional encoding*, es decir, ya que los Transformers no tienen una estructura secuencial, se añade información de posición a los embeddings para que el modelo pueda capturar el orden de los elementos en la secuencia.

El mecanismo o módulo de autoatención *multihead* es lo que distingue los Transformers de arquitecturas precedentes, ya que es capaz de procesar una secuencia de entrada de manera simultánea en lugar de secuencial (palabra a palabra) como otros modelos como las Redes Neuronales Recurrentes (RNN), haciéndolo más rápido y permitiendo una mejor comprensión entre las palabras de una oración debido a que puede capturar relaciones a largo plazo. Esto significa que puede entender cómo una palabra al comienzo de una oración se relaciona con otra al final de esta. Esta capa genera tres matrices a partir de los embeddings, Query (Q), Key (K) y Value (V) donde:

- **Query**: matriz formada por los embeddings de los tokens que vamos a evaluar.
- **Key**: la capa de atención construye un diccionario blando (*soft dictionary*) que contiene la atención entre cada palabra calculada a partir del embedding y la matriz de pesos. Así, Key es una matriz que contiene los tokens nuevamente, pero como claves de dicho diccionario.
- **Value**: matriz que contienen los embeddings de salida de todos los tokens.

En base a esto, la atención se calcula a partir de la siguiente fórmula:

$$Z = Softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Si nos olvidamos por ahora de Softmax y el *multihead*, la atención es una multiplicación matricial Q por K traspuesta, lo cual llamaremos factor y que multiplicaremos por V. Así, ya que tanto K como Q son los valores de los embeddings (tokens n-dimensionales), al hacer el producto vectorial obtenemos matemáticamente la 'similitud' entre los vectores. Cuanto mayor sea el valor, mayor semejanza.

Ahora, debemos destacar que la capa de atención es *multihead*, esto quiere decir que en vez de calcular la atención de todos los tokens de una oración una vez con las 512 dimensiones del embedding, se subdividen los valores en grupos para calcular su atención individualmente. De manera algo arbitraria, el paper original de Vaswani [6] propuso 8 heads, por lo que se calculan 8 atenciones de a 64 valores de embeddings por cada token, obteniendo buenos resultados. La atención final será un promedio de los resultados obtenidos en cada cabeza.

En cuanto a la fórmula para calcular la atención, la función Softmax se emplea para obtener pesos normalizados y luego aplicarlos a los valores estimados, V. Por otro lado, la expresión $\frac{1}{\sqrt{d_k}}$ es un factor de escala para normalizar la estimación donde d_k hace referencia la dimensión de Key.

Por último, destacamos tres tipos de atención:

- **Self attention**: El modelo Query-Key-Value se crea a partir de las propias entradas de los tokens de entrada. Es el más simple y el proceso es el acabo de explicar.
- **Cross attention**: Las entradas del decodificador son los valores obtenidos por el codificador. Esto supone que con solo el valor, V, pueda modelar la salida de atención buscada. Dicho con otras palabras, el decodificador utiliza las claves, K, y valores, V, del encoder para calcular los pesos de atención de la secuencia de entrada y así indicar que partes son más relevantes por parte de esta para cada palabra que el decodificador procesa.
- **Masked attention**: Se enmascara la parte triangular superior de la matriz de atención para no caer en *data leakage*, es decir, para no 'adelantar información futura' que el output no podría tener en su momento, previniendo el sobreajuste. En general, es empleada en tareas de traducción.

Continuando con las demás capas del codificador, encontramos la red neuronal prealimentada, *feed forward*. Esta capa es una red tipo MLP (Perceptrón multicapa) o capa densa, es decir, cada neurona de una capa está conectada a todas las neuronas de otra adyacente. Esto, junto con funciones de activación no lineales, ayuda a que se puedan detectar patrones complejos en el lenguaje. Esta capa también se encarga de la regularización y prevención del sobreajuste mediante técnicas como dropout.

Por último, también destacamos las capas de normalización y de salto residuales. La primera mencionada tiene la función de estabilizar y acelerar el entrenamiento mediante la normalización de datos a lo largo de las dimensiones de la característica en cuestión (no confundir con *batch normalization* que normaliza los datos a lo largo de las dimensiones del lote). En cuanto a la capa de salto residual permite mantener el valor de origen del input a través de la red de aprendizaje profundo evitando que sus pesos se desvanezcan con el paso del tiempo. Es una técnica muy empleada en las famosas ResNets para clasificación de imágenes.

A continuación, pasaremos a hablar del decodificador. Si volvemos a fijarnos en la estructura de un modelo Transformers [Figura 2.4], podemos darnos cuenta que emplea los mismos tipos de bloques que el codificador. Sin embargo, su función es distinta pues se encarga de generar la secuencia de salida basada en la información procesada por el encoder y la secuencia de salida generada hasta el momento. No solo eso, si vemos de nuevo la arquitectura, vemos que también recibe una entrada. En modelos de entrenamiento, generalmente (y sobre todo en aprendizaje supervisado) pasamos

un input y su etiqueta, para indicar como debe ser nuestra salida, al mismo tiempo. Dicha etiqueta suele usarse únicamente para validar el modelo y ajustar los pesos de la red durante el *backpropagation*. Sin embargo, la salida que generaría dicha etiqueta es también tomada como entrada por el decodificador en este tipo de modelos y es procesada como mencionamos previamente: obtenemos embeddings, posicionamiento, capas de autoatención... De este modo, el decodificador también forma parte del aprendizaje del modelo.

En cuanto a la salida, el modelo pasa por una capa lineal (o densa) y aplica otro softmax.

En general, este es el funcionamiento de este tipo de algoritmos. Hay que recordar que la arquitectura completa implica la creación de L codificadores y decodificadores, el paper original proponía el empleo de seis.

Evolución de los Transformers

Desde sus inicios hasta hoy los Transformers se han convertido en un modelo 'de facto' para todo tipo de tareas de NLP, de las cuales vamos a prestar mayor atención a NER. Para ello, emplearemos técnicas diversas técnicas de *deep learning* para poder trabajar con ellos y nuestros datos.

Además, a raíz de poder entrenar mediante GPU, lo que supone reducción de tiempo y dinero, surgieron varios modelos para NLP que han sido entrenados con datasets cada vez más grandes. Se tenía la creencia de que cuantas más palabras, más acertado sería el modelo, lo que llevó a alcanzar un límite y en 2020 pareció haber alcanzado un tope de accuracy. A partir de esa fecha, no se dio tanta importancia a aumentar tanto el vocabulario y se comenzaron a enfocar otros aspectos como la optimización, desarrollo de arquitecturas derivadas (Switch Transformers, Longformer...) o escalabilidad. Así, una gran cantidad de nuevos modelos han ido surgiendo con el paso de los años, tal y como se aprecia en la siguiente imagen [Figura 2.5].

Tipos de transformers

Para terminar, vamos a hacer una clasificación de los distintos tipos de Transformers de cara a determinar cuales de ellos emplearemos en este proyecto. Ya que sabemos los componentes básicos que forman esta arquitectura, encontramos:

- **Modelos auto-regresivos**: Utilizan una arquitectura de solo decodificador. Por tanto, su funcionamiento se basa en la generación de texto token a token, donde cada uno es generado a partir de los anteriores.

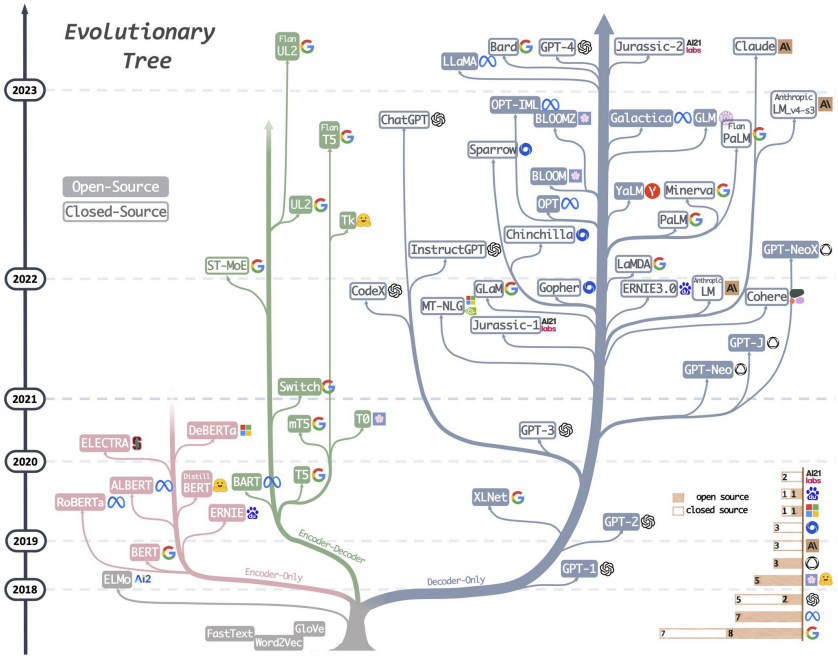

Figura 2.5: Árbol de evolución de los Transformers

Por este motivo se les conoce como autoregresivos. Ejemplos claros son los modelos GPT de OpenAI o LLaMA de Meta.

- **Modelos auto-codificadores**: Utilizan una arquitectura de solo codificador. Por tanto, codifican toda la secuencia de entrada en una representación continua, sin tener en cuenta la autoregresión. Esto genera que la salida del modelo se puede utilizar para tareas como clasificación, etiquetado de secuencias o generación de texto condicionada. Por ejemplo, BERT desarrollado por Google y sus diversas adaptaciones (RoBERTa, DistilBERT, CamemBERT...).

- **Modelos de secuencia a secuencia**: Utilizan una arquitectura de codificador-decodificador. Codifican una secuencia de entrada en una representación intermedia y luego decodifican esta representación para generar la secuencia de salida. En resumen, es la arquitectura que hemos descrito anteriormente. Por ejemplo, el modelo BART de Meta o los modelos T5 y Transformer-XL de Google.

Por las características de este proyecto, nos enfocaremos sobre todo en los modelos auto-codificadores dado que trataremos problemas de clasificación sobre los que emplearemos NER y, como ya hemos visto, son modelos que se ajustan a este contexto. En menor medida, también exploraremos el funcionamiento de los auto-regresivos como GPT aplicados a este dominio.

Capítulo 3

Tecnologías y herramientas empleadas en el desarrollo del sistema

En el siguiente apartado se detallarán las tecnologías y herramientas clave que se han utilizado en el desarrollo del sistema que presentamos en este proyecto, junto con una justificación de la elección.

3.1. Google Colaboratory

Google Colaboratory, o Google Colab, es una plataforma basada en la nube que permite a los usuarios escribir y ejecutar código, Python o R, directamente desde el navegador. Es especialmente popular entre la comunidad de aprendizaje automático y ciencia de datos debido a su facilidad de uso y acceso a recursos computacionales potentes. Además, ayuda a evitar ciertas incompatibilidades que pueden surgir al ejecutar nuestros programas en un ordenador personal por programas externos que podamos tener instalados. Asimismo, permitiría la ejecución de nuestro programa en otros navegadores y sistemas distintos al nuestro debido a que depende de Google.

Este posee varias versiones, gratuitas y de pago, en nuestro caso solo nos hemos ceñido a la primera mencionada. En ella se proporciona un entorno con interfaz tipo Jupyter Notebook, permitiendo combinar código ejecutable con texto e incluso elementos multimedia. Asimismo, también permite el acceso a unidades de procesamiento gráfico (GPU) y tensorial (TPU), muy útiles para tareas de *deep learning*. Aunque de manera limitada, alrededor de 12 horas a no ser que se agoten los recursos computacionales antes. En cuanto a la RAM, permite un acceso de hasta 12.67GB. Otras limitaciones son la

desconexión automática si se detecta inactividad (lo cual cobra relevancia para ejecuciones largas) o espacio de almacenamiento temporal por defecto al devolver archivos de salida.

Por otro lado, posee integración con Google Drive, facilitando el acceso y gestión de archivos, así como la colaboración y acceso a la nube con otros usuarios a tiempo real.

Así, ya que vamos a tratar un problema que implica aprendizaje automático, con todo lo que eso conlleva (bibliotecas, GPU...), y Colab nos facilita mucho esta tarea así como la posibilidad de comentar paso a paso las decisiones tomadas, se ha decidido que puede ser una buena opción su empleo. Además, es una herramienta que ha sido usada en varias ocasiones durante el grado.

3.1.1. Python

Como lenguaje de programación, usaremos Python, el cual ya mencionamos que es soportado por Google Colaboratory. Python es un lenguaje de programación de alto nivel, interpretado y de propósito general, conocido por tener una sintaxis clara y legible. A nivel de uso destaca en campos de desarrollo web, análisis de datos, inteligencia artificial, aprendizaje atomático y muchos otros relacionados con ingeniería. En definitiva, es ideal para nuestro proyecto debido a que cuenta con un vasto ecosistema de bibliotecas y frameworks que facilitan el desarrollo de sistemas basados en inteligencia articifial y, entre ellos, NLP.

No solo con esto, también presenta otras ventajas como ser multiplataforma, fácil integración con otras tecnologías y sobre todo, posee una comunidad activa de desarrolladores que contribuyen al desarrollo de nuevas herramientas y a la resolución de problemas, por lo que podemos encontrar una gran cantidad de información en internet.

En cuanto a sus bibliotecas básicas dedicadas a aprendizaje automático se han empleado:

- **Pytorch**: Es una biblioteca de código abierto ampliamente utilizada para construir y entrenar modelos de aprendizaje profundo. Además, es bastante flexible y fácil de usar y tiene soporte nativo para CUDA, lo que permite aprovechar la aceleración de hardware proporcionada por las GPUs, en nuestro caso de Colab.
- **Pandas**: Es una biblioteca de código abierto utilizada para la manipulación y análisis de datos. Es muy útil dado que ofrece estructuras de datos para manipular tablas númericas y una gran cantidad de operaciones para trabajar con ellas.

- **Matplotlib**: Es una biblioteca para dibujar gráficas de calidad en una gran cantidad de formatos. Además, es bastante genérica y flexible.
- **Seaborn**: Es una biblioteca basada en Matplotlib, pero más especializada en gráficos estadísticos.

3.2. Hugging Face

Hugging Face[1] es una empresa de tecnología y una comunidad de desarrolladores conocida por su enfoque en el procesamiento del lenguaje natural (NLP), por lo que es ideal para nuestro cometido. Esta compañía ha desarrollado una serie de herramientas y bibliotecas de código abierto que facilitan la implementación y uso de modelos y datasets de aprendizaje profundo.

Entre sus características, destacamos que su plataforma ofrece *Model Hub*, es decir, permite a los desarrolladores compartir modelos de *deep learning*. Así, estos pueden ser buscados y descargados de cara a ser usados directamente, fine-tuning, o cualquier otra tarea. Por otro lado, posee una API intuitiva y bien documentada (tutoriales, ejemplos prácticos, etc), lo que facilita en gran medida el uso de modelos NLP. También cabe destacar los *spaces* de Hugging Face, una de sus plataformas que permite desplegar aplicaciones web interactivas que utilizan los modelos creados por los desarrolladores. Aunque no hayan sido empleados en el proyecto como tal, sí que han sido usados de cara a ver ejemplos simples de como funcionan ciertos algoritmos NER durante la etapa de investigación.

De entre todas las bibliotecas que proporciona, para este proyecto se han empleado:

- **Transformers**: Es la biblioteca más podera y versátil de las que ofrece. Su diseño facilita el uso de modelos NLP basados en algoritmos Transformers. Así, es considerada un estándar debido a su amplio soporte y herramientas para el aprendizaje profundo. Gracias a ella, podremos acceder a diversos modelos derivados de BERT y GPT. Además, tiene soporte para diversos frameworks de aprendizaje profundo, como Pytorch, ya mencionado. Por otro lado, posee herramientas de cara a la tokenización de texto, paso crucial en NLP. Las herramientas concretas que emplearemos se mencionarán en apartados posteriores.
- **Datasets**: Esta biblioteca proporciona estructuras de datos diseñadas para simplificar el acceso, carga, manipulación y evaluación de conjuntos de datos en tareas que impliquen NLP y AA. Asimismo, también

[1] `https://huggingface.co/`

destaca por su eficiencia, flexibilidad y compatibilidad con grandes volúmenes de datos y otras bibliotecas como Pandas.

- **Evaluate**: Su función es facilitar la evaluación de modelos de aprendizaje automático en tareas NLP. Así, es compatible con una gran cantidad de modelos de Hugging Face y proporciona métricas diversas como precisión, recall y F1-score, muy importantes cuando trabajemos con NER.
- **Seqeval**: Como tal no es una biblioteca de Hugging Face, pero es ampliamente utilizada junto con dichos modelos debido a su fácil integración. Haciendo sinergia con Evaluate, nos ayudará a la hora de tomar las métricas de modelos NER específicamente.
- **Accelerate**: Facilita el entrenamiento de modelos de aprendizaje automático en un amplia variedad de dispositivos (GPUs y TPUs) y entornos. Cobra bastante relevancia en aquellos casos donde necesitemos hacer costosos entrenamientos con GPU.

3.2.1. GLiNER

A continuación, dedicaremos especial atención a GLiNER pues es un modelo realmente reciente, 2023, del que actualmente no existen muchos desarrollos pero que ha presentado grandes resultados. Así, a fecha de 2024 este algoritmo no esta incluido en ninguna otra biblioteca de Hugging Face, sino que posee una propia (llamada GLiNER) a partir de la cual trabajaremos. Sin embargo, merece la pena realizar una mención a ciertas características de este algoritmo de una manera más detenida.

GLiNER (Generalist Model for NER), fue introducido en Noviembre de 2023 en su paper homónimo [5]. Este modelo está diseñado específicamente para la identificación y clasificación de entidades, concretamente es un Transformer bidireccional (Bidirectional Language Modeling, BiLM), es decir, tiene la capacidad de considerar el contexto de un token tanto de izquierda a derecha como viceversa dentro de una secuencia de texto. Con respectos a otros LLMs, facilita la extracción paralela de entidades y con técnicas zero shot muestra rendimientos mayores a otros modelos como GPT o fine-tuned LLMs para una amplia variedad de benchmarks.

Su arquitectura [Figura 3.1], emplea un BiLM que toma como entrada los tipos de entidades a clasificar, categorías, y un texto. Cada entidad es separada en tokens de modo que el BiLM devuelva una representación para cada una. Las representaciones de los tipos de entidades pasan por una red *FeedForward* mientras que las representaciones de las palabras de entrada pasan por una capa de representación de intervalo (*span representation layer*), en ambos casos el objetivo es generar sus embeddings. Finalmente, se

calcula el *score*, como de bien se ajusta una palabra a su categoría, entre 0 y 1 para el total de representaciones (mediante producto escalar y función sigmoide). En la figura en cuestión se puede ver como la representación (0,1), Alan Farley, tiene un alto *score* con la categoría "Persona".

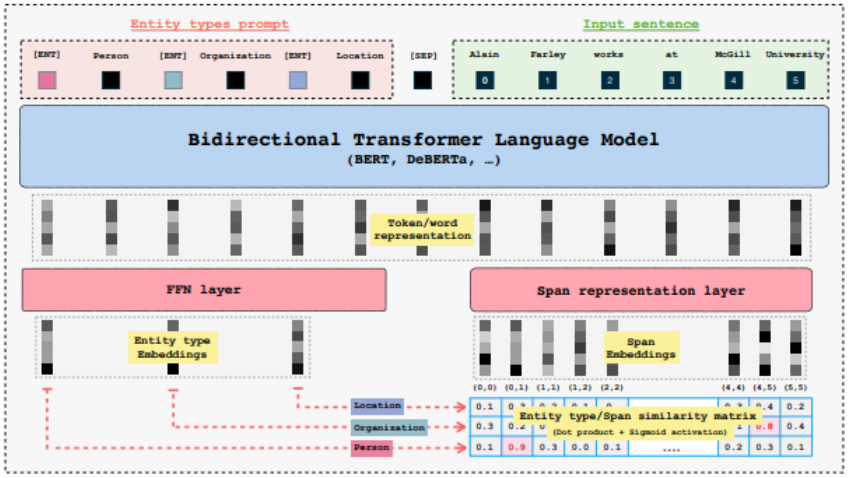

Figura 3.1: Arquitectura del modelo GLiNER

A partir de esto, el modelo fue evaluado de tres maneras. La primera, empleó un contexto zero-shot en benchmarks comunes para NER [Figura 3.3]; la segunda, se enfocó en evaluar el rendimiento en contextos multilingües [Figura 3.2]; y, la última, consistió en evaluar su actuación ante un conjunto en concreto de benchmarks para NER que son comúnmente usados para entrenamiento supervisado [Figura 3.3]. Se comparó con ChatGPT, UniNER (un modelo que emplea LLaMa fine-tuned sobre un dataset generado por ChatGPT) y InstructUIE (un modelo fine-tuned en varios datasets NER) en sus versiones más recientes. Presentó los siguientes resultados (F1-score):

	Language	Sup.	ChatGPT	GLiNER En	GLiNER Multi
Latin	German	64.6	37.1	35.6	**39.5**
	English	62.7	37.2	**42.4**	41.7
	Spanish	58.7	34.7	38.7	**42.1**
	Dutch	62.6	35.7	35.6	**38.9**
Non-Latin	Bengali	39.7	23.3	0.89	**25.9**
	Persian	52.3	25.9	14.9	**30.2**
	Hindi	47.8	27.3	11.3	**27.8**
	Korean	55.8	**30.0**	20.5	28.7
	Russian	59.7	27.4	30.3	**33.3**
	Turkish	46.8	**31.9**	22.0	30.0
	Chinese	53.1	18.8	6.59	**24.3**
Average		54.9	29.9	23.6	**32.9**

Figura 3.2: GLiNER Multilingüe

Dataset	ChatGPT	UniNER-7B	GLiNER-L
ACE05	26.6	**36.9**	27.3
AnatEM	30.7	25.1	**33.3**
bc2gm	40.2	46.2	**47.9**
bc4chemd	35.5	**47.9**	43.1
bc5cdr	52.4	**68.0**	66.4
Broad Tweeter	61.8	**67.9**	61.2
CoNLL03	52.5	**72.2**	64.6
FabNER	15.3	**24.8**	23.6
FindVehicle	10.5	22.2	**41.9**
GENIA	41.6	54.1	**55.5**
HarveyNER	11.6	18.2	**22.7**
MIT Movie	5.3	42.4	**57.2**
MIT Restaurant	32.8	31.7	**42.9**
MultiNERD	58.1	59.3	**59.7**
ncbi	42.1	60.4	**61.9**
OntoNotes	29.7	27.8	**32.2**
PolyglotNER	33.6	41.8	**42.9**
TweetNER7	40.1	**42.7**	41.4
WikiANN	52.0	55.4	**58.9**
WikiNeural	57.7	69.2	**71.8**
Average	36.5	45.7	**47.8**

Dataset	InstructUIE w/o	UniNER-7B w/	GLiNER-L w/	GLiNER-L w/o
ACE05	79.9	**86.7**	82.8	81.3
AnatEM	88.5	88.5	**88.9**	88.4
bc2gm	80.7	82.4	**83.7**	82.0
bc4chemd	87.6	**89.2**	87.9	86.7
bc5cdr	89.0	**89.3**	88.7	88.7
Broad Twitter	80.3	81.2	82.5	**82.7**
CoNLL03	91.5	**93.3**	92.6	92.5
FabNER	78.4	**81.9**	77.8	74.8
FindVehicle	87.6	**98.3**	95.7	95.2
GENIA	75.7	77.5	**78.9**	77.4
HarveyNER	**74.7**	74.2	68.6	67.4
MIT Movie	89.6	**90.2**	87.9	87.5
MIT Restaurant	82.6	82.3	**83.6**	83.3
MultiNERD	90.3	93.7	**93.8**	93.3
ncbi	86.2	87.0	**87.8**	87.1
OntoNotes	88.6	**89.9**	89.0	88.1
PolyglotNER	53.3	**65.7**	61.5	60.6
TweetNER7	**65.9**	65.8	51.4	50.3
WikiANN	64.5	**84.9**	83.7	82.8
wikiNeural	88.3	**93.3**	91.3	91.4
Average	81.2	**84.8**	82.9	82.1

Figura 3.3: GLiNER Zero-shot y GLiNER Supervisado, respectivamente

Cabe destacar que existen tres modelos GLiNER en función del número de parámetros: small, medium y large. Con 50M, 90M y 0.3B de parámetros, respectivamente. En sus evaluaciones se empleó la versión Large.

Brevemente, con zero-shot [Figura 3.3] se provó que la arquitectura GLiNER es bastante robusta y capaz de adaptarse a un amplio espectro de dominios, a excepción de aquellas áreas donde predomina el lenguaje coloquial e informal. En cuanto al contexto multilingüe [Figura 3.2], los resultados también fueron bastante prometedores en la mayoría de lenguas, sobresaliendo en aquellas que presentan influencia romance. En cuanto al dominio supervisado [Figura 3.3], el modelo presentó buenos resultados pero de manera mucho más ligera ya que el modelo UniNER le superó en varios benchmarks.

Por estos motivos, GLiNER ha demostrado ser un modelo bastante competitivo en la actualidad que es capaz de demostrar mejores rendimientos que otras arquitecturas reconocidas. Además, al ser tan reciente, la documentación y los recursos disponibles para su desarrollo y empleo son bastante limitados, por lo que es un buen punto de partida para ser de los primeros en poner en práctica su rendimiento en ámbitos reales y concretos, como es el campo de la medicina.

De este modo, volviendo a su biblioteca de Hugging Face, podremos acceder a una gran cantidad de métodos los cuales utilizaremos, previa explicación, para hacer un fine-tuning con nuestros datos.

3.2.2. BERT

También mostraremos especial atención a los modelos BERT y sus diferentes adaptaciones. Aunque en este caso sí que son realmente conocidos en varios campos de NLP.

Los modelos BERT (Bidirectional Encoder Representations from Transformers) también se basan en la arquitectura transformer y fueron desarrollados por Google y presentados en 2018. Al igual que GLiNER, es un BiLM cuya estructura está basada en codificadores. De hecho, fue de los primeros en emplearlo.

En cuanto a su preentrenamiento, destacamos dos tareas. La primera, es el modelado de lenguaje enmascarado (MLM, *Masked Language Model*) en donde aleatoriamente se selecciona una palabras de entrada, la cual es reemplazada con un token [MASK]. Se busca que el modelo pueda predecirla basándose en el contexto bidireccional. En segundo lugar, la predicción de la siguiente oración (NSP, *Next Sentence Prediction*), que implica predecir si una oración B sigue a otra A. Para ello el corpus lo componen pares de oraciones, algunas correctas y otras no, para que que el modelo pueda aprender a distinguirlas.

Como vemos, no son tareas propias de NER en sí. Esto se debe a que los modelos BERT han sido diseñados para enfocarse más en la comprensión de lenguaje y así luego ser extendido para tareas específicas del NLP. Por ello, presentan una estructura más genérica y flexible que puede ser ajustada mediante fine-tuning. Esta es la principal diferencia con GLiNER, el cual ha sido diseñado específicamente para NER a partir de varias optimizaciones y modificaciones.

Debido a esto, BERT es un modelo que presenta una cantidad de variantes realmente grande, destacamos las siguientes:

- **DistilBERT**: Es una versión más ligera y rápida de BERT. Sin embargo, mantiene prácticamente todo su rendimiento a la vez que reduce significativamente el tamaño del modelo.
- **A Lite BERT**: Conocido como ALBERT, introduce técnicas de factorización de parámetros y repetición de capas con el objetivo de reducir el tamaño del modelo y acelerar el entrenamiento sin afectar al rendimiento.
- **Robustly optimized BERT approach**: Conocido como RoBERTa, es una versión mejorada de BERT que se entrena con más datos y por más tiempo, eliminando la tarea de NSP.

Existen muchas más variantes. De todas ellas, en este proyecto nos hemos

decantado por el uso de RoBERTa debido a que sus mejoras han demostrado una capacidad de generalización superior en comparación con BERT [52], lo cual es de vital importancia en NER de cara al fine-tuning para captar los matices del lenguaje biomédico. En segundo lugar, al presentar un entrenamiento más intensivo, mejorará su robustez, lo cual es deseable cuando vamos a trabajar con datos variados y ruidosos como los presentes en casos clínicos.

Por último, el modelo RoBERTa que vamos a emplear ha sido preentrenado en el ámbito biomédico con textos en español. En concreto, ha sido desarrollado por el Plan de Tecnologías del Lenguaje (PlanTL) del Gobierno de España[2]. Su tarea principal se centra en Fill-Mask, es decir, rellenar una o varias palabras faltantes en una sentencia. Sin embargo, mediante fine-tuning puede ser extendido a más campos, como el que nos interesa, NER. En cuanto a su entrenamiento, empleó un corpus de casos biomédicos en español compuesto de una amplia y variada gama de fuentes (entre ellas Scielo, PubMed, MISC...).

3.3. OpenAI

OpenAI [3] es una empresa fundada en 2015 cuyo propósito reside en la investigación y desarrollo de modelos basados en inteligencia artificial, tocando específicamente los campos de aprendizaje automático y procesamiento de lenguaje natural. Originalmente, su objetivo era que dicho desarrollo fuera de código libre. A día de hoy, posee una gran cantidad de inversores, entre los cuales el más destacado es Microsoft y en cuya plataforma de supercomtuación basada en Azure da servicio a los sistemas informáticos de OpenAI.

Sus proyectos más destacados son sus modelos GPT (Generative Pretrained Transformer), basados en el procesamiento del lenguaje desde el enfoque generativo; DALL-E, otro modelo generativo capaz de crear imágenes a partir de texto; y Sora, de nuevo, un modelo generativo pero de texto a vídeo. Este último es realmente reciente, pues fue lanzado en febrero de 2024. Por último, hay que destacar ChatGPT, un chat bot basado en los modelos GPT que actualmente es capaz de leer, oír y hablar.

Además, presenta una API que permite a los usuarios trabajar y desarrollar sus propios modelos personalizados. Es este punto el que nos interesa para este proyecto, pues la emplearemos para imitar a un modelo NER y poder hacer una comparación de hasta que punto podría ser beneficiosa, viendo que ventajas y desventajas presenta.

El modelo en concreto que se va a emplear es GPT-3.5 Turbo, debido

[2] `https://huggingface.co/PlanTL-GOB-ES/roberta-base-biomedical-es`
[3] `https://openai.com/`

a que es rápido, económico y ha sido desarrollado para tareas simples. Este punto es importante, pues la API no es gratuita. Sin embargo, gracias a Fujitsu, ha sido posible su implementación y uso en este proyecto y este ha sido el modelo proporcionado.

3.3.1. GPT-3.5 Turbo

El modelo GPT-3.5 Turbo ofrecido por la API es el más rápido de todos los disponibles, así como el más económico. Sin embargo, no es el que mejores resultados puede ofrecer, pues otros más avanzados como GPT-4o (el más reciente) o GPT-4, son capaces de desarrollar labores que requieren más complejidad.

De entre las tareas que puede completar, destacamos realizar resúmenes, traducción, chatbot, entre otras muchas. En nuestro caso, nos centraremos en el análisis iterativo. Es decir, a partir de un prompt, entendiendo este como un texto inicial que proporcionamos al modelo para dirigir la generación de respuestas, iremos iterando por todos los casos clínicos con el objetivo de que el modelo puede analizarlos y detectar las entidades que nos interesan en base a las categorías proporcionadas. En resumen, queremos que nuestro modelo GPT actúe de la manera más similar posible a un modelo NER.

Esto es realmente interesante pues el propósito real de la estructura GPT es simplemente generativa, con todo lo que ello implica. Sin embargo, es tan potente que existe la posibilidad de abordar otras disciplinas con ella. Para entenderlo mejor, realizaremos una breve introducción a que son y como funcionan este tipo de modelos.

El modelo GPT-3.5 está basado en la arquitectura transformers y dado que su propósito es generativo, no disponen de codificadores. Su mecanismo es conocido como autoatención unidireccional enmascarada por este motivo. Así, sus decodificadores, como se ha mencionado anteriormente, los componen subcapas de atención y redes neuronales *feed-forward*. Al igual que otros modelos, disponen de varios tamaños en función del número de parámetros. El mayor de ellos está compuesto por 175 billones de parámetros, se entiende que GPT-3.5 Turbo emplea este, aunque aún no se ha divulgado el número de parámetros explícitamente. Motivo por el cual son capaces de capturar matices complejos del lenguaje y producir respuestas más precisas y relevantes.

3.4. Codalab

Codalab[4] es una plataforma en línea que ofrece un entorno para la colaboración, la competencia y la evaluación de algoritmos y modelos de diversas áreas, en nuestro caso nos centraremos en aprendizaje automático.

Una de las características más destacadas de esta plataforma es la capacidad de organizar competiciones en las que los participantes pueden cargar sus modelos y algoritmos, ejecutarlos sobre un conjunto de datos específicos (datasets) y comparar el rendimiento con el de otros participantes.

Así, se ha escogido el empleo de esta plataforma ya que a fecha de 2024 parte del equipo de Fujitsu, empresa que en acuerdo con la UGR ha propuesto diversos TFGs y este entre ellos, propuso una de estas competiciones[5]. En ella se proporciona un dataset cuyos datos, de los que hablaremos a continuación, se adaptan perfectamente a los algoritmos que vamos a emplear, previo preprocesado. Además, la plataforma posee un sistema de evaluación del que hablaremos en el siguiente apartado. Como tal, el objetivo no es participar activamente en la competición pero sí valernos de sus datos y procedimientos propuestos.

3.5. Dataset

3.5.1. Descripción

El corpus (dataset) con el que trabajaremos es denominado GenoVarDis, está compuesto por literatura científica sobre variantes de genomas, genes y enfermedades asociadas. Una de sus características más relevantes es que se encuentra en español, algo que actualmente es escaso en el dominio de NER y enfermedades. Principalmente, esto ha sido posible gracias a la traducción del inglés de diversos textos, así como su posterior revisión por expertos humanos.

Otro de los motivos por los que este dataset destaca, haciéndolo novedoso y relevante, es porque enfrenta importantes desafíos a la hora de reconocer entidades relacionadas con enfermedades y variantes, entendiendo este concepto como variaciones genéticas del ADN. Actualmente, los datasets NER relacionados con variantes son casi inexistentes, incluso en inglés, por ejemplo, tmVar3 [53] con aproximadamente 500 documentos o BERN2 [54] (que utiliza tmVar2) con solo 158 documentos. Entre otros motivos, esto se debe a que la mayoría de herramientas para este propósito emplean expresiones regulares, lo cual puede ser algo limitante a la hora de reconocer ciertas

[4] `https://codalab.lisn.upsaclay.fr/`

[5] `https://codalab.lisn.upsaclay.fr/competitions/17733`

variantes.

De este modo, este corpus es un desafío de cara a la identificación de variantes y sus enfermedades relacionadas, procedentes de literatura biomédica. Sobre todo en el ámbito de la medicina de precisión. Esto se debe al papel crucial a la hora de adaptar tratamientos y realizar diagnósticos personalizados basados en la genética del individuo. Además, entendemos que la compresión de enfermedades genéticas depende en gran medida de la recopilación y síntesis automatizada de conocimientos publicados en la literatura científica.

3.5.2. Estructura de los archivos

El contenido del dataset será explorado con detalle en capítulos posteriores pero de cara a comprender su estructura, explicaremos y mostraremos brevemente como se organizan sus archivos.

Este está compuesto por seis archivos en formato TSV, *Tab-Separated Values*, los cuales específicamente contienen información procedente de documentos de tmVar32 (de PubMed3, una base de datos de libre acceso que comprende artículos de investigación biomédica) junto con otros 200 casos procedentes de PubMed en español y de SciELO (otra biblioteca del mismo índole que PubMed).

Los dos primeros archivos son *train_text.tsv* y *train_annotation.tsv*, ambos relacionados. El primero está compuesto por 427 casos clínicos con las características mencionadas previamente. Su estructura [Tabla 3.1] la conforman tres columnas:

- **Pmid**: ID del caso clínico en PubMed.
- **Filename**: Nombre del documento de donde se extrajo el caso clínico.
- **Text**: Descripción del caso clínico.

En cuanto al segundo mencionado, contiene las anotaciones del primero. Es decir, cuales y donde se encuentran las entidades para cada caso clínico de *train_text.tsv*. Su estructura [Tabla 3.2] presenta la siguiente información:

- **Pmid**: ID del caso clínico en PubMed.
- **Filename**: Nombre del documento donde se extrajo el caso clínico.
- **Mark**: Identificador de la anotación dentro del documento.
- **Label**: Nombre del tipo de entidad en la anotación.

- **Offset1**: Comienzo del span en la descipción del caso clínico.
- **Offset2**: Fin del span en la descripción del caso clínico.
- **Span**: Secuencia de texto que define la entidad en cuestión.

A continuación, enumeraremos y definiremos los diferentes tipos de entidades que podemos etiquetar:

- **DNAMutation**: Variante en una secuencia de ADN.
- **SNP**: *Single Nucleotide Polymorphism*, son variaciones en una sola posición del ADN entre individuos de una especie.
- **DNAAllele**: Se refiere a un alelo en la secuencia de ADN. Un alelo es cada una de las diferentes formas que puede tomar un gen en una posición específica (locus) del cromosoma.
- **NucleotideChange/BaseChange**: Cambio en una base nucleotídica específica dentro de la secuencia de ADN (p.e. guanina a citosina).
- **OtherMutation**: Una variante cuya información es insuficiente.
- **Gene**: Tipo de gen.
- **Disease**: Cualquier tipo de enfermedad o síntoma.
- **Transcript**: Los transcritos son secuencias de ARN mensajero producidas de la transcripción del ADN y contienen la información genética necesaria para la síntesis de proteínas y otras funciones celulares.

A continuación, encontramos un tercer archivo, *dev_text.tsv*. Su estructura es exactamente igual que en *train_text.tsv* [Tabla 3.1] y su función es utilizar los casos clínicos descritos como conjunto de validación para nuestro algoritmo. Está compuesto por 70 casos. Al igual que en el caso anterior, también presenta un archivo *dev_annotation.tsv*, análogo a *train_annotation.tsv*.

Por último, encontramos los archivos *test_text.tsv* y *test_annotation.tsv*. El primero es el archivo test empleado para evaluar el modelo. Lo conforman 136 casos clínicos. Para realizar dicha evaluación solo será necesario emplear nuestro modelo con dichos datos y generar un archivo TSV análogo a la estructura de *train_annotation.tsv*, pero sin la columna mark. Dicho archivo lo subiremos a la competición de Codalab y el propio servidor nos dará las métricas de precisión, recall y F1-score para nuestros resultados.

Pmid	Filename	Text
12673366	pmid-12673366.txt	12673366|t|Análisis del polimorfismo G/C en la región no traducida 5' del gen RAD51 en cáncer de mama...
12716337	pmid-12716337.txt	12716337|t|Polimorfismo en la posición -174 del gen IL-6 se asocia con susceptibilidad a periodontitis crónica...
...	...	...

Cuadro 3.1: Estructura de archivos *train_text.tsv*, *dev_text.tsv* y *test_text.tsv*

Pmid	Filename	Mark	Label	Offset1	Offset2	Span
12673366	pmid-12673366.ann	T1	NucleotideChange	37	40	G/C
12673366	pmid-12673366.ann	T2	Gene	78	83	RAD51
12673366	pmid-12673366.ann	T3	Disease	87	101	cáncer de mama
...	...	...	...	...	...	...

Cuadro 3.2: Estructura de archivos *text_annotation.tsv*, *dev_annotation.tsv* y *test_annotation.tsv*

Capítulo 4

Especificación de requisitos

En esta sección, abordaremos la planificación detallada del proyecto a partir de la definición de requisitos funcionales, no funcionales y de documentación. Todo ello desde el marco de la investigación.

4.1. Requisitos funcionales

1. **Obtención de datos**: El sistema debe ser capaz de obtener los datos de manera eficiente y dar respuesta a consultas específicas.
2. **Generación de resultados**: El sistema debe generar y guardar los resultados, previo uso del modelo, en un archivo en formato TSV de estructura similar al proporcionado por el dataset.
3. **Visualización de gráficas**: El sistema debe proporcionar métodos para generar gráficas que representen los datos de manera visual y faciliten su comprensión.
4. **Visualización de progreso**: El sistema debe proporcionar información sobre el progreso del entrenamiento de nuestro modelo, mostrando información sobre las métricas obtenidas durante su ejecución.
5. **Visualización de resultados**: El sistema debe ser capaz de generar gráficas que representen los resultados obtenidos de manera visual.
6. **Evaluación de modelos**: El sistema debe ser capaz de evaluar diferentes algoritmos NER utilizando métricas estándar como la precisión, recall y F1-score.
7. **Optimización de parámetros**: El sistema debe incluir mecanismos para ajustar y optimizar los hiperparámetros de los modelos NER, buscando mejorar su rendimiento.

8. **Gestión de datos**: El sistema debe contar con funcionalidades para preprocesar los datos de entrada, asegurando que se encuentren en un formato adecuado para su análisis.

4.2. Requisitos no funcionales

1. **Rendimiento**: El sistema debe ser capaz de generar modelos que no experimenten degradación en el rendimiento durante su uso.
2. **Tiempo de respuesta**: El tiempo de respuesta del sistema durante la consulta de datos y evaluación de casos clínicos concretos debe ser ágil y no presentar demoras notables.
3. **Escalabilidad**: La arquitectura del sistema debe ser escalable, permitiendo la incorporación de nuevos o más cantidad de datos sin afectar la funcionalidad existente.
4. **Documentación**: El sistema debe estar documentado de manera exhaustiva y clara, proporcionando información suficiente para su uso y comprensión.
5. **Compatibilidad**: El sistema debe ser compatible con una variedad de navegadores web modernos y sistemas operativos.
6. **Diseño**: El sistema debe estar dividido en apartados claramente diferenciados, cada uno con un propósito específico.
7. **Privacidad**: El sistema debe cumplir con las regulaciones de privacidad y protección de datos relevantes, garantizando que los datos de los pacientes empleados se manejen de manera confidencial.
8. **Mantenibilidad**: El sistema debe estar diseñado de manera modular y documentada, facilitando su mantenimiento y actualización futura.
9. **Usabilidad**: El sistema debe ser intuitivo y fácil de usar para los investigadores.
10. **Portabilidad**: El sistema debe ser fácil de trasladar e implementar en diferentes entornos de hardware y software.

4.3. Requisitos de documentación

- **Documentación técnica**: Se debe proporcionar documentación técnica que describa la arquitectura del sistema, las tecnologías utilizadas y los detalles de implementación.

- **Ejemplos de uso**: La documentación debe incluir ejemplos suficientes para entender de dónde y cómo se pueden obtener los resultados mostrados.
- **Consideraciones de desarrollo**: La documentación puede abordar decisiones de diseño, problemas encontrados durante el desarrollo y soluciones aplicadas.
- **Instalación y configuración**: Si fuese necesario, se debe proporcionar información sobre cómo instalar, configurar y emplear las diferentes herramientas.
- **Manual de usuario**: El sistema debe proporcionar suficiente información como para que el usuario sea capaz de entender como utilizarlo.
- **Documentación de APIs**: Si el sistema incluye APIs (Interfaces de Programación de Aplicaciones), se debe proporcionar documentación completa de las mismas, incluyendo detalles como parámetros empleados y ejemplos de uso.
- **Pruebas y validación**: El sistema debe incluir información sobre el proceso de pruebas y validación, describiendo los casos de prueba, resultados esperados y criterios de aceptación.
- **Soporte y contactos**: Se deberá proveer de información de recursos de soporte en caso de que se necesitase asistencia adicional.

Capítulo 5

Planificación y presupuesto del proyecto

En este apartado detallaremos las distintas fases y cronología por las que ha pasado este proyecto, estableceremos un presupuesto estimado del coste de su desarrollo diferenciada en tareas y, finalmente, profundizaremos en el impacto en el medio ambiente que ostenta el uso de este tipo de modelos pues, a consideración personal, es un aspecto bastante relevante y que tiene mayor importancia de la que en un principio podemos darle.

5.1. Planificación temporal y económica

Para la realización y ejecución del proyecto, se han identificado diversas tareas que contribuyen a la consecución de los objetivos planteados. Así, destacamos varias fases de las cuales algunas son dependientes de otras, pues no pueden realizarse simultáneamente. A continuación, pasaremos a detallar en que ha consistido cada etapa:

5.1.1. Planteamiento del proyecto

Etapa consistente en definir el alcance y los objetivos del proyecto en base a consultas con los tutores. Para ello, se introdujeron los conceptos básicos a estudiar, posibles algoritmos, ejemplos de estado del arte y recomendaciones de software y corpus a emplear, entre otros.

De esta etapa dependen el reparto de tareas, la investigación y la búsqueda de software.

5.1.2. Reparto de tareas

Esta etapa aborda la asignación de estimaciones temporales para cada objetivo y tarea identificados. Asimismo, establecer un sistema de seguimiento y comunicación para garantizar el progreso y colaboración efectiva.

5.1.3. Investigación

Una fase de considerable extensión dentro del proyecto, se concentra en una disciplina específica de la inteligencia artificial, NLP, y dentro de esta, en particular, en NER. A pesar de su enfoque acotado, abarca una amplia gama de conceptos que van desde la revisión del estado del arte hasta la comprensión y aplicación de la inteligencia artificial y modelos NLP en el contexto del proyecto, con la integración indispensable en el ámbito clínico. Por consiguiente, el espectro de algoritmos existentes, sus variaciones y su funcionamiento es notablemente amplio. Con esto se quiere decir que esta etapa implica no solo la investigación sustancial, sino también la determinación del nivel de profundidad necesario. Esto se debe a que al abordar los fundamentos del proyecto, cada uno de ellos podría, en sí mismo, constituir otro proyecto independiente.

Por este motivo, esta etapa se realiza simultáneamente con muchas otras. Aunque, como es obvio, si, por ejemplo, vamos a emplear algoritmos Transformers, quiere decir que previamente ya ha pasado una "subetapa"para su investigación.

5.1.4. Búsqueda del software necesario

Esta fase engloba la búsqueda de herramientas que son necesarias para completar los objetivos propuestos. Así, no solo abarca que tecnologías web usar, sino también que algoritmos, que bibliotecas y, dentro de ellas, que métodos nos son más adecuados para nuestras necesidades. Asimismo, comprobaciones de que los corpus a emplear puedan ser usados para nuestro propósito o que lenguajes de programación nos convienen.

Como es obvio, hasta que dichos requisitos no hayan sido establecidos, no es posible comenzar la etapa de implementación.

5.1.5. Aprendizaje

Esta etapa se centra en la familiarización con las tecnologías a emplear en el proyecto. Conjuntamente con la búsqueda de software, una vez elegidas ciertas herramientas es muy importante consultar su documentación oficial y consultar recursos de aprendizaje en línea. Del mismo modo, se realizaron

cursos cortos sobre Prompt Engineering en DeepLearning (gratuito) o sobre conceptos básicos de los modelos Transformers en Codecademy.

De manera similar a la fase anterior, no se puede comenzar la implementación sin conocer las tecnologías a emplear, como mínimo de manera básica.

5.1.6. Implementación

Esta etapa engloba todo el desarrollo del proyecto a nivel práctico, de código. Desde la exploración y carga de datos, aprendizaje del modelo, estudio del corpus o preprocesado de los datos, entre otros aspectos. Es otra etapa de larga duración pues hay que dedicar tiempo a asegurar que no haya factores que perjudiquen el rendimiento de los modelos, el tiempo que dedican al aprendizaje o la creación de métodos de cierta complejidad.

5.1.7. Evaluación

Esta etapa es dependiente de la de implementación pues, una vez hemos adaptado nuestros modelos a nuestros datos, es cuando podemos comenzar la evaluación de los mismos a partir de unas métricas, permitiéndonos sacar conclusiones, detectar errores, problemas de rendimiento y realizar mejoras.

5.1.8. Documentación

Junto con el período de investigación, este es uno de los más largos. La realización de la memoria es un proceso relativamente lento en el que se debe explicar detalladamente los aspectos más importantes de este proyecto, eligiendo en que partes es necesario profundizar más y hasta qué punto. Todo con el objetivo de dar un contexto al trabajo a realizar, cómo se ha llevado a cabo y sacar unas conclusiones.

A continuación, se detalla gráficamente cada una de las etapas previamente mencionadas a lo largo de un periodo de varios meses mediante un diagrama de Gantt [Figura 5.1]. Las interdependencias discutidas se representan visualmente a través de líneas conectivas.

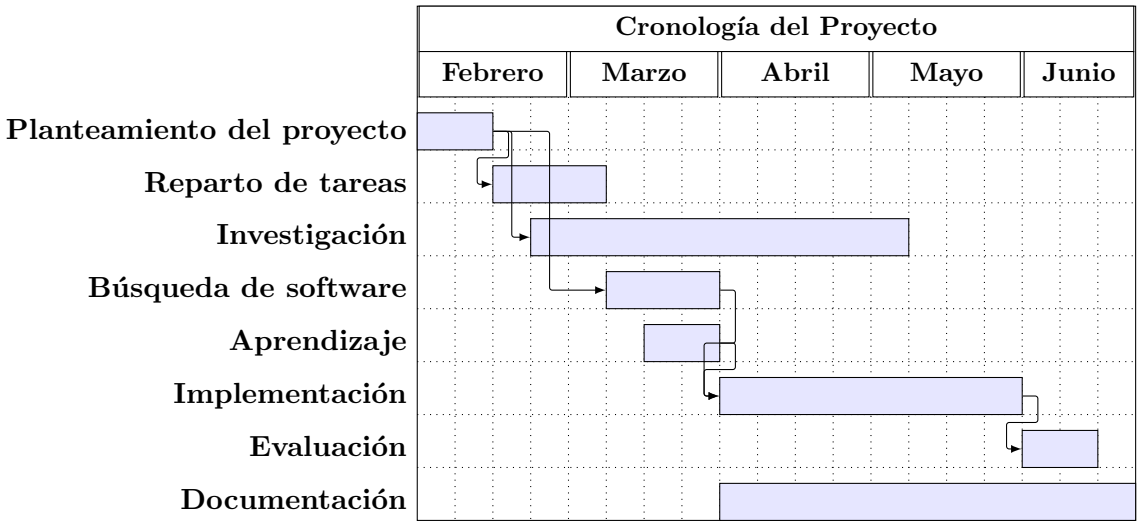

Figura 5.1: Diagrama de Gantt de la cronología del proyecto.

5.2. Presupuesto

Este apartado proporciona información sobre el presupuesto del proyecto. Esto incluye un desglose de los costos asociados con cada aspecto de este, como hardware, software, recursos humanos, capacitación, entre otros. Además, proporciona una estimación de los gastos totales y un análisis de su distribución en diferentes áreas tomando como referencia el diagrama de Gantt mostrado anteriormente [Figura 5.1]. Por último, se tendrán en cuenta aspectos relevantes como el coste ambiental que supone el empleo de estas tecnologías.

Para ello, comenzaremos presentando cuáles son las características del dispositivo empleado durante el desarrollo del proyecto:

Hardware:

- Procesador: AMD Ryzen 5 3400G with Radeon Vega Graphics, 3700 Mhz, 4 procesadores principales, 8 procesadores lógicos.
- Memoria física instalada (RAM): 8GB.
- Memoria virtual total: 11.9GB.
- Almacenamiento: 930GB.
- Placa base: B450M DS3H-CF versión x.x.
- Monitor: 1920x1080, 75Hz, HDMI 84.2kHz.

Software:

- Sistema operativo: Microsoft Windows 10 Home.

Entorno web:

- Google Colaboratory con disco de 78.2GB, RAM de 12.7GB y GPU de 15GB a partir del backend de Google Compute Engine para Python 3.

Partiendo de esto, encontramos que el equipo de desarrollo está compuesto por el alumno. El coste de un desarrollador será estimado a 22 euros por hora de trabajo. Además, también se contará con la supervisión de dos tutores, por parte de la universidad y de la empresa Fujitsu, que realizarán revisiones que pueden ir desde los treinta minutos a una hora, por lo que estimaremos 45 minutos por reunión cada dos semanas. El coste estimado de la supervisión es 70 euros la hora.

Tarea	Coste del desarrollador	Coste del tutor	Total
Planteamiento y reparto	110€	175€	285€
Investigación	2.340€	390,5€	2.730,5€
Búsqueda de software	220€	80,5€	300,5€
Aprendizaje	1.100€	168€	1.268€
Implementación	3.168€	87.5€	3.255,5€
Evaluación	528€	133€	661€
Documentación	2.112€	175€	2.287€
Total	9.578€	1.209,5€	10.787,5€

Cuadro 5.1: Coste del desarrollo del proyecto

Si a este costo le sumamos el precio de adquisición del hardware y software utilizado durante el proyecto (mencionado al comienzo), que ha sido aproximadamente de 600€, el costo total del proyecto ascendería a **11.387,5€**.

En cuanto al costo del empleo de la API de GPT-3.5 turbo, ya está incluido en la etapa de implementación. En concreto, el modelo empleado ha sido GPT-3.5 Turbo-0125 (pues existe otra variante más cara), cuyo precio por cada millón de tokens de entrada es 0,50 US$ y 1,50 US$ por cada millón de salida. Para hacernos una idea, 1.000 tokens equivaldrían aproximadamente a 750 palabras, que corresponde más o menos con los casos clínicos más extensos empleados.

El equipo de desarrollo del sistema como tal se compone únicamente del alumno, quien ha invertido significativamente en su formación y habilidades a través de una serie de cursos. Estos han sido ofrecidos por destacadas com-

pañías como DeepLearning[1], enfocada específicamente en la enseñanza de técnicas avanzadas de aprendizaje automático y profundo; y Codecademy[2], conocida por sus cursos interactivos de programación de todos los niveles, que ha recibido varios premios y reconocimientos. Gran parte del costo del aprendizaje se basa en la calidad y profundidad de la educación obtenida a partir de dichos cursos. A continuación, se presentan aquellos completados por el desarrollador:

DeepLearning

- **ChatGPT Prompt Engineering for Developers**: Enfocado en el funcionamiento de la ingeniería del prompt y sus capacidades para resumir, inferir, transformar, traducir y creación de chatbots, entre otras características. Todas ellas empleando ChatGPT como herramienta. Este curso es gratuito.

  Verificación de curso completado: `https://learn.deeplearning.ai/accomplishments/19951983-8989-4dd6-85ac-2ac6bf7af32f?usp=sharing`

Codecademy

- **Intro to Generative AI**: Similar al realizado en DeepLearning, ofrece una visión simple sobre las características a la IA generativa.

  ID de la credencial: 66365FCCEA o Figura 5.2.

Figura 5.2: Verificación QR de curso: Intro to Generative AI

- **Intro to Hugging Face**: Explica qué es Hugging Face, su funcionamiento, sus distintos apartados, bibliotecas, etc. Profundizando en cuales son las más utilizadas y como emplearlas en proyectos reales.

  ID de la credencial: 663665297F o Figura 5.3.

- **AI Transformers**: Explica de manera simple en que consiste la estructura Transformer, que tipos existen junto con ejemplos, su evolución y, finalmente, enseña como trabajar con ellos usando las bibliotecas de Hugging Face.

[1] `https://www.deeplearning.ai/`
[2] `https://www.codecademy.com/`

Figura 5.3: Verificación QR de curso: Intro to Hugging Face

ID de la credencial: 663CF4FBAB o Figura 5.4.

Figura 5.4: Verificación QR de curso: AI Transformers

- **Finetuning Transformer Models**: Es el curso más complejo de los realizados. Enseña como realizar un fine-tuning de LLMs de manera eficiente, así como preparar, entrentar y optimizar nuestros modelos para dicho propósito.

  ID de la credencial: 663F42F7DF o Figura 5.5.

Figura 5.5: Verificación QR de curso: Finetuning Transformer Models

Es relevante señalar que los cursos de Codecademy realizados no fueron gratuitos, aunque la plataforma ofrece una versión sin coste con contenido limitado. Dado que el alumno ya disponía de un plan de suscripción, aprovechó esta oportunidad para ampliar sus conocimientos y aplicarlos en este proyecto.

Estos cursos no solo demuestran el compromiso del alumno con la mejora continua de sus habilidades, sino que también son un testimonio de su capacidad para enfrentar desafíos complejos en el desarrollo de software. La aplicación de los conocimientos adquiridos en estos cursos al proyecto se espera que resulte en un producto de mayor calidad y funcionalidad.

El costo del aprendizaje refleja no solo el tiempo invertido en la adquisición de nuevas habilidades, sino también la confianza en la capacidad del desarrollador para contribuir significativamente al proyecto.

5.3. Impacto ambiental de modelos Transformers

Cuando nos referimos a inteligencia artificial, normalmente la consideramos como una entidad abstracta y etérea que no tiene impacto en nuestra realidad física. Sin embargo, esto no así.

Nuestros modelos de aprendizaje son ejecutados en hardware físico impulsado por electricidad. Y, si nos centramos en nuestro caso, los modelos de aprendizaje profundo y, en concreto, los Transformers presentan un poder computacional tremendo. Por tanto, la cantidad de electricidad consumida durante su entrenamiento e inferencias es realmente grande y, a no ser que proceda de fuentes de cero emisiones, eso se traduce en una gran cantidad de gases de efecto invernadero (CO_2) liberados a la atmósfera.

Uno de los primeros papers en tratar este tema (en 2019), *Energy and Policy Considerations for Deep Learning in NLP* [55], menciona textualmente (traducido al español):

Los avances recientes en hardware y metodología para el entrenamiento de redes neuronales han dado paso a una nueva generación de grandes redes entrenadas con abundantes datos [... que] dependen de la disponibilidad de recursos computacionales excepcionalmente grandes que requieren un consumo de energía igualmente sustancial.

Como resultado, estos modelos son costosos de entrenar y desarrollar, tanto financieramente [. . .] como ambientalmente, debido a la huella de carbono requerida para alimentar el hardware moderno de procesamiento de tensores.

Es decir, a más poder computacional, más energía. De hecho, de acuerdo a la Universidad de Stanford en su *AI Index 2023 report*, donde analiza la situación de la IA en dicho año, GPT-3 tuvo un total de emisiones equivalente a 500 vuelos de Nueva York a San Francisco [Figura 5.6].

Hay que tener en cuenta que GPT-3 fue lanzado en 2020, lo que implica que a día de hoy existan muchos modelos más potentes, algunos de ellos mencionados en el estado del arte previamente. Sin embargo, en la actualidad los LLMs más potentes no son de código abierto por lo que no podemos saber cuáles son sus huellas de carbono. Otro ejemplo interesante del que se dispone información es LLaMA 2 [Figura 5.7]:

Tanto en este caso como en el previo de GPT-3 la huella de carbono mostrada fue liberada durante el preentrenamiento, pues es la etapa en la

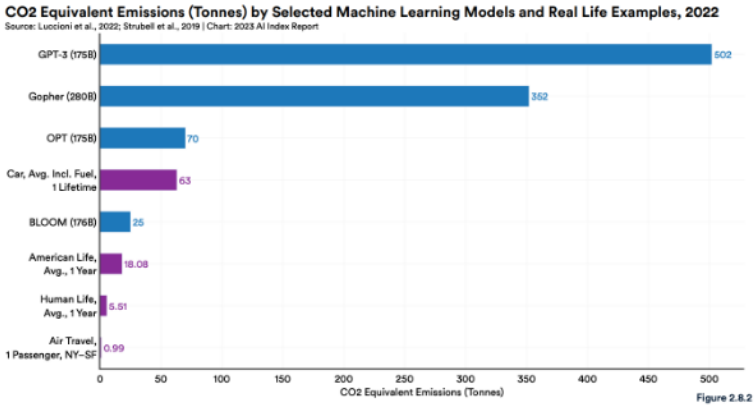

Figura 5.6: Emisiones CO_2 de modelos de AA (azul) en comparación con contextos cotidianos (morado).

		Time (GPU hours)	Power Consumption (W)	Carbon Emitted (tCO_2eq)
Llama 2	7B	184320	400	31.22
	13B	368640	400	62.44
	34B	1038336	350	153.90
	70B	1720320	400	291.42
Total		3311616		539.00

Figura 5.7: Emisiones CO_2 de LLaMA 2 para distintos tamaños.

que los modelos Transformers consumen recursos muy intensivamente.

Por otro lado, también ha habido intentos de cuantificar las emisiones de los modelos durante su etapa de inferencia (aquella donde se usa el modelo preentrenado para realizar predicciones). Aunque sí que es verdad que es más pequeña que durante el preentrenamiento, hay que tener en cuenta que es una etapa que se realiza mucho más frecuentemente por lo que podría alcanzar una cantidad significativa. El paper *Power Hungry Processing: Watss Driving the Cost of AI Deployment* [56] examina el coste de una gran variedad de tareas de inferencia [Figura 5.8] (realizando 1000 peticiones para cada una) de Transformers y encuentra que la generación de contenido, especialmente de imágenes, es la que presentan mayores emisiones de carbono.

Y con respecto a lo que nos interesa en este proyecto, si nos fijamos en la tarea de Token Classification [Figura 5.8], pues es la que realizamos al usar NER, vemos que es una de las que produce menos emisiones. Además, si nos fijamos en la caja en cuestión, encontramos que la variabilidad de emisiones

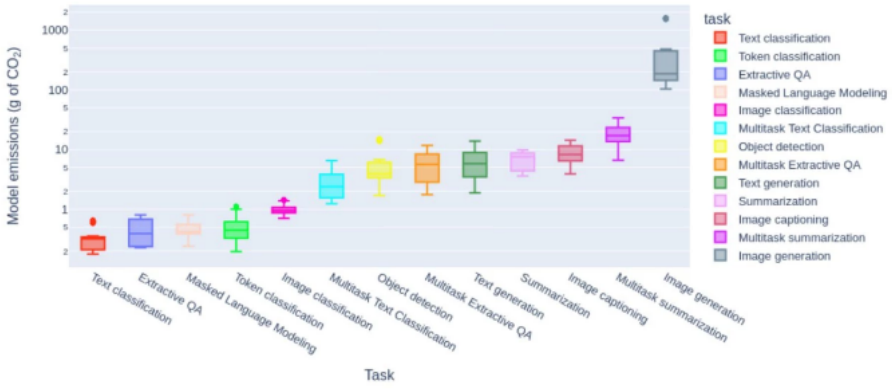

Figura 5.8: Diagrama de cajas de huellas de carbono según la tarea de inferencia de modelos Transformers.

no es tan preocupante, pues la longitud de esta es bastante pequeña; además, la mediana se encuentra justo en el centro, lo que nos indica que las emisiones son consistentes en torno a ese valor.

En términos de posibles soluciones, resaltamos la importancia de seleccionar un modelo cuyo tamaño sea adecuado para los objetivos que se desean lograr. Por ejemplo, LLaMA ofrece diversas opciones de tamaño, como se ilustró previamente (véase [Figura 5.7]), al igual que lo hace otro algoritmo que utilizaremos, GLiNER, los cuales fueron mencionados en secciones anteriores y serán discutidos más detalladamente posteriormente. Por lo tanto, es fundamental considerar que muchas tareas pueden abordarse satisfactoriamente utilizando modelos de menor escala, sin necesidad de recurrir a opciones más grandes.

Por supuesto, también se podría mejorar el tipo de fuente de energía utilizada por los centros responsables del entrenamiento de los modelos para reducir las emisiones. Sin embargo, dicha mejora queda fuera de nuestro control y puede requerir acciones a nivel institucional o gubernamental.

Como objetivo de concienciación sobre un aspecto relevante en el empleo de modelos Transformer, se propone considerar un aumento de costos asociado con este proceso. En este contexto, recomendamos establecer una estimación de 200 euros por cada hora de entrenamiento durante el proceso de fine-tuning. Esta medida no solo busca internalizar los costos operativos asociados con el consumo de recursos computacionales y energía durante el entrenamiento de los modelos, sino que también pretende destacar la importancia de considerar el impacto ambiental y los recursos necesarios en la implementación de dichas tecnologías. Así, el costo total aumentaría aproximadamente 600 euros dando un nuevo total de **11.987,5€**.

Capítulo 6

Planteamiento

En este capítulo detallaremos cuál es el problema a tratar de una manera más concreta en base a la información vista en capítulos anteriores. Así, no solo haremos una descripción del mismo, sino que también abordaremos las perspectivas sobre las que se ha resuelto y de qué modo hemos medido sus resultados.

6.1. Descripción del problema

De manera simplificada, el objetivo del proyecto es ayudar a la codificación automática de enfermedades y procedimientos y, para ello, planteamos el empleo de modelos NER. Para conseguirlo, deberemos de emplear dichos modelos sobre un conjunto de casos clínicos, almacenados en el corpus GenoVarDis descrito previamente.

Por lo tanto, lo primero es entender en detalle los datos con los que vamos a trabajar, lo que se conoce como exploración. Será bastante importante pues no solo nos dará pistas sobre como trabajar con ellos, sino que nos ayudará entender mejor los resultados obtenidos en el futuro.

Este proyecto presenta tres enfoques. Uno de ellos es emplear la famosa arquitectura GPT de cara a la detección de entidades, es decir, queremos que actúe como un modelo NER tanto como sea posible. Esto implica que la clave va a residir en el prompt que empleemos, pues queremos que el modelo sea capaz de entender lo mejor posible cual es su función. Esto se debe a que para este caso no existe ninguna fase de entrenamiento, por lo que la única manera de evaluar su rendimiento es a partir de los resultados que proporcione sobre el conjunto de test directamente.

A continuación, procederemos a estudiar cuáles son los resultados de emplear nuestro corpus con modelos NER como tal. En específico, lo eva-

luaremos con un modelo de tipo BERT, como RoBERTa; y con GLiNER. Así, en estos casos sí que destacamos etapas de entrenamiento y experimentación. Sin embargo, para ello usaremos técnicas como fine-tuning, pues no tendría sentido (ni capacidad computacional) entrenar los algoritmos desde cero. Estas medidas suponen que la elección de hiperparámetros cobre bastante relevancia a la hora de obtener mejores o peores resultados, por lo que dedicaremos bastante atención a ellos.

Finalmente, pondremos sobre la mesa una comparación de los resultados de estas tres evaluaciones, tanto de manera individual como colectiva en base a diversas tablas y gráficas, en especial matrices de confusión, que nos ayudarán a ello.

6.2. Métricas a emplear

Para la evaluación de los resultados, emplearemos las métricas más comunes al usar NER, es decir, precisión, recall y F1-score. Siendo esta última a la que más atención prestaremos pues hay que recordar que es la media armónica de las anteriores. Cuánto más cercanas a 1 sean, mejores resultados; aunque más adelante discutiremos algunas consideraciones al respecto.

Por otro lado, ya veremos que durante el fine-tuning emplearemos métodos ya existentes y optimizados por las propias bibliotecas. Algunos de ellos nos van a proporcionar otras métricas que no mencionamos previamente, como la exactitud, o *accuracy*. Esta se define como el ratio de las predicciones correctas entre las totales realizadas. Lo cual podemos expresarlo como:

$$Accuracy = \frac{Verdaderos\ positivos\ +\ Verdaderos\ negativos}{Total\ predicciones}$$

La analizaremos y discutiremos hasta que punto podemos considerarla adecuada.

Por último, aunque no son métricas, también prestaremos atención a las pérdidas de entrenamiento y validación de cara a entender cual es la actuación de nuestro modelo o si presenta problemas como *overfitting*[1] o, por el contrario, *underfitting*. Al contrario que los casos anteriores, las pérdidas deben tender a decrecer con el tiempo, por lo que cuanto más cerca se encuentren de cero, por lo general, mejores resultados obtendremos.

[1]El *overfitting* es un fenómeno común en el aprendizaje automático en donde un modelo se ajusta demasiado a los datos de entrenamiento y, como resultado, tiene un rendimiento deficiente en datos nuevos o no vistos. En cambio, cuando no puede ajustarse correctamente, se conoce como *underfitting*.

Capítulo 7

Implementación

En este capítulo se detallará minuciosamente el desarrollo del proyecto a nivel de implementación. Así, se analizarán las bibliotecas utilizadas, se explicará la justificación de su elección y se describirán los métodos empleados, tanto de las bibliotecas como propios.

Primero, realizaremos un estudio exhaustivo de los datos iniciales. Este paso es conocido como análisis exploratorio y es fundamental en proyectos de aprendizaje automático.

Posteriormente, explicaremos la implementación como tal de los modelos empleados, el prototipado, abordando etapas clave como el preprocesamiento de datos y entrenamiento de los modelos. Asimismo, se discutirán las ventajas y limitaciones que se han presentado durante el proceso.

El objetivo es garantizar una compresión clara del proceso de implementación en donde se resalte la importancia de cada etapa y los métodos empleados para alcanzar los objetivos del proyecto.

7.1. Carga y exploración de datos

Comenzaremos sentando las bases para el futuro análisis de resultados. Para ello, primero describiremos el proceso de carga de los datos.

Recordemos que estamos trabajando en un entorno de Google Colaboratory con integración con Google Drive. Por lo que debemos establecer una estructura de directorios:

```
TFG_VictorOliverosVillena/
|-- Data/
|   |-- train_annotation.tsv y train_text.tsv
|   |-- dev_annotation.tsv y dev_text.tsv
|   |-- test_annotation.tsv y test_text.tsv
|-- EvaluationTSV/
|   |-- Archivos TSV generados al evaluar un modelo.
|-- Logs/
|   |-- Almacena los modelos tras el fine-tunig.
|-- LoadExploration/
|   |-- Archivos ipynb sobre carga y exploración de datos.
|-- Prototypes/
|   |-- Archivos ipynb con los prototipados de los modelos.
|-- Experimentation/
|   |-- Archivos ipynb con la evaluacion de los modelos.
```

Así, para la carga y exploración de datos existe un archivo CargaExploracion.ipynb. En él, lo primero que tenemos que hacer es montar Google Drive en el sistema de archivos del entorno de ejecución con el objetivo de manipular los archivos que tenemos almacenados en nuestro Google Drive dentro del entorno Colab.

Listing 7.1: Montaje de Google Drive

```
drive.mount('/content/drive')
```

A continuación, se procede a la lectura de los archivos. Para ello, la mejor opción sin duda es usar la biblioteca Pandas y emplear sus estructuras de datos especializadas para el manejo de estos, como los Dataframes. Su funcionamiento queda fuera del propósito de este proyecto, por lo que simplemente nos limitaremos a explicar los métodos en específico de estos que hayamos empleado.

Listing 7.2: Carga de archivos en Dataframes

```
# Ruta de los archivos TSV en Google Drive
path_train_text = '/content/drive/MyDrive/
    TFG_VictorOliverosVillena/Data/train_text.tsv'
path_train_annotation = '/content/drive/MyDrive/
    TFG_VictorOliverosVillena/Data/train_annotation.tsv'
path_dev_text = '/content/drive/MyDrive/
    TFG_VictorOliverosVillena/Data/dev_text.tsv'
path_dev_annotation = '/content/drive/MyDrive/
    TFG_VictorOliverosVillena/Data/dev_annotation.tsv'
path_test_text = '/content/drive/MyDrive/
    TFG_VictorOliverosVillena/Data/test_text.tsv'
```

```
7
8  # Lectura de archivos TSV mediante pandas
9  train_text = pd.read_csv(path_train_text, sep='\t')
10 train_annotation = pd.read_csv(path_train_annotation, sep
       ='\t')
11 dev_text = pd.read_csv(path_dev_text, sep='\t')
12 dev_annotation = pd.read_csv(path_dev_annotation, sep='\t
       ')
13 test_text = pd.read_csv(path_test_text, sep='\t')
```

Los archivos TSV tienen un formato parecido a los CSV, a excepción de que usan tabuladores como separadores. Indicando esta distinción, podemos leer los archivos con el método *read_csv* de Pandas. En este punto, ya tenemos todos los archivos que nos interesan cargados en nuestro entorno de ejecución.

De cara a la exploración, prescindimos del archivo de test, pues es irrelevante a la hora entrenar nuestro modelo. En caso contrario, caeríamos en lo que se conoce en aprendizaje automático como *Data Snooping*, que podemos definirlo como el uso inapropiado de los datos para encontrar patrones que podrían parecer significativos pero que en realidad son el resultado de un análisis exhaustivo y oportunista que realmente lo que hace es aumentar el riesgo de información falsa positiva.

Así, los dataframes correspondientes a los archivos text tendrían tres columnas cada uno (pmid, filename y text) [véase Tabla 3.1], mientras que los archivos annotation tienen siete (pmid, filename, mark, label, offset1, offset2 y span) [véase Tabla 3.2].

Los archivos que realmente nos proporcionan información relevante para el análisis son los annotation, ya que contienen información sobre las entidades a encontrar y su tipo. El texto como tal del caso clínico de los archivos text ahora mismo nos es irrelevante. Ya que ambos van a ser utilizados durante el entrenamiento, podemos concatenar los dataframes de annotation para obtener una información más clara de los datos. Es importante destacar que esto lo hacemos con propósitos de exploración, de modo que no realizaremos cambios ni ajustes basados en el análisis combinado, pues estaríamos contaminando el modelo. Esto supondría una posible sobreestimación, u *overfitting*, afectando a la generalización[1] del mismo.

Listing 7.3: Concatenación de Dataframes para análisis exploratorio

```
1 data = pd.concat([train_annotation, dev_annotation],
      ignore_index=True)
```

[1]Cuando hablamos de generalización, nos referimos a la capacidad de nuestro modelo entrenado para realizar predicciones precisas en datos nuevos o no vistos, que no fueron utilizados durante el proceso de entrenamiento.

El resultado es un Dataframe de 8199x3, siendo 8199 la suma de las filas de ambos archivos annotation. Donde recordemos cada instancia corresponde a un entidad detectada en un determinado caso clínico.

A continuación, mostraremos diversas gráficas con el objetivo de conseguir un mejor entendimiento de los datos. Para ello, se han empleado las bibliotecas Matplotlib y Seaborn. Comenzaremos con una bastante simple pero importante, una gráfica de frecuencias de tipos de entidades.

Listing 7.4: Generación de gráfica de frecuencias

```
data.groupby('label').size().plot(kind='barh', color=sns.
    palettes.mpl_palette('Paired'))
plt.ylabel('Tipos de Entidades')
plt.xlabel('Frecuencia de entidades')
plt.gca().spines[['top', 'right',]].set_visible(False)

# Mostramos el numero concreto de apariciones
frecuencia_labels = data['label'].value_counts()
print(frecuencia_labels)
```

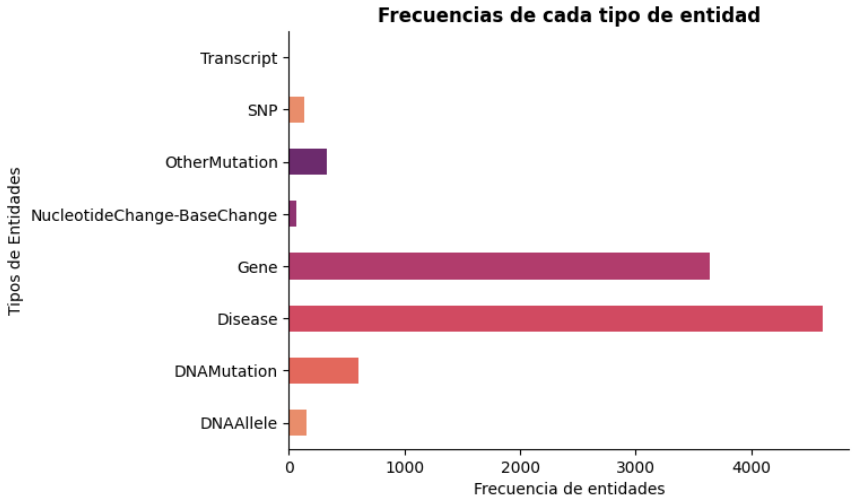

Figura 7.1: Gráfica de frecuencias

Esta gráfica [Figura 7.1] muestra la cantidad de veces que cada tipo de entidad aparece en total entre todos los casos clínicos.

Lo primero que vemos es una alta frecuencia de los tipos Disease y Gene [Tabla 7.1], lo cual nos puede indicar que van a ser el tipo de entidad prevalente en futuras predicciones. En cierto modo es lógico, ya que es típico

Cuadro 7.1: Frecuencias concretas por tipo de entidad

Label	Frecuencia
Disease	4616
Gene	3643
DNAMutation	599
OtherMutation	324
DNAAllele	151
SNP	135
NucleotideChange-BaseChange	62
Transcript	2

en textos médicos. Por el contrario, las demás etiquetas tienen frecuencias mucho menores, lo cual ya no solo indica que son mucho menos comunes, sino también que la dificultad de nuestro modelo para detectarlas aumentará. Para tipos concretos como Transcript, será especialmente complicado, pues solo existen dos instancias.

Por tanto, debemos ser conscientes de que partimos de un conjunto donde los tipos de entidades están desbalanceados y que esto afectará a los resultados futuros. Una solución podría ser realizar un sobremuestreo, una técnica del AA que consiste en aumentar el número de ejemplos de las clases minoritarias. Sin embargo, al tratar el dominio médico, esto es bastante difícil pues necesitaríamos una opinión experta para asegurar que no hay errores durante la adición, tal y como vimos cuando describimos el dataset. Por el contrario, podríamos realizar un submuestro, que consiste en eliminar instancias de la clase mayoritaria. Tampoco sería una solución adecuada, pues nuestro dataset ya de por sí no es demasiado extenso y etiquetas como Transcript nos obligarían a reducirlo en exceso, dando lugar a más inconvenientes que ventajas. Así, trabajaremos con el corpus original, siempre siendo conscientes de estas limitaciones.

Incluso, podríamos hacer un análisis más exhaustivo aún. Existe la posibilidad de que, para un tipo de entidad concreta, la entidad detectada en sí sea la misma. Para ello, plantemos mostrar una gráfica de frecuencias pero basada en entidades únicas por tipo.

Listing 7.5: Generación de gráfica de frecuencias para entidades únicas

```
1 data.groupby('label')['span'].nunique().plot(kind='barh',
      color=sns.color_palette('Paired'))
2 plt.xlabel('Tipo de Entidad')
3 plt.ylabel('Numero de Entidades Unicas')
4 plt.title('Numero de Entidades Unicas por Tipo',
     fontweight='bold')
5 plt.show()
```

```
6
7 # Mostramos el numero concreto de apariciones
8 frecuencia_labels = data.groupby('label')['span'].nunique
    ()
9 print(frecuencia_labels)
```

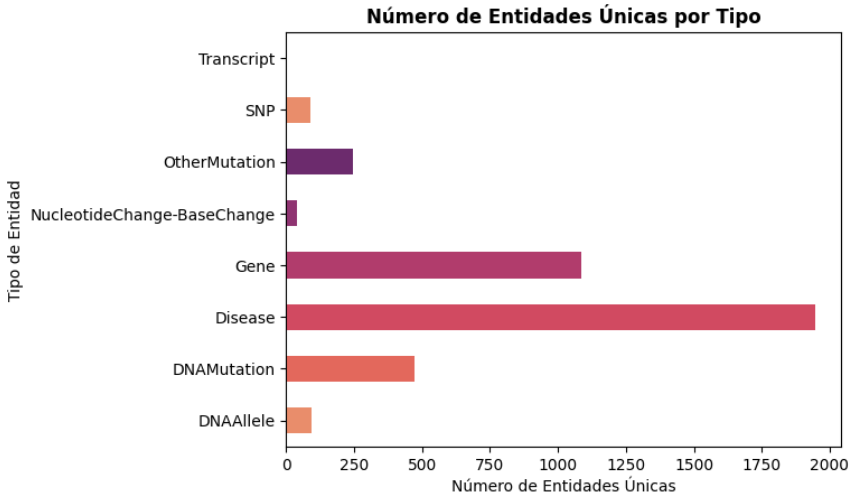

Figura 7.2: Gráfica de frecuencias para entidades únicas por tipo

Cuadro 7.2: Frecuencias concretas por entidades únicas por tipo

Label	Frecuencia
Disease	1946
Gene	1087
DNAMutation	471
OtherMutation	245
DNAAllele	95
SNP	87
NucleotideChange-BaseChange	41
Transcript	2

Si nos fijamos en la Tabla 7.2 y la comparamos con la Tabla 7.1 podemos darnos cuenta que en todos los casos y sobre todo en los tipos más comunes, la frecuencia ha disminuido. Esto nos ayuda ya que podemos sacar conclusiones más representativas. Si en un conjunto de datos encontramos un tipo de entidad muy abundante pero luego vemos que la entidad como tal es siempre la misma, el modelo aprenderá muy bien a detectarla, pero luego no generalizará correctamente cuando se encuentre con nuevos casos. Con esto queremos decir que cuando entrenemos el modelo, este aprenderá mejor no solo a detectar aquellos tipos con mayores frecuencias, sino también aquellas

palabras que más se repitan para un mismo tipo de entidad. Esto es positivo siempre y cuando haya un conjunto variado de entidades para cada tipo.

Aún así, la distribución como tal no ha variado, pues el orden de frecuencia de aparición por tipos no varía. En ese sentido, podemos mantener las conclusiones que sacamos al analizar la gráfica anterior [Figura 7.1].

Basándonos en esto, una representación visual bastante buena y que todos hemos visto alguna vez es una nube de palabras. Con la biblioteca WordCloud la podemos generar fácilmente.

Listing 7.6: Generación de nube de palabras

```
text = ' '.join(data['span'])
wordcloud = WordCloud(width=800, height=400,
    background_color='white').generate(text)

plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

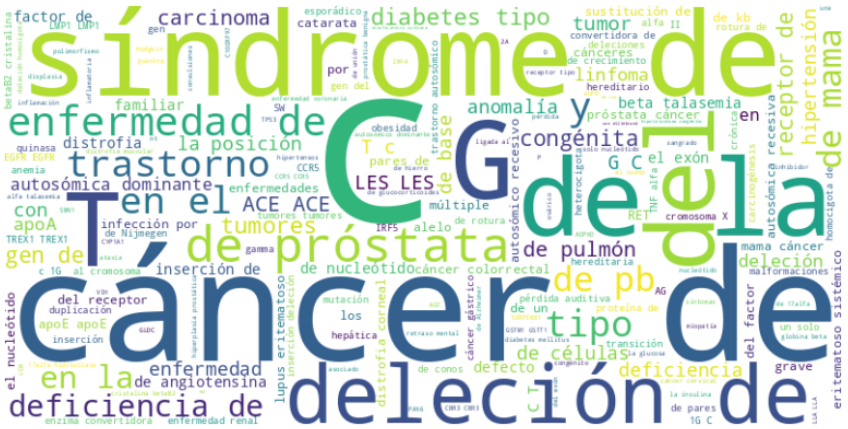

Figura 7.3: Nube de palabras

En esta nube, el tamaño de los distintos términos viene determinado por su frecuencia. Así, vemos que palabras como 'cáncer' o 'síndrome' son bastante comunes, siendo estas enfermedades. Sin embargo, el corpus sí que presenta una amplia variedad para los distintos tipos de cáncer o síndromes, en este caso, pues no aparecen en la nube. También destacamos la aparición de letras como C (citosina), G (guanina) o T (timina), tres de las cuatro bases nitrogenadas del ADN. Esto es realmente lógico pues, dependiendo del contexto, podrían ser clasificadas con distintas etiquetas. Por ejemplo, 'sustitución G por C en la posición 135' correspondería al tipo DNAMu-

tation mientras que 'conversión de C a T' sería de tipo NucleotideChange-BaseChange. O incluso podría referirse a algún gen, como la cinasa, la cual también se puede representar mediante C.

En resumen, esta nube nos ayuda a visualizar mejor las conclusiones anteriores. Siendo capaces de ver como los conceptos más abundantes corresponden a enfermedades y genes y aquellos menos comunes ni siquiera aparecen.

Listing 7.7: Generación de gráfica de distribución de entidades por documento

```
entities_per_doc = data.groupby('filename')['span'].count()
sns.histplot(entities_per_doc, bins=30, kde=True, color='green')
plt.xlabel('Numero de Entidades por Documento')
plt.ylabel('Frecuencia')
plt.title('Distribucion de entidades por documento', fontweight='bold')
plt.show()
```

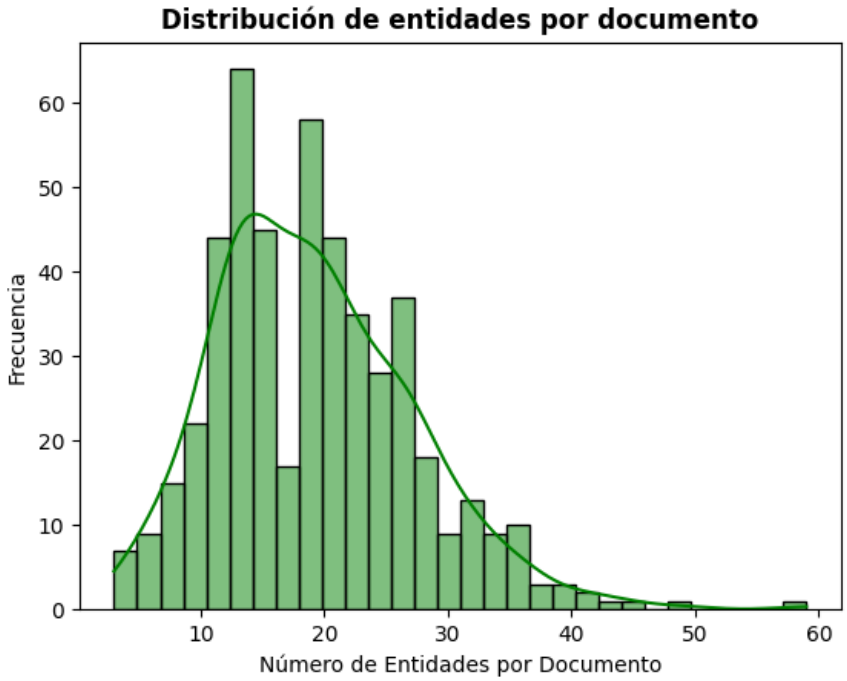

Figura 7.4: Gráfica de distribución de número de entidades por caso clínico

La siguiente gráfica también es bastante interesante, pues es un histogra-

ma que muestra la distribución del número de entidades que aparecen por documento, es decir, por caso clínico. El eje x corresponde al número de entidades por documento, mientras que el eje y corresponde con la frecuencia con la que aparecen. Cada barra, representa un intervalo de número de entidades por caso. La línea verde suavizada se conoce como curva de densidad y proporciona una estimación visual de la distribución de los datos.

Viendo la gráfica, podemos notar que la mayor parte de casos clínicos presentan entre diez y treinta entidades aproximadamente, encontrando un pico en torno a las 15-20. Además, la distribución posee una cola a la derecha, es decir, existen documentos que tienen un número relativamente alto de entidades, pero no predominan. En cuanto a la curva de densidad, sugiere una distribución unimodal, pues presenta un único pico.

Tras esto, podemos concluir que la gráfica indica que existe uniformidad, pues existe una tendencia entre diez y treinta entidades por documento. Así, los datos no están excesivamente desbalanceados, lo cual ayudará al entrenamiento del modelo y la generalización del mismo ya que evitamos que haya ruido. Además el hecho de que haya una cola también podemos considerarlo positivo, ya que estamos agregando cierta variedad que obliga al modelo a exponerse a nuevos escenarios, lo cual, de nuevo, también ayudará a la generalización.

Listing 7.8: Generación de gráfica de distribución de longitudes de entidades

```
entity_lengths = data['span'].apply(len)
sns.histplot(entity_lengths, bins=30, kde=True, color='
    green')
plt.xlabel('Longitud de las entidades')
plt.ylabel('Frecuencia')
plt.title('Distribucion de longitudes de las entidades',
    fontweight='bold')
plt.show()
```

Continuamos con otra gráfica de distribución, aunque en este caso de longitudes de entidades [Figura 7.5. El eje x representa la longitud de las entidades en términos de número de caracteres; mientras que el eje y representa la frecuencia con la que aparecen. Al igual que en el caso anterior, la línea verde corresponde a la curva de densidad. En cuanto a las barras, cada una representa un intervalo de cinco caracteres.

Gracias a ella, podemos saber que la mayor parte de entidades presentan una corta longitud, con un pico bastante pronunciado alrededor de uno y diez caracteres. Por tanto, sugiere que las entidades mayoritariamente son o bien palabras individuales, o bien combinaciones de palabras cortas. En cuanto a la cola, es bastante larga hacia la derecha, pues la distribución es asimétrica, indicando que aunque menos común, hay algunas entidades

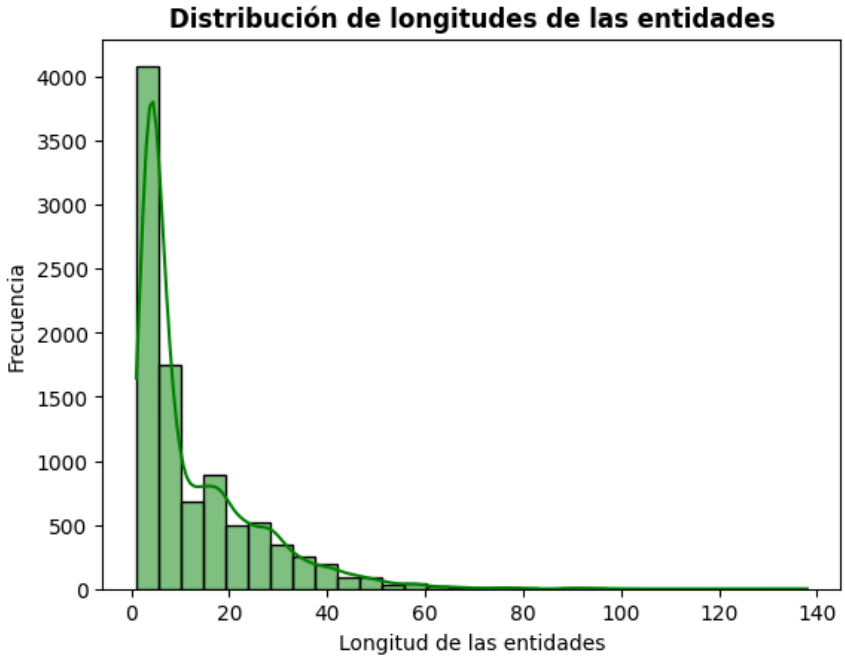

Figura 7.5: Gráfica de distribución de longitudes de entidades

con longitudes significativamente mayores, que podrían incluso a llegar a ser frases completas. Esto aumenta la complejidad y aunque es verdad que añade variabilidad al modelo y puede ayudarle a aprender, también significa que habrá que tenerlas en cuenta durante el preprocesamiento de datos y su futura tokenización.

Listing 7.9: Generación de gráfica de correlación de entidades

```
entities_per_doc = data.groupby('filename')['span'].count()
sns.histplot(entities_per_doc, bins=30, kde=True, color='green')
plt.xlabel('Numero de Entidades por Documento')
plt.ylabel('Frecuencia')
plt.title('Distribucion de entidades por documento', fontweight='bold')
plt.show()
```

También es una práctica común en AA analizar la matriz de correlación entre características. En nuestro caso, tipos de entidades. En ella, se muestra la relación que tienen unas con otras. Para ello, se emplea una matriz triangular.

Lo primero que destacamos en la gráfica [Figura 7.6] es su diagonal,

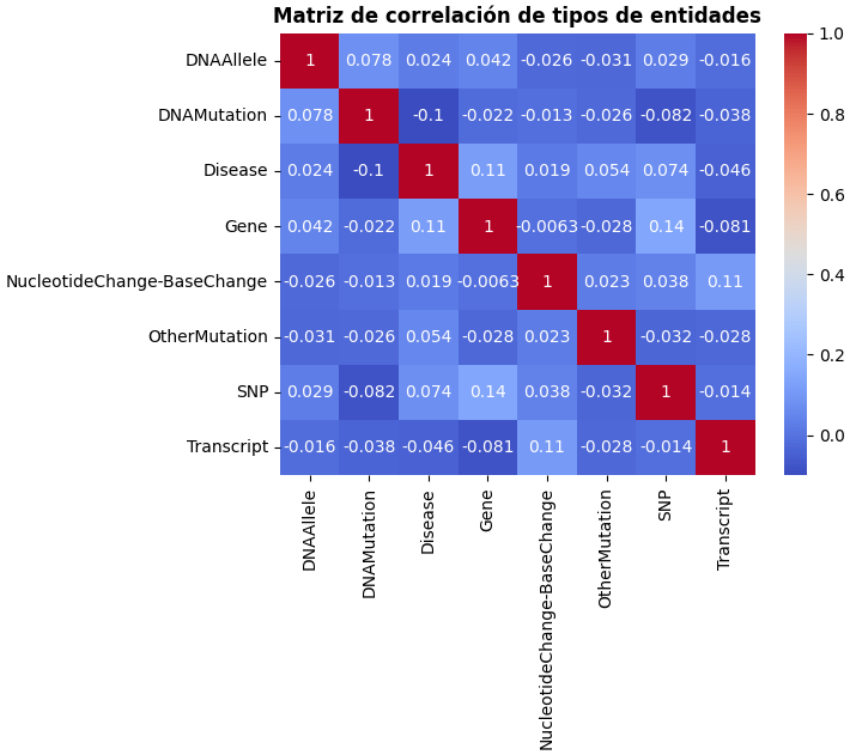

Figura 7.6: Gráfica de correlación entre tipos de entidades

con valor máximo 1. Esto se debe a que cada tipo de entidad presenta una correlación máxima consigo misma. Sin embargo, la relación de entidades con otras es bastante baja, dado que los valores son bastante cercanos a cero. De hecho, las mayores correlaciones se encuentran entre genes y bases nucleótidas y entre genes y enfermedades. Motivo que también nos ayuda a entender el porqué en la gráfica de frecuencias previa ambas etiquetas sean las que valores más altos presentan (a más enfermedades, más genes y viceversa). Otra característica importante que aparece es que hay valores negativos, lo que sugiere ya no solo que no haya relación entre ese par de categorías, sino que también hay ocasiones en las que una es excluyente de otra. Aún así, recordemos que algunos tipos son bastante escasos, de modo que no podríamos asegurar esto con certeza ya que no hay ejemplos suficientes para confirmarlo.

Por tanto, de esta gráfica podemos concluir que la mayoría de entidades NER del corpus aparecen de forma independiente unas de otras. Aunque sí que hay algunas relaciones específicas, como entre genes y enfermedades, coincidiendo en que ambos tipos son los más predominantes.

7.2. Prototipado del sistema

Es esta sección abordaremos el proceso de implementación y, cuando sea pertinente, de fine-tuning de diversos modelos. El objetivo es poder evaluar su desempeño en la identificación de entidades médicas en el corpus previamente mencionado.

Comenzaremos con modelos de menor a mayor complejidad, iniciando con el modelo GPT, seguido del modelo BERT, RoBERTa y, finalmente, GLiNER.

Es importante destacar que durante el prototipado se han llevado a cabo varias fases. Inicialmente, se procedió a identificar únicamente el tipo de entidad 'Disease' con el fin de familiarizarnos con los modelos. Por tanto, de cara a no mostrar información irrelevante para este estudio, nos centraremos directamente en el desempeño de los modelos con todas las categorías.

Recordemos que, paralelamente, estamos trabajando con Codalab, que a su vez ha pasado por dos fases. La primera consistió en emplear el conjunto de validación (archivos *dev*) como test, mientras que la segunda implicó un desarrollo estándar donde los archivos *train* y *dev* fueron usados durante el entrenamiento y el archivo *test* para la evaluación. Los resultados de mayor interés corresponden a esta segunda fase, por lo que se le dará especial atención en nuestro análisis. La primera fase será mencionada de forma más breve. Es relevante señalar que durante la primera fase aún no se tenía acceso a los modelos GPT, por lo que estos han sido empleados siguiendo las normas establecidas en la segunda.

7.2.1. Prototipado de GPT-3.5

Esta sección aborda la implementación del modelo GPT aplicado a NER, toda ella se encuentra dentro del archivo Prototipado_GPT.ipynb.

Así, lo primero que debemos hacer es instalar las bibliotecas necesarias, pues no vienen por defecto.

Listing 7.10: Instalación de bibliotecas

```
# Descarga e importacion de la API de OpenAI
!pip install -q openai==0.28
import openai
```

A continuación, montamos Google Drive como vimos en el apartado anterior y leemos los archivos necesarios. En este caso solo usaremos el archivo *test_text* pues GPT no precisa de etapas de entrenamiento como tal.

Listing 7.11: Instalación de bibliotecas

```
# Configuracion de la API de OpenAI
openai.api_type = 'azure'
openai.api_base = 'URL'
openai.api_version = '2023-05-15'
openai.api_key = 'Key'
```

El siguiente paso es la configuración de la API de OpenAI. De primera a última línea, especificamos que estamos utilizando la API de OpenAI proporcionada por Azure, definimos la URL base del punto de acceso (endpoint) para la API en Azure para poder realizar llamadas a esta, especificamos la versión de la API que queremos emplear, y autenticamos las solicitudes a la API a través de una clave privada a la cuenta de Azure vinculada. Esta configuración es la proporcionada por Fujitsu y por motivos de privacidad se ha ocultado la URL y la clave empleadas.

Debemos tener en cuenta que, como ya mencionamos, los modelos GPTs son modelos generativos, lo que implica que su propósito como tal no es el reconocimiento de entidades. Esto es realmente importante pues lo siguiente a establecer es el prompt que vamos a pasar al modelo. En él tenemos que considerar que debemos ser bastante concretos, indicando detalladamente la tarea a realizar, el contexto de la misma, el formato de salida y demás consideraciones.

Vamos a ir paso a paso, explicando brevemente los prompts que fueron empleados hasta llegar al usado finalmente. Para empezar, siempre debemos especificar el contexto:

"*Eres un modelo de inteligencia artificial especializado en el reconocimiento de entidades nombradas (NER) en el campo de la biomedicina.*"

Asimismo, debemos indicarle siempre que etiquetas queremos que detecte:

"*Debes identificar los siguientes tipos de entidades en el texto: Gene, Disease, DNAMutation, SNP, DNAAllele, NucleotideChange-BaseChange, OtherMutation, Transcript.*"

Por último, le informamos del formato de salida (cuando empleamos {} le estamos indicando que es una variable):

"*Los resultados deben mostrarse en el formato:* $\{tipo\ entidad\} \rightarrow \{entidad\}\backslash n$"

Si juntamos estas tres sentencias en un mismo prompt, obtenemos el más simple en donde especificamos los requisitos más importantes. Sin embargo, da cabida a muchos errores pues todavía es bastante ambiguo. Entre otros, el modelo añade más etiquetas de las especificadas, en caso de no encontrar entidades devuelve un texto totalmente distinto al formato de salida, no las clasifica por orden de aparición y en ocasiones ignora tildes y mayúsculas.

En este punto, además de intentar corregir dichos fallos, también aplicamos otra técnica vista durante el curso de Prompt Engineering. Esta consiste en mostrar ejemplos al modelo para que pueda entender mejor la tarea. Así, se procedió a añadir dos ejemplos por cada tipo de entidad en el prompt. En cuanto a las tres sentencias anteriores, se mejoraron:

"*Eres un modelo de inteligencia artificial especializado en el reconocimiento de entidades nombradas (NER) en el campo de la biomedicina. Debes identificar sola y únicamente los siguientes tipos de entidades en el texto: Gene, Disease, DNAMutation, SNP, DNAAllele, NucleotideChange-BaseChange, OtherMutation, Transcript. Cualquier otra etiqueta daría lugar a error. Los resultados, siempre y cuando se hayan detectado entidades, deben mostrarse exactamente en el formato:* $\{tipo\ entidad\} \rightarrow \{entidad\}\backslash n$. *Además, es muy importante que aparezcan por orden de aparición y escritas exactamente igual que en el texto (tildes, mayúsculas).*"

Tras esto, los resultados no podemos decir que fueran mejores. El hecho de haber añadido ejemplos hizo que el modelo entendiera que por cada tipo de entidad, se imprimieran todas las entidades correspondientes a esta en una misma línea. Cuando no detectaba entidades, seguía devolviendo un texto ajeno al formato especificado. Además, el modelo intentaba siempre encontrar todas las entidades en cada caso clínico. Sin embargo, sí que se consiguió que los tipos devueltos fueran siempre los especificados y que tuviera en cuenta las tildes y mayúsculas.

Así, tras diversas pruebas establecimos un prompt final que lejos de ser ideal, era bastante mejor que los anteriores. En él, eliminamos los ejemplos (menos es más) y fuimos más concretos en aquellos requisitos en los que el modelo aún presentaba fallos. Así, el prompt final fue el siguiente:

"*Eres un modelo de inteligencia artificial especializado en el reconocimiento de entidades nombradas (NER) en el campo de la biomedicina. Debes identificar sola y únicamente los siguientes tipos de entidades en el texto: Gene, Disease, DNAMutation, SNP, DNAAllele, NucleotideChange-BaseChange, OtherMutation, Transcript. Cualquier otra etiqueta daría lugar a error. Recuerda que no todas estas entidades necesariamente aparecerán en cada caso clínico, pero debes ser riguroso al clasificarlas según corresponda. Los resultados, siempre y cuando se hayan detectado entidades, deben mostrarse exactamente en el formato:* $\{tipo\ entidad\} \rightarrow \{entidad\}\backslash n$. *Escribe cada entidad encontrada en una línea distinta, aunque sean del mismo tipo. Además, es muy importante que aparezcan por orden de aparición y escritas exactamente igual que en el texto (tildes, mayúsculas). En caso de no encontrar entidades devuelve un string vacío: ' '.*"

Sabiendo esto, vamos a continuar con la implementación per se para luego discutir sus resultados finales en los apartados de experimentación.

Listing 7.12: Método para detectar entidades por caso clínico

```
def detect_entities(text):
    response = openai.ChatCompletion.create(
        engine='gpt-35-turbo-model',
        messages=[
            {'role': 'system', 'content': system_content
                },
            {'role': 'user', 'content': text}
        ]
    )
    return response['choices'][0]['message']['content']
```

Esta función se encarga de, dado un caso clínico (almacenado en el argumento *text*), devolver las entidades junto con su tipo detectadas. Para ello, vemos que usamos GPT-3.5 Turbo junto con dos roles, *system* indica el contexto del modelo, el prompt que acabamos de mencionar; mientras que *user* corresponde al texto con el caso clínico.

La salida es un diccionario producto de un JSON, formado por varias respuestas. Puesto que no tenemos ningún criterio para saber cuál es mejor, siempre escogemos el contenido de la primera.

El siguiente paso es crear un archivo TSV con el formato visto en la Tabla 3.2. En general, ya veremos que en futuros modelos este paso se separa en un archivo nuevo dedicado exclusivamente a la evaluación. Sin embargo, ya que estamos trabajando con el archivo *test_text.tsv* directamente, es más cómodo generar ahora el TSV y dejar el archivo de evaluación de GPT para gráficas del análisis de resultados.

Listing 7.13: Creación del archivo annotationGPT3.5Turbo.tsv

```
filename = '/content/drive/MyDrive/My_TFG/
    DatasetGenovardis/EvaluationTSV/annotationGPT3.5Turbo
    .tsv'
tags = 'pmid\tfilename\tlabel\toffset1\toffset2\tspan\n'
labels = ["Gene", "Disease", "DNAMutation", "SNP", "
    DNAAllele", "NucleotideChange-BaseChange", "
    OtherMutation", "Transcript"]

with open(filename, 'w') as file:
  # Escribimos cabeceras
  file.write(tags)
  # Por cada caso clinico...
  for i, clinic_case in data.iterrows():
    # Guardamos pmid, archivo en formato .ann y texto del
        caso
    pmid = clinic_case['pmid']
    ann = clinic_case['filename'].replace('.txt', '.ann')
```

```
13      text = clinic_case['text']
14      # Inicio y fin para cada entidad
15      offset1 = 0
16      offset2 = 0
17      set_of1 = set()
18      set_of2 = set()
19
20      # Detectamos entidades con nuestro modelo GPT
21      entities = detect_entities(text)
22      # Guardamos cada entidad individual en una lista
23      list_entitites = entities.split('\n')
24
25      # Por cada entidad...
26      for entity in list_entitites:
27        # Separamos el tipo de entidad de la entidad
28        entity_split = entity.split('-->')
29        # Si es una entidad vacia o el formato devuelto por
              el modelo es incorrecto, la ignoramos
30        if entity == '' or len(list_entitites) == 1 or len(
              entity_split) != 2:
31          continue
32        label, span = entity_split
33        span = span.strip()
34        label = label.strip()
35        # Si ha devuelto una etiqueta incorrecta, ignoramos
              la entidad
36        if label not in labels:
37          continue
38        # Buscamos las apariciones de la entidad en el
              texto, biblioteca re
39        pattern = rf'\b{re.escape(span)}\b'
40        matches = re.finditer(pattern, text)
41
42        # Por cada aparicion...
43        for m in matches:
44          # Comprobamos que no corresponda con otra entidad
                previa
45          if m.start() not in set_of1 and m.end() not in
                set_of2:
46            offset1 = m.start()
47            offset2 = m.end()
48            set_of1.add(offset1)
49            set_of2.add(offset2)
50            break
51        # La escribimos en el archivo TSV
52        file.write(f"{pmid}\t{ann}\t{label}\t{offset1}\t{
              offset2}\t{span}\n")
```

Si nos fijamos en el código, se realizan bastantes comprobaciones de la respuesta devuelta por el modelo. Esto se debe a que a pesar de haber mejorado el prompt, las respuestas no siempre son adecuadas. Así, debemos filtrar aquellos casos donde se devuelva un tipo de entidad distinto o el formato no corresponda al indicado.

En cuanto a la localización de una entidad en el caso clínico (posición de caracteres de inicio y fin), se ha empleado la biblioteca re (Regular Expressions), la cual nos facilita mucho el proceso de búsqueda. El modelo tampoco devuelve las entidades en orden, por lo que no tendría sentido iterar de inicio a fin de cada caso clínico comprobando la posición de inicio y fin de cada entidad detectada. Con esta biblioteca, obtenemos un iterador con todas las apariciones de una entidad, incluyendo su posicion de inicio (offset1) y fin (offset2), de modo que simplemente debemos tener en cuenta que no correspondan con otra entidad detectada previamente. Así, no importa el orden devuelto por el modelo GPT.

Cabe destacar que también se contempló la opción de que el modelo también devolviera las posiciones de inicio y fin de cada entidad en el caso clínico. Sin embargo, como se esperaba, los resultados fueron bastante malos.

7.2.2. Prototipado de RoBERTa

En capítulos anteriores ya mencionamos que RoBERTa es un modelo tipo BERT que ha obtenido resultados bastantes buenos cuando ha sido aplicado al dominio NER. Además, recordemos que el modelo a emplear fue preentrenado por el PlanTL sobre un amplio conjunto de textos biomédicos. Sin embargo, este modelo en concreto se centra en Fill-Mask, pudiendo también ejercer sobre NER, pero deberemos hacer un fine-tuning si queremos obtener resultados adecuados.

Esto quiere decir que vamos a distinguir dos etapas durante la implementación, el preprocesado y el entrenamiento del modelo. Lo primero siempre es descargar las bibliotecas a emplear que no vienen por defecto en Colab.

Listing 7.14: Instalación de bibliotecas

```
# Descarga e importacion las bibliotecas de Hugging Face para NER
!pip install transformers datasets evaluate seqeval accelerate

from datasets import Dataset, DatasetDict
from transformers import AutoTokenizer
from transformers import DataCollatorForTokenClassification
import evaluate
```

```
from transformers import AutoModelForTokenClassification,
    TrainingArguments, Trainer
from transformers import pipeline
from transformers import BasicTokenizer
```

Iremos explicando los métodos concretos importados conforme los usemos. El siguiente paso es montar el entorno Drive y leer los archivos a emplear. En este caso, sí que necesitamos leer tanto los archivos de train como los archivos dev.

Previamente mencionamos que este modelo siguió dos etapas debido al uso de Codalab. En una usamos los archivos dev como test (archivo Prototipado_RoBERTa_DevTest.ipynb) y en la otra los empleamos como validación junto a los train (archivo Prototipado_RoBERTa_DevVal.ipynb). A nivel de implementación los cambios son muy sutiles, por lo que se resaltarán cuando sea necesario.

Preprocesado

El preprocesado es una fase clave en aprendizaje automático, con él nos referimos al conjunto de pasos que se realizan antes de entrenar un modelo sobre nuestros datos. Así, hay que asegurar que los datos se encuentren en el formato adecuado, corregir desbalanceos de clases, manejar entidades atípicas, entre otras técnicas.

Durante la fase de exploración de los datos hemos identificado varios desafíos significativos, destacando principalmente el desequilibrio de clases. Este desbalanceo, aunque común en muchos conjuntos de datos, presenta una dificultad particular en nuestro contexto, dado que trabajamos con datos técnicos donde la generación de ejemplos adicionales o la reducción de muestras no es una opción viable. Es por este motivo que se decidió escoger un modelo RoBERTa previamente entrenado con datos biomédicos, los cuales se ha considerado que podrían ser capaces de suplir estas carencias hasta cierto punto.

En base a esto, nuestro preprocesamiento se va a basar en adaptar los datos a un formato compatible con nuestro modelo. El formato en concreto que deben presentar es el siguiente:

Listing 7.15: Formato de modelos BERT. Ejemplo de Hugging Face.

```
{'id': '0',
 'ner_tags': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    7, 8, 8, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0],
 'tokens': ['@paulwalk', 'It', "'s", 'the', 'view', 'from
    ', 'where', 'I', "'m", 'living', 'for', 'two', '
    weeks', '.', 'Empire', 'State', 'Building', '=', '
```

```
    ESB', '.', 'Pretty', 'bad', 'storm', 'here', 'last',
     'evening', '.']
}
```

En general, la mayoría de modelos BERT necesitan que cada instancia presente este formato cuando son aplicados a NER. Este ejemplo de Hugging Face es bastante claro y representa un diccionario con tres claves, la primera es ID, un identificador para la instancia que en nuestro caso corresponderá con el pmid de cada caso; los NER tags, que son dígitos correspondientes al tipo de entidad detectada para cada token; y los tokens, las diversas palabras correspondientes a una sentencia separadas para su posterior análisis.

Vamos a centrarnos en la segunda clave, los NER tags. Intuitivamente, podríamos suponer que simplemente basta con asignar un dígito por cada tipo de entidad a detectar. En nuestro caso, con ocho serían suficientes. Sin embargo, la realidad es un poco más compleja.

En muchos modelos, tipo BERT entre ellos, se adopta un enfoque donde se asignan dos dígitos por entidad, además del dígito cero, que se reserva para palabras que no corresponden a ningún tipo. Por ejemplo, consideremos la etiqueta 'Disease' y supongamos que le asignamos los dígitos 1 y 2. Ahora, si encontramos una entidad como 'cáncer de cerebro', que consta de más de una palabra, necesitamos una manera de indicar que todas estas palabras están relacionadas y forman parte de la misma entidad. Por lo tanto, la asignación de etiquetas para este ejemplo podría representarse como un vector [1, 2, 2], donde el primer número indica que se ha detectado una entidad del tipo 'Disease', y los siguientes números indican que las palabras subsiguientes están relacionadas y son dependientes de esta. Si asignáramos el mismo número para todas ellas, el modelo interpretaría no solo que las tres son entidades independientes, sino que partículas como 'de' por sí misma correspondería a una enfermedad. Para facilitar más este proceso, por convención la etiqueta se divide varios tipos (B, begin e I, inside), 'B-Disease' y 'I-Disease', correspondientes a los dígitos 1 y 2, respectivamente, en este caso. Este esquema es conocido como BIO y es el que aplicaremos para todos los tipos de entidades.

Gracias a este enfoque, el modelo es capaz de capturar tanto la presencia de una entidad como su estructura jerárquica y las relaciones entre las palabras que la forman, consiguiendo una mejor compresión.

Listing 7.16: Diccionario dígito-tipo de entidad

```
idlabel = {
    "Disease": 1,
    "Gene": 3,
    "DNAMutation": 5,
    "SNP": 7,
```

```
    "DNAAllele": 9,
    "NucleotideChange-BaseChange": 11,
    "OtherMutation": 13,
    "Transcript": 15,
}
```

Con este diccionario, establecemos los dígitos de begin (B) correspondientes a cada tipo de entidad. Para los casos de inside (I), simplemente será necesario sumar uno al anterior.

Listing 7.17: Método para adaptar el formato a RoBERTa

```
# Tokenizador basico
basic_tokenizer = BasicTokenizer()
# Dataframe de instancias formateadas
data_df = pd.DataFrame(columns=['id', 'tokens', 'ner_tags'])

# Dada una instancia con informacion sobre Pmid, Filename y Text, la convierte a un formato compatible con modelos BERT
#   Argumentos:
#   - example (dic): Instancia.
#   - type_set (str): Conjunto de datos a tratar, train o dev.
#   Return:
#   - Dataframe con una fila mas de la instancia en formato BERT
def tokenize_and_set_ids(example, type_set='train'):
  global data_df
  # Lista para almacenar indices de entidades encontradas
  indices = []

  pmid = example['pmid']
  text = example['text']
  # Tokenizamos el texto
  tokens = np.array(basic_tokenizer.tokenize(text))
  # NER tags, por defecto todos cero
  tags = np.zeros(len(tokens), dtype=int)

  # Leemos el tipo de Dataset indicado (train o dev) y lo
  # odenamos en base a los valores de offset1
  if type_set == 'dev':
    sort_df = dev_annotation[dev_annotation['pmid'] == pmid].sort_values(by='offset1')
  else:
    sort_df = train_annotation[train_annotation['pmid'] == pmid].sort_values(by='offset1')

```

```
31    # Guardamos los valores de las etiquetas y las
          entidades
32    labels = sort_df['label'].values.astype(str)
33    spans = sort_df['span'].values.astype(str)
34
35    index = 0
36    # Por cada par de entiqueta-entidad
37    for l, d in zip(labels, spans):
38      # Tokenizamos la entidad
39      span_split = np.array(basic_tokenizer.tokenize(d))
40      # Obtenermos los indices donde aparece dicha entidad
41      index = np.where(tokens == span_split[0])[0]
42      found = False
43      # Si tiene tamanio 1...
44      if len(d) == 1:
45        i = 0
46        # Mientras que no la encotremos...
47        while not found and i < len(index):
48          # Si la posicion actual no corresponde a otra
              entidad previa...
49          if index[i] not in indices and (len(indices) == 0
                or index[i] > max(indices)):
50            indices.append(index[i])
51            found = True
52            # Actualizamos el NER tag de la posicion
                correspondiente al indice
53            tags[index[i]] = idlabel[l]
54          i += 1
55      # Si tiene un tamanio mayor que 1...
56      else:
57        k = 0
58        # Mientras que no la encontremos...
59        while not found and k < len(index):
60          i = index[k]
61          # Si la posicion actual no corresponde a otra
                entidad previa...
62          if i not in indices and (len(indices) == 0 or i >
                max(indices)):
63            # Si todos los elementos de la entidad
                  coinciden...
64            if np.array_equal(tokens[i:i+len(span_split)],
                  span_split):
65              index_range = np.arange(i, i+len(span_split))
66              indices.extend(index_range)
67              found = True
68              # Actualizamos los NER tags de las posiciones
                    correspondientes
69              tags[index_range[0]] = idlabel[l]
70              tags[list(index_range[1:])] = idlabel[l] + 1
```

```
71            k += 1
72
73    # Diccionario con la instancia formateada para BERT
74    result = {'id': str(pmid), 'tokens':tokens, 'ner_tags':
          tags}
75
76    # Aniadimos una nueva fila al Dataframe con la entidad
77    data_df.loc[len(data_df)] = result
78
79    return data_df
```

En resumen, el funcionamiento del código anterior es relativamente simple. Dada una instancia de un caso clínico, tomamos su texto y lo dividimos en palabras. Para ello, usamos el método BasicTokenizer de Transformers. No es un tokenizador formal que podamos usar directamente en nuestros modelos, pero para lo que nos incumbe es ideal.

Más concretamente, divide cada elemento del texto, incluyendo incluso los signos de puntuación. Este detalle es crucial, ya que otros métodos de manipulación de cadenas, como split, no los tienen en cuenta. Por ejemplo, si encontramos palabras seguidas de una coma, split devolvería una lista con un único elemento, como ['tumor,'], mientras que BasicTokenizer devolvería una lista donde la coma se trata como un token separado, como ['tumor', ','. Esto resulta más conveniente, ya que queremos que nuestro modelo pueda distinguir la palabra "tumor"por sí sola.

Tras ello, obtenemos las entidades correspondientes a cada caso clínico de los archivos annotation a partir del pmid. De ellos, únicamente nos interesan las etiquetas y las entidades asociadas. Por último, iteramos por cada entidad, previamente ordenadas a partir del offset1 (por lo tanto, por orden de aparición en el caso), las dividimos en palabras y buscamos cuál es su índice en el texto previamente dividido. Una vez encontrado, asignamos los NER tags correspondientes en función del tipo de entidad, teniendo en cuenta que si hay más de una palabra deberemos de sumar uno a los tokens correspondientes ya que serán del tipo inside, I.

Listing 7.18: Mapeado de la función sobre todas las instancias de los archivos de entrada

```
train_text.apply(tokenize_and_set_ids, axis=1)
dev_text.apply(lambda row: tokenize_and_set_ids(row,
    type_set='dev'), axis=1)
}
```

Este paso consiste en lo que se conoce como mapeado, es decir, aplicar la función que acabamos de implementar sobre todas las instancias de los archivos de entrada. Si, además, queremos añadir el archivo dev al con-

junto de entrenamiento, la maperíamos también sobre él. De este modo, obtendríamos un dataframe final, que hemos llamado *data_df*, que incluye todas las instancias ya formateadas y listas para ser usadas.

Listing 7.19: Conjuntos de entrenamiento y validación

```
# Si usamos archivos dev durante el entrenamiento
len_dev_text = len(dev_text)

data_train = Dataset.from_pandas(data_df.head(len(data_df) - len_dev_text))
data_val = Dataset.from_pandas(data_df.tail(len_dev_text))

data = DatasetDict({
    'train': data_train,
    'test': data_val
})

# Si no usamos archivos dev durante el entrenamiento
data = Dataset.from_pandas(data_df)
data = data.train_test_split(test_size=0.1)
}
```

El siguiente paso del preprocesado es dividir los archivos en un conjunto de entrenamiento y validación. Si no se emplea dev durante el entrenamiento, se ha procedido a tomar un 10 % de los datos iniciales para validación, pues es una práctica común y recomendable, más aún en nuestro caso donde no disponemos de un corpus realmente extenso. Por el contrario, cuando usamos dev para entrenamiento, usamos este para validación. La competición de Codalab ya nos lo proporciona para usarlo como tal ya que es una manera de asegurarnos que el modelo no presente desbalanceo de clases entre conjuntos. Lo cual podría ocurrir cuando lo dividimos aleatoriamente.

Destacamos el empleo de estructuras de datos como Dataset y DatasetDict, ambas de Hugging Face. Estas nos aseguran una carga y trabajo eficiente (bastante relevante) de los datos a la hora de emplear modelos Transformers, pues poseen compatibilidad directa con ellos. Además, son fáciles de usar, pues podemos obtenerlas a partir de nuestros dataframes previos, e internamente usan muchos métodos de Scikit-learn, como el empleado *train_test_split*.

Listing 7.20: Tokenización

```
# Cargamos tokenizer de RoBERTa
tokenizer = AutoTokenizer.from_pretrained("BSC-TeMU/roberta-base-biomedical-es")
```

```
3
4  # Metodo para arreglar la discordancia entre entidades y
       etiquetas tras la tokenizacion
5  #    Argumentos:
6  #    - examples (dic): Instancia de un caso clinico.
7  #    Return:
8  #    - Instancia tokenizada para RoBERTa
9  def tokenize_and_align_labels(examples):
10     # Tokenizamos la instancia dada como argumento
11     tokenized_inputs = tokenizer(examples["tokens"],
           truncation=True, is_split_into_words=True)
12
13     labels = []
14     # Por cada uno de sus tokens...
15     for i, label in enumerate(examples[f"ner_tags"]):
16         # Mapeamos los tokens a su respectiva palabra
17         word_ids = tokenized_inputs.word_ids(batch_index=
               i)
18         previous_word_idx = None
19         label_ids = []
20         for word_idx in word_ids:
21             # Establecemos los tokens especiales a -100
22             if word_idx is None:
23                 label_ids.append(-100)
24             elif word_idx != previous_word_idx:
25                 # Solo etiquetamos el primer token de la
                       palabra
26                 label_ids.append(label[word_idx])
27             else:
28                 label_ids.append(-100)
29             previous_word_idx = word_idx
30         labels.append(label_ids)
31
32     tokenized_inputs["labels"] = labels
33     return tokenized_inputs
34
35 tokenized_data = data.map(tokenize_and_align_labels,
       batched=True)
```

El siguiente paso es realizar una tokenización. No confundir con el anterior donde usamos el BasicTokenizar, ya que realmente lo que hicimos fue dividir el texto en palabras. La tokenización propiamente dicha depende en gran medida de cada modelo. En nuestro caso, RoBERTa lo hace del siguiente modo, imaginemos que tenemos una sentencia que dice así: *Análisis de mutaciones en glioblastoma y meduloblastoma*. Si aplicamos su tokenizador, devolvería una lista con los siguientes tokens: ['<s>', '#Análisis', '#de', '#mutaciones', '#en', '#glio', 'blastoma', '#y', '#med', 'ulo', 'blastoma', '</s>']. Se aprecia que el tokenizador divide palabras en subpalabras y, pa-

ra guardar constancia de ello, emplea los símbolos # siempre al comienzo de cada una. Además, emplea tokens especiales que indican el inicio y fin de la sentencia a analizar.

Cuando hablamos de la estructura Transformer ya vimos la vital importancia de la tokenización. Sin embargo, al proceder de este modo, nos encontramos que tras realizar este paso, hay una discrepancia entre las palabras y los NER tags que asignamos con anterioridad. La función implementada soluciona este problema. Esta tokeniza cada palabra de cada caso clínico y crea una nueva lista en donde por cada token, si corresponde a un token especial o es un subtoken de otra palabra se añade el valor -100. En caso contrario, es decir, es el primer token de una palabra, asignamos el valor del NER tag calculado al comienzo. Así, en el ejemplo anterior únicamente las palabras que vienen precedidas por # presentarían un valor correspondiente al NER tag, mientras que las demás serían -100.

El valor -100 es comúnmente utilizado como marcador especial durante el entrenamiento en modelos NER. Podemos decir que su función es actuar como una máscara de pérdida, es decir, indica al modelo que no se deben tener en cuenta estos tokens durante el cálculo de la pérdida.

Ya que nuestros datos son de tipo DatasetDict, podemos mapear esta función directamente sobre todos ellos y así obtener directamente nuestro conjunto de datos tokenizado.

Por último, mencionar que para obtener el tokenizador empleamos la estructura AutoTokenizer junto con el método *from_pretrained*. En él indicamos el modelo con el que vamos a trabajar que, en nuestro caso corresponde al *BSC-TeMU/roberta-base-biomedical-es* del PlanTL.

Entrenamiento

En este punto ya poseemos un conjunto de datos listo para poder ser usado durante el entrenamiento en el fine-tuning. Así, lo que resta es configurar el modelo antes de que comience el aprendizaje.

Listing 7.21: Método para el cálculo de métricas

```
label_list = [
    "O",
    "B-Disease", "I-Disease",
    "B-Gene", "I-Gene",
    "B-DNAMutation", "I-DNAMutation",
    "B-SNP", "I-SNP",
    "B-DNAAllele", "I-DNAAllele",
    "B-NucleotideChange-BaseChange", "I-NucleotideChange-
       BaseChange",
```

```
    "B-OtherMutation", "I-OtherMutation",
    "B-Transcript", "I-Transcript",
]
# Funcion seqeval para calculo de metricas
seqeval = evaluate.load("seqeval")

# Metodo que calcula las metricas de evaluacion de nuestro modelo
#   Argumentos:
#   - p (tupla): Tupla con las predicciones del modelo y etiquetas verdaderas
#   Return:
#   - Diccionario con las metricas
def compute_metrics(p):
    # Desempaqueta las predicciones y etiquetas verdaderas
    predictions, labels = p
    # Decodifica las predicciones
    predictions = np.argmax(predictions, axis=2)

    # Extrae las etiquetas verdaderas y predichas, ignorando aquellas con valor -100
    true_predictions = [
        [label_list[p] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]
    true_labels = [
        [label_list[l] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]

    # Calcula metricas de evaluacion con seqeval
    results = seqeval.compute(predictions=true_predictions, references=true_labels)

    return {
        "precision": results["overall_precision"],
        "recall": results["overall_recall"],
        "f1": results["overall_f1"],
        "accuracy": results["overall_accuracy"],
    }
}
```

Primero, definimos un método que calcule las métricas de evaluación del desempeño del modelo. Este devuelve cuatro métricas: precisión, recall, F1-score y exactitud. Esta última no es tan habitual en NER como el resto,

pero merecerá la pena comentarla durante la evaluación del modelo. De ahí que también queramos mostrarla.

Listing 7.22: Configuración del modelo

```
id2label = {
    0: "O",
    1: "B-disease", 2: "I-disease",
    3: "B-gene", 4: "I-gene",
    5: "B-DNAMutation", 6: "I-DNAMutation",
    7: "B-SNP", 8: "I-SNP",
    9: "B-DNAAllele", 10: "I-DNAAllele",
    11: "B-NucleotideChange-BaseChange", 12: "I-
       NucleotideChange-BaseChange",
    13: "B-OtherMutation", 14: "I-OtherMutation",
    15: "B-Transcript", 16: "I-Transcript",
}
label2id = {
    "O": 0,
    "B-disease": 1, "I-disease": 2,
    "B-gene": 3, "I-gene": 4,
    "B-DNAMutation": 5, "I-DNAMutation": 6,
    "B-SNP": 7, "I-SNP": 8,
    "B-DNAAllele": 9, "I-DNAAllele": 10,
    "B-NucleotideChange-BaseChange": 11, "I-
       NucleotideChange-BaseChange": 12,
    "B-OtherMutation": 13, "I-OtherMutation": 14,
    "B-Transcript": 15, "I-Transcript": 16,
}

model = AutoModelForTokenClassification.from_pretrained(
    "BSC-TeMU/roberta-base-biomedical-es", num_labels=17,
        id2label=id2label, label2id=label2id
)

data_collator = DataCollatorForTokenClassification(
   tokenizer=tokenizer)

training_args = TrainingArguments(
    output_dir="/content/drive/MyDrive/My_TFG/
       DatasetGenovardis/logs/RoBERTa",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=20,
    weight_decay=0.01,
    eval_strategy="epoch",
    save_strategy="no"
)
```

```
40
41 trainer = Trainer(
42     model=model,
43     args=training_args,
44     train_dataset=tokenized_data["train"],
45     eval_dataset=tokenized_data["test"],
46     tokenizer=tokenizer,
47     data_collator=data_collator,
48     compute_metrics=compute_metrics,
49 )
50
51 trainer.train()
52 trainer.save_model("/content/drive/MyDrive/My_TFG/
    DatasetGenovardis/logs/RoBERTa/model_RoBERTa_Dev20")
```

El último paso es entrenar el modelo. Para ello, Hugging Face proporciona en la biblioteca Transformers varias estructuras de datos específicas, TrainingArguments y Trainer. Además, previamente debemos cargar el modelo a partir de AutoModelForTokenClassification con el método *from_pretrained* de manera similar a como cargamos el tokenizer previamente e indicando la cantidad de etiquetas y que índice corresponde a cada una y vicerversa.

También hemos empleado DataCollatorForTokenClassification, una clase muy usada en tareas NLP y sobre todo en NER. Su función principal es preparar los datos para el entrenamiento, más allá de comprobar que la tokenización actual sea correcta, también realiza tareas de padding[2], batching[3] y optimizaciones del rendimiento.

A continuación, en la estructura TrainingArguments, indicamos los hiperparámetros que vamos a emplear durante el entrenamiento del modelo:

- **output_dir**: Directorio donde guardar el modelo en caso de ser necesario.
- **learning_rate**: Tasa de aprendizaje, la cual controla los pasos que se toman durante el proceso de optimización. En nuestro caso hemos escodigo un valor de 2^{-5} ya que es comúnmente usada en modelos preentrenados de RoBERTa y suele funcionar bien en tareas NER.
- **per_device_train_batch_size**: El tamaño de los batches a emplear por nuestro modelo. En nuestro caso nos hemos decantado por 16 ya no solo por ser comúnmente utilizado, sino también porque Colab presenta limitaciones de GPU y batches más grandes consumirían más

[2]El *padding* se encarga de que todas las secuencias de entrada tengan la misma longitud. Por defecto, se toma como longitud de referencia la sentencia más larga.

[3]Los *batches* en aprendizaje automático son agrupaciones de datos, lotes, que facilitan y mejoran mucho el entrenamiento de modelos.

memoria. Hay que tener en cuenta que si empleamos batches muy pequeños favoreceríamos que nuestro modelo tienda a overfitting.

- **num_train_epochs**: Es el número de épocas[4] de entrenamiento. Hemos elegido 20 pues ya veremos en futuros apartados que el tiempo empleado por época es relativamente corto y puesto que no tenemos un corpus realmente extenso, se ha considerado que este número de épocas es razonable para evaluar la evolución del modelo. El porqué de 20 épocas concretamente es algo arbitrario, pues el objetivo en sí es ver hasta que punto nuestro modelo puede aprender. De modo que una cantidad algo más baja o más alta también habría sido válida.
- **weight_decay**: Es una técnica de regularización para evitar el *overfitting*. El valor de 0.01 es comúnmente utilizado y en nuestro caso ha funcionado correctamente, como ya veremos.
- **eval_strategy**: La establecemos a 'epoch' ya que queremos que se realice una evaluación con el conjunto de validación en cada época. Gracias a esto podremos tener más información de la evolución del aprendizaje del modelo.
- **save_strategy**: La establecemos a 'no' ya que no nos interesa que se guarde un modelo en cada época, pues supondría tener 20 modelos y consumiría gran cantidad de memoria.

Esta estructura con los hiperparámetros junto con el modelo, los conjuntos de entrenamiento y validación tokenizados, el tokenizador, el data collator y la función para el cálculo de métricas son dadas como argumentos a la estructura Trainer.

Con ella, y el método *train*, podremos entrenar al modelo. Para guardarlo empleamos el método *save_model* indicando la ruta y el nombre que le queremos dar.

Evaluación

Para evaluar el modelo RoBERTa entrenado previamente (archivo EvaluacionRoBERTa.ipynb), primero necesitamos montar Google Drive y leer el archivo *test_text* o *dev_text* según corresponda.

Listing 7.23: Instanciar modelo

```
classifier = pipeline(
    "ner",
```

[4]Cuando hablamos de épocas en aprendizaje automático, nos referimos a cuántas veces el modelo pasará por el conjunto de entrenamiento completo.

```
    model="/content/drive/MyDrive/My_TFG/
        DatasetGenovardis/logs/RoBERTa/
        model_RoBERTa_Dev20",
    aggregation_strategy="simple"
)
```

El siguiente paso es obtener una instancia de nuestro modelo. Para ello, podemos usar la estructura Pipeline, de nuevo, de la biblioteca Transformers. En ella indicamos que queremos crear un pipeline para el reconocimiento de entidades, seguida de la ruta del modelo a emplear y de la opción *aggregation_strategy* en modo *simple*. Esta última se encarga de combinar todos aquellos tokens consecutivos para una misma entidad detectada. En otras palabras, es el proceso contrario de la tokenización que vimos previamente en donde partimos de una palabra, ['#med', 'ulo', 'blastoma'] donde el primer elemento sería de tipo 'B-Disease' y los demás 'I-Disease', a obtener un único token, 'meduloblastoma'.

Listing 7.24: Realizar predicciones

```
predictions = classifier(text)

# Que devolveria una lista de este tipo
[{'entity_group': 'disease',
  'score': 0.967987,
  'word': 'Sindrome de Gorlin',
  'start': 11,
  'end': 29},
  ...
]
```

Una vez tenemos el clasificador, simplemente nos queda realizar predicciones y almacenar los resultados en un archivo TSV en formato similar al de la Tabla 3.2. Para realizar una predicción, debemos pasar el caso clínico concreto a nuestro modelo y devolverá un vector con todas las entidades detectadas e información como el score, comienzo y fin de la entidad en el texto.

El score es una medida que indica la confianza con la que un modelo ha identificado una entidad para una categoría concreta.

La creación del archivo TSV a partir de este modelo es realmente similar a como lo implementamos para el modelo GPT [Listing 7.13], por lo que obviaremos su código para evitar redundancia. Simplemente, hay que tener en cuenta que solo tomamos aquellas entidades cuyo score sea mayor que 0.5 (50 %) ya que, en cierto sentido, aceptar entidades con un score menor es equiparable a tomar una decisión al azar entre clasificar dicha entidad para un tipo dado o no. Tampoco se ha considerado tomar un valor más

alto ya que hay que ser conscientes que el tipo de entidades que estamos considerando poseen cierta dificultad, pues son muy concretas y podríamos estar limitando demasiado al modelo.

7.2.3. Prototipado de GLiNER

Ya se habló previamente que GLiNER es una de las estructuras más recientes dedicadas exclusivamente al reconocimiento de entidades. Actualmente su enfoque innovador y su desempeño sobresaliente lo sitúan potencialmente como uno de los modelos más competitivos en este campo, con amplias perspectivas de desarrollo futuro.

Sin embargo, el proceso de fine-tuning de GLiNER ha presentado ciertas complicaciones debido a la escasez de información disponible en línea sobre este modelo. Aunque se ha obtenido cierto conocimiento a través de su directorio GitHub[5], la información sigue siendo limitada en comparación con otros modelos establecidos. Sin embargo, es crucial destacar que a finales de abril de 2024, el proceso de fine-tuning de GLiNER experimentó una actualización significativa que resultó en mejoras sustanciales en los resultados.

Para esa fecha, ya se había llevado a cabo un proceso de fine-tuning con la versión anterior del modelo, lo que implicó que las implementaciones existentes sufrieran cambios importantes con el objetivo de mantener este proyecto lo más actualizado posible para la fecha de su entrega. Por ello, se presentarán ambas versiones pero haremos más énfasis en la más actual.

Por otro lado, el fine-tuning se ha llevado a cabo con dos versiones de GLiNER, la Small (archivo Prototipado_GLiNERSmall_DevTest.ipynb) y la Medium (archivo Prototipado_GLiNERMedium_DevTest.ipynb y el archivo Prototipado_GLiNERMedium_DevVal.ipynb). Desafortunadamente no se pudo con la Large por limitaciones de Colab. En concreto, la versión actualizada del fine-tuning la aplicaremos al modelo de tamaño Medium (archivo Prototipado_GLiNERMedium_Trainer.ipynb) pues fue el que nos devolvió mejores resultados.

Listing 7.25: Instalación de bibliotecas necesarias

```
!pip install gliner transformers

import numpy as np
import torch
import os
from google.colab import drive
import pandas as pd
from gliner import GLiNER
```

[5] https://github.com/urchade/GLiNER

```
from transformers import BasicTokenizer
from transformers import get_cosine_schedule_with_warmup
from tqdm import tqdm
```

Partiendo de este punto, lo primero que debemos hacer es descargar las bibliotecas necesarias. GLiNER posee la suya propia dentro de Hugging Face, por lo que no posee una integración directa con la biblioteca Transformers.

De nuevo, montamos Google Drive en el sistema de archivos y leemos los archivos para el entrenamiento, tanto los train como los dev si corresponde, junto con sus respectivos archivos annotation.

Preprocesado

Para el modelo GLiNER el preprocesado es distinto que con modelos BERT. En este caso, los datos se deben representar dentro de una lista compuesta por diccionarios. Cada uno deberá contener dos claves, la primera es *tokenized_text*, que contiene una lista con la división del texto de entrada (el caso clínico) en palabras; y la segunda, *ner*, que contiene una lista de listas. Cada una contiene tres elementos, el índice de inicio y fin de cada entidad dentro de *tokenized_text* y su tipo.

El código para realizar este preprocesado es trivial ya que es realmente similar al que empleamos para RoBERTa [Listing 7.17]. Simplemente tenemos que tomar las consideraciones de formato que acabamos de mencionar.

Resumidamente, volveríamos a dividir el texto de un caso clínico en sus palabras con BasicTokenizer, siendo este resultado el valor para la clave *tokenized_text*. A continuación, buscaríamos la posición en la que se encuentran todas las entidades para dicho caso a partir del archivo annotation correspondiente. Si la entidad a buscar tiene longitud unidad (solo es una palabra), su índice de inicio y fin será el mismo dentro del texto ya dividido; si, por el contrario, lo forman varias palabras, el índice de fin será distinto. De este modo, para cada entidad, guardaremos tantos estos índices como su etiqueta en una lista que almacenaremos en la clave *ner*.

Este proceso lo mapearíamos sobre el conjunto de datos de entrenamiento y así obtendríamos una lista formada por tantos diccionarios como casos clínicos vamos a emplear durante el aprendizaje.

En general, cuando hemos hablado de listas en apartados anteriores nos referíamos a listas Numpy pues son más eficientes y nos permiten hacer operaciones complejas de forma fácil. Sin embargo, para GLiNER, las listas que empleemos deben ser de tipo primitivo de Python, pues actualmente es lo que exige el modelo.

Con todo esto, ya tendríamos nuestro conjunto de datos listo para poder

ser usado durante el fine-tuning.

Entrenamiento

Mientras que la etapa de preprocesado era más sencilla de realizar en comparación con el modelo anterior, la de entrenamiento es algo más compleja. Al poseer GLiNER una biblioteca propia, ser tan actual y no presentar tan buena integración con otras bibliotecas de Hugging Face, este proceso se hace algo más rudimentario, pues nos debemos de ceñir a los métodos que esta nos proporciona.

Listing 7.26: Obtención del modelo

```
# Para GLiNER Medium
model = GLiNER.from_pretrained("urchade/gliner_medium-v2.1")
# Para GLiNER Small
model = GLiNER.from_pretrained("urchade/gliner_small-v2.1")
```

Lo primero que debemos de hacer es descargarnos el modelo que vamos a utilizar para el fine-tuning. En nuestro caso hemos empleado los modelos Small y Medium de Hugging Face en su versión 2.1, la más reciente, con 166M y 209M de parámetros, respectivamente.

A continuación, la implementación varía según que versión empleemos para el fine-tuning. Comenzaremos por la antigua.

Listing 7.27: Definición de hiperparámetros

```
from types import SimpleNamespace

config = SimpleNamespace(
    num_steps=20,
    train_batch_size=2,
    eval_every=1,
    save_directory="/content/drive/MyDrive/My_TFG/DatasetGenovardis/logs",
    lr_encoder=1e-5,
    lr_others=5e-5,
    freeze_token_rep=False,
    max_types=8,
    shuffle_types=True,
    random_drop=True,
    max_neg_type_ratio=1,
    max_len=700,
    warmup_ratio=0.1,
```

```
17 )
```

De manera similar a cuando usábamos la estructura TrainingArguments en RoBERTa, podemos emplear la clase SimpleNamespace para GLiNER. Esta clase es de Python como tal y proporciona una manera sencilla de crear objetos cuya finalidad es almacenar atributos.

El objeto creado contiene los hiperparámetros que emplearemos en nuestro modelo. Con el objetivo de garantizar una comparación equitativa en futuros análisis, se ha procurado asignar valores lo más justos posibles a los parámetros. El número de épocas, por ejemplo, es el mismo para ambos.

Sin embargo, parámetros como el tamaño del batch no pueden ser los mismos. Esto se debe a que los modelos GLiNER, ya sea por su arquitectura (capas, complejidad...), tamaño de los tensores[6], cantidad de operaciones u optimización del propio modelo, no permiten valores tan altos. En RoBERTa, un batch de tamaño 16 no supuso ningún problema; para GLiNER, el tamaño máximo que podemos emplear es 2, cuatro veces menor. En caso contrario, consume toda la GPU disponible de Colab.

Esto tiene varias implicaciones. Si hemos reducido el tamaño del batch, deberemos decrementar también la tasa de aprendizaje. Cuando empleamos tamaños de batches pequeños, los gradientes[7] se estiman con menos datos, lo que puede conducir a una estimación más ruidosa y una convergencia más lenta. Así, si en estos casos empleamos una tasa más baja, podríamos llegar a evitar actualizaciones de parámetros demasiado grandes e inestabilidades en el entrenamiento. Por el contrario, cuando los batches son más grandes, la convergencia suele ser más rápida y para ello es mejor aplicar tasas de aprendizaje más altas para poder actualizar los modelos correctamente.

Por estos motivos, la tasa de aprendizaje escogida ha sido de 1^{-5}. Esta corresponde a la empleada por el codificador. La empleada en otras capas es más alta, concretamente de 5^{-5}. Es la recomendada por GLiNER, probablemente debido a que las capas internas del modelo estén formadas por un conjunto alto de parámetros que requieran ajustes más rápidos para adaptarse a los datos.

Otros parámetros interesantes son *freeze_token_rep*, el cual esta establecido a falso. Este parámetro indica al modelo si se deben 'congelar' las representaciones de tokens durante el entrenamiento, es decir, si se permite su actualización o no. En el caso de GLiNER le hemos dado valor false ya

[6]Los tensores en informática son estructuras de datos multidimensionales que generalizan los conceptos de escalares, vectores o matrices de dimensiones superiores. Por ejemplo, un vector es un tensor de grado 1 y un escalar de grado 0.

[7]El gradiente indica la dirección y la magnitud del cambio más rápido de la función de pérdida con respecto a sus parámetros. Con él, obtenemos información para ajustarlos y minimizar dichas pérdidas.

que queremos que el modelo se adapte cuanto más a nuestros datos, pues trabajamos con conceptos muy específicos dentro de la medicina. En el caso de RoBERTa, el modelo fue preentrenado con un conjunto de datos médicos del PlanTL, por lo que no tocamos el parámetro equivalente ya que por defecto está congelado. Es decir, no queremos actualizar las representaciones de tokens que ya poseía del preentreno para no arriesgarnos a perder información.

Por otro lado, el GitHub de GLiNER recomienda usar otros parámetros como *shuffle_types*, para indicar si queremos mezclar los tipos de entidades durante el entrenamiento de cara a evitar sesgos; *random_drop*, para indicar si se deben eliminar aleatoriamente tipos de entidades durante el entrenamiento en cada iteración, para prevenir el *overfitting*; *max_neg_type_ratio*, especifica el ratio entre el número de muestras positivas[8] y el número de muestras negativas para cada tipo de entidad en el entrenamiento. Al presentar valor 1, queremos que haya un balance entre ambas muestras, lo que ayuda al modelo a aprender y generalizar. Por último, *warmup_ratio* indica la proporción de pasos de entrenamiento dedicados al calentamiento del optimizador, en nuestro caso, un 10 %. Este calentamiento es una estrategia que implica aumentar gradualmente la tasa de aprendizaje durante los primeros pasos del entrenamiento y luego mantenerla constante o reducirla. Además de posibilitar que el modelo sea más rápido, mejora su estabilidad y facilita su convergencia en muchos casos.

Para terminar, el parámetro *max_types* indica la cantidad de tipos de entidades a detectar. En cuanto a *max_len*, establece la longitud máxima de tokens de la sentencia, que en nuestro caso se ha establecido a 700. Aunque hay casos clínicos con mayor longitud, por limitaciones de Colab este ha sido el valor máximo que hemos podido emplear.

Listing 7.28: Método para entrenar el modelo

```
# Metodo para entrenar nuestro conjunto de datos
#   Argumentos:
#   - model: Modelo empleado durante el entrenamiento
#   - condig: Parametros de configuracion (hiperparametros)
#   - train_data: Conjunto de entrenamiento
#   - eval_data: Conjunto de validacion (si hay)
def train(model, config, train_data, eval_data=None):
    model = model.to(config.device)

    # Establecemos los hiperparametros del modelo
    model.set_sampling_params(
        max_types=config.max_types,
```

[8]Una muestra positiva en NER es una instancia en la que el modelo debe identificar correctamente una entidad. El caso contrario es una muestra negativa.

```
13          shuffle_types=config.shuffle_types,
14          random_drop=config.random_drop,
15          max_neg_type_ratio=config.max_neg_type_ratio,
16          max_len=config.max_len
17      )
18
19      # Configuramos el modelo para el entrenamiento
20      model.train()
21
22      # Creamos un data loader para los datos de entrada
23      train_loader = model.create_dataloader(train_data,
            batch_size=config.train_batch_size, shuffle=True)
24
25      # Cargamos el optimizador
26      optimizer = model.get_optimizer(config.lr_encoder,
            config.lr_others, config.freeze_token_rep)
27
28      # Creamos una barra de progreso
29      pbar = tqdm(range(config.num_steps))
30
31      # Configuramos el calentamiento
32      if config.warmup_ratio < 1:
33          num_warmup_steps = int(config.num_steps * config.
                warmup_ratio)
34      else:
35          num_warmup_steps = int(config.warmup_ratio)
36
37      # Programador de tasa de aprendizaje con
            calentamiento con decaimiento cosenoidal
38      scheduler = get_cosine_schedule_with_warmup(
39          optimizer,
40          num_warmup_steps=num_warmup_steps,
41          num_training_steps=config.num_steps
42      )
43
44      # Iterador de los datos de entrenamiento
45      iter_train_loader = iter(train_loader)
46
47      for step in pbar:
48          try:
49              x = next(iter_train_loader)
50          except StopIteration:
51              # Reiniciar iterador cuando alcancemos el
                    final del data loader
52              iter_train_loader = iter(train_loader)
53              x = next(iter_train_loader)
54
55          # Movemos los datos al dispositivo especificado
56          for k, v in x.items():
```

```
57              if isinstance(v, torch.Tensor):
58                  x[k] = v.to(config.device)
59
60          loss = model(x)  # Forward pass
61
62          # Comprobamos si la perdida es NaN
63          if torch.isnan(loss):
64              continue
65
66          loss.backward()  # Calculo de gradientes
67          optimizer.step()  # Actualizacion de parametros
68          scheduler.step()  # Actualizacion del programador
                de tasa de aprendizaje
69          optimizer.zero_grad()  # Reinicio de gradientes
70
71          description = f"step: {step} | epoch: {step //
               len(train_loader)} | loss: {loss.item():.2f}"
72          pbar.set_description(description)
73
74          # Si tenemos que evaluar el modelo...
75          if (step + 1) % config.eval_every == 0:
76              model.eval()
77
78              # Si existe un conjunto de validacion...
79              if eval_data is not None:
80                  # Evaluamos el modelo
81                  results, f1 = model.evaluate(eval_data["
                        samples"], flat_ner=True, threshold=0
                        .5, batch_size=12,
82                                               entity_types=
                                                    eval_data["
                                                    entity_types"])
83
84                  print(f"Step={step}\n{results}")
85
86              if not os.path.exists(config.save_directory):
87                  os.makedirs(config.save_directory)
88
89              if step == config.num_steps - 1:
90                model.save_pretrained(f"{config.
                      save_directory}/
                      finetuned_MediumAll_Dev_{step}")
91
92              # Volvemos a entrenar
93              model.train()
```

Este código corresponde con el método empleado para el entrenamiento del modelo. En resumidas palabras, se configuran los parámetros del modelo a partir de los almacenados en el SimpleNamespace y cargamos los

datos de entrada junto con un optimizador. Así, a partir de un bucle, realizamos los pasos hacia delante (*forward pass*) para el cálculo de pérdidas, seguidos de los pasos hacia atrás (*backward pass o backpropagation*) para actualizar los parámetros del modelo. Asimismo, la tasa de aprendizaje es ajustada mediante un programador de calentamiento y un decaimiento cosenoidal[9]. Por último, evaluamos nuestro modelo periódicamente con los datos de validación, mostrando las métricas pertinentes y guardándolo cuando sea necesario.

Listing 7.29: Entrenamiento del modelo

```
# Si empleamos dev como validacion
len_val = len(dev_text)
# En caso contrario
len_val = int(len(data))*0.1

eval_data = {
    "entity_types": ["Gene", "Disease", "DNAMutation", "
       SNP", "DNAAllele", "NucleotideChange-BaseChange",
        "OtherMutation", "Transcript"],
    "samples": data[-len_val:]
}

train(model, config, data[:-len_val], eval_data)
```

Así, lo único que necesitamos para evaluar el modelo es crear un diccionario con dos claves, los tipos de entidades y el conjunto de validación con el mismo formato que el de entrenamiento. Con esto, los hiperparámetros establecidos previamente y el conjunto de entrenamiento, podemos llamar al método *train* que acabamos de implementar.

Esta versión para finetunear GLiNER es totalmente lícita. Sin embargo, la última propuesta no solo es más rápida, sino que ofrece mejores resultados como ya veremos.

En ella se propone el uso de un archivo Python trainer.py[10] externo el cual es proporcionado por el propio repositorio GitHub de GLiNER. Su contenido se basa en una serie de funciones y clases que facilitan y mejoran la implementación del ciclo de entrenamiento, evaluación y guardado de modelos.

Dado que el contenido de este archivo es relativamente extenso, y algunas de sus funciones son triviales, nos centraremos en resaltar sus secciones más

[9] El decaimiento cosenoidal es una técnica que se encarga de ajustar la tasa de aprendizaje de nuestro modelo siguiendo una función consenoidal, haciendo que varíe gradualmente.

[10] `https://github.com/urchade/GLiNER/blob/main/examples/finetuning/trainer.py`

significativas y relevantes para comprender su funcionalidad y estructura general.

Listing 7.30: Favorece paralelización de GPU

```
DEFAULT_DDP_KWARGS = DistributedDataParallelKwargs(
    find_unused_parameters=True
)
```

La gran primera diferencia es que favorece el entrenamiento paralelo del modelo en distintas GPUs. Para ello usa la clase *DistributedDataParallelKwargs* de la biblioteca *Accelerate*, que definimos en capítulos anteriores. Con ella podemos definir parámetros adicionales con los que trabajará Pytorch. En este caso *find_unused_parameters*, que indica que se deben buscar parámetros no utilizados durante el *forward pass* para evitar errores en los gradientes.

A continuación, existe una clase dedicada especialmente para el entrenamiento de este tipo de modelo, llamada *GlinerTrainer*, que hereda de *torch.nn.Module* (una clase que sirve como estructura base para construir redes neuronales personalizadas). Gracias a ella, podemos manejar mucho más fácilmente los hiperparámetros que anteriormente teníamos almacenados en un SimpleNamespace. Además, proporciona métodos mucho más robustos y detallados, incluido el encargado del entrenamiento del modelo como tal.

Su constructor establece unos parámetros por defecto para el modelo en caso de que no se den, cuyos valores son realmente parecidos a los que usamos anterioridad [Listing 7.27] (ya que son los más comunes). Asimismo, se encarga de realizar las configuraciones necesarias para la ejecución distribuida y paralela del modelo.

En cuanto a otros métodos relevantes:

- **trackers**: Es un decorador de contexto que se encarga de inicializar los pesos y biases[11] para el registro de métricas.
- **log**: Registra datos durante el entrenamiento del modelo, como métricas de rendimiento, pérdidas, puntuaciones de validación, etc. Estos registros son usados posteriormente para realizar un seguimiento del rendimiento del modelo y poder comparar diferentes configuraciones de entrenamiento.
- **device**: Devuelve el disposito donde se encuentra el modelo (CPU, GPU (CUDA)...).

[11] Los biases, o sesgos, son parámetros adicionales que se utilizan en cada capa de un red neuronal, aparte de los pesos. Gracias a ellos, la red es capaz de modelar funciones más complejas.

- **wait**: Sincroniza todos los procesos que están ejecutando una instancia de entrenamiento.
- **unwrapped_model**: Devuelve el modelo desempaquetado si estamos usando aceleración distribuida.

Los demás métodos son más triviales y se encargan de guardar o cargar el estado actual del modelo, o trabajar con los procesos que se están ejecutando de manera paralela.

El método *train* que nos interesa presenta una estructura notablemente similar al que fue discutido previamente [Listing 7.28]. Sin embargo, esta versión incluye adiciones significativas, a partir de los métodos que acabamos de mencionar. Por esto, no solo se mejora la eficiencia y robustez del proceso de entrenamiento, sino que también conduce a un rendimiento general del modelo más satisfactorio.

Listing 7.31: Importación de bibliotecas

```
!pip install beartype

import sys

sys.path.append("/content/drive/MyDrive/My_TFG/
    DatasetGenovardis")
from trainer import GlinerTrainer
```

Sabiendo como se estructura el archivo trainer.py, lo siguiente que tenemos que hacer es importarlo en nuestro entorno Colab. Para ello, necesitamos instalar la biblioteca Beartype, ya que es requerida por trainer.py. Esta biblioteca ofrece herramientas para verificación de tipos en tiempo de ejecución.

A continuación, con la biblioteca sys, de Python estándar, añadimos el path donde se encuentra trainer.py a nuestro entorno de ejecución. Con esto, ya podemos importar la clase GlinerTrainer sin problema.

Listing 7.32: Entrenamiento del modelo, versión 2

```
device = torch.device("cuda" if torch.cuda.is_available()
    else "cpu")
model = model.to(device)

eval_data = {
    "entity_types": ["Gene", "Disease", "DNAMutation", "
       SNP", "DNAAllele", "NucleotideChange-BaseChange",
        "OtherMutation", "Transcript"],
    "samples": data[-len(dev_text):]
```

```
}

trainer = GlinerTrainer(model,
    train_data = data[:-len(dev_text)],
    batch_size = 2,
    grad_accum_every = 16,
    lr_encoder = 1e-5,
    lr_others = 5e-5,
    freeze_token_rep = False,
    val_every_step = 214,
    val_data = eval_data,
    checkpoint_every_epoch = 10,
    max_types=25,
    max_len=700,
)

torch.cuda.empty_cache()

trainer.train(num_epochs=20)

trainer.model.save_pretrained("/content/drive/MyDrive/
   My_TFG/DatasetGenovardis/logs/
   finetuned_MediumAll_Dev_Trainer")
```

Para ejecutarlo, volvemos a crear un diccionario con el conjunto de validación y las etiquetas a identificar. A continuación, creamos un objeto de tipo GlinerTrainer en donde los argumentos serán el modelo GLiNER a entrenar y sus hiperparámetros, los cuales poseen los mismos valores que en la versión previa de fine-tuning. Por último, comenzamos el entrenamiento indicando el número de épocas y guardamos el modelo resultante.

Es importante destacar que antes de comenzar dicho entrenamiento, debemos haber movido el modelo a un dispositivo que, de ser posible, será CUDA. Esto indica que hay GPUs disponibles para ser usadas. Del mismo modo, hay que cerciorarse de liberar la memoria que pueda haber en la GPU antes de la ejecución, debido a las limitaciones de Colab.

Evaluación

Para evaluar el modelo GLiNER entrenado previamente (archivo EvaluacionGLiNER.ipynb), primero necesitamos montar Google Drive y leer el archivo *test_text* o *dev_text* según corresponda.

Listing 7.33: Instanciar modelo

```
my_model = GLiNER.from_pretrained("/content/drive/MyDrive
   /My_TFG/DatasetGenovardis/logs/
```

```
    finetuned_MediumAll_Dev_Trainer", local_files_only=
    True)
```

El siguiente paso es obtener una instancia de nuestro modelo. Para ello, usamos el método *from_pretrained* de la biblioteca GLiNER de Hugging Face, indicando la ruta del mismo.

Listing 7.34: Realizar predicciones

```
entities = my_model.predict_entities(text, labels,
    threshold=0.5)

#( Que devolveria una lista de este tipo
[{'start': 11,
  'end': 35,
  'text': 'Deficiencia total de C4B',
  'label': 'Disease',
  'score': 0.6811932921409607}
  ...
]
```

Una vez tenemos el clasificador, simplemente nos queda realizar predicciones y almacenar los resultados en un archivo TSV en formato similar al de la Tabla 3.2. Para realizar una predicción, debemos pasar el caso clínico concreto junto con los tipos de entidad y el umbral de aceptación (de nuevo, 0.5). Devolverá un vector con todas las entidades detectadas e información como el score, comienzo y fin de la entidad en el texto.

Como vemos, GLiNER permite filtrar directamente las entidades a devolver a partir del umbral. Mientras que con RoBERTa lo tuvimos que hacer manualmente.

De nuevo, obviaremos la creación del archivo TSV ya que es realmente parecida a la mostrada en secciones anteriores [Listing 7.13].

Capítulo 8

Experimentación

En este capítulo analizaremos los resultados obtenidos por los tres modelos implementados para la tarea de NER. Primero, discutiremos los resultados individuales de cada modelo, lo que nos permitirá identificar y evaluar sus fortalezas y debilidades. Luego, compararemos los resultados para extraer conclusiones significativas sobre su desempeño.

Para los modelos autocodificadores, RoBERTa y GLiNER, también comentaremos la evolución durante el proceso de aprendizaje a partir de las métricas obtenidas por cada época. Además, comenzaremos viendo brevemente los resultados tras emplear el archivo *dev_text* como test. Posteriormente, veremos si ha habido mejores en los resultados al emplear el mismo para entrenamiento como conjunto de validación y el archivo *test_text* para test.

Para el caso concreto de GLiNER, también comentaremos las diferencias de usar el modelo Small frente al Medium. Asimismo, analizaremos la aplicación de las dos versiones de fine-tuning implementadas.

Los archivos empleados para este apartado son EvaluacionGPT.ipynb, EvaluacionRoBERTa.ipynb y EvaluacionGLiNER.ipynb.

8.1. Evaluación de GPT-3.5

Como sabemos, GPT es un modelo generativo y, como ya mencionamos durante su implementación, a pesar de haber proporcionado un prompt bastante detallado sobre que tarea queremos que realice, no siempre los resultados fueron los esperados.

Es por ello que, antes de comenzar el análisis a partir de métricas y gráficas, es bastante interesante mostrar algunos casos concretos que nos devolvió el modelo para que podamos ver como todavía presenta ciertas

limitaciones. Los fallos que mas se repetían fueron los siguientes:

- Detección de tipos de entidades completamente distintos a los especificados. A pesar de que lo habíamos enfatizado en el prompt con la sentencia 'sola y únicamente detecta estos tipos de entidades'.

Listing 8.1: Detección de tipos ajenos al problema

```
['Factores neurotroficos --> neurotroficos',
'Trastorno por deficit de atencion/hiperactividad
   --> TDAH',
'Neurotrofinas --> neurotrofinas',
'Proteinas --> proteinas',
...
]
```

- Mostrar una respuesta en un formato erróneo al no encontrar entidades en el texto. En estos casos recordemos que debía devolver la cadena vacía, no siempre lo hacía.

Listing 8.2: Formato erróneo al no encontrar entidades

```
['Tipo de entidad no encontrada.']
```

- No respectar el formato de salida aún cuando se encontraban entidades:

Listing 8.3: Formato erróneo al mostrar entidades

```
['Transcript -->']
# O bien
['Gene --> BRCA1', '', 'Gene --> BRCA2']
```

Así, cuando durante la implementación mencionamos que realizamos bastantes comprobaciones de las respuestas que nos devolvía el modelo, nos referíamos a estos casos en específico, ya que desafortunadamente no son de utilidad y deben ser ignorados.

Internamente podemos pensar que si probamos con más tipos de prompts, más detallados aún quizás, podríamos tratar este tipo de fallos. Sin embargo, realmente no existe ninguna certeza de ello. La posibilidad de que surjan otros nuevos seguiría ahí y es algo totalmente normal. Hay que tener en cuenta que estamos pidiendo al modelo una tarea realmente compleja para la cual no fue entrenado específicamente, por lo que obtener unos resultados precisos no es tan fácil, incluso imposible.

Así, el TSV que generó este modelo tras aplicarlo al archivo *test_text.tsv* presentó los siguientes resultados (recordemos, este TSV lo subimos a la competición de Codalab y el propio servidor es el que nos los proporcionó. Aunque también se podrían obtener manualmente):

Modelo	Precisión	Recall	F1-Score
GPT-3.5	0.400000	0.186578	0.254463

Cuadro 8.1: Métricas de evaluación test de GPT-3.5

Para explicarlos mejor, nos valdremos de su matriz de confusión, pues es una de las técnicas más empleadas en aprendizaje automático para evaluar modelos. En ellas, podemos ver el número de predicciones correctas e incorrectas hechas por el modelo en comparación con los resultados reales.

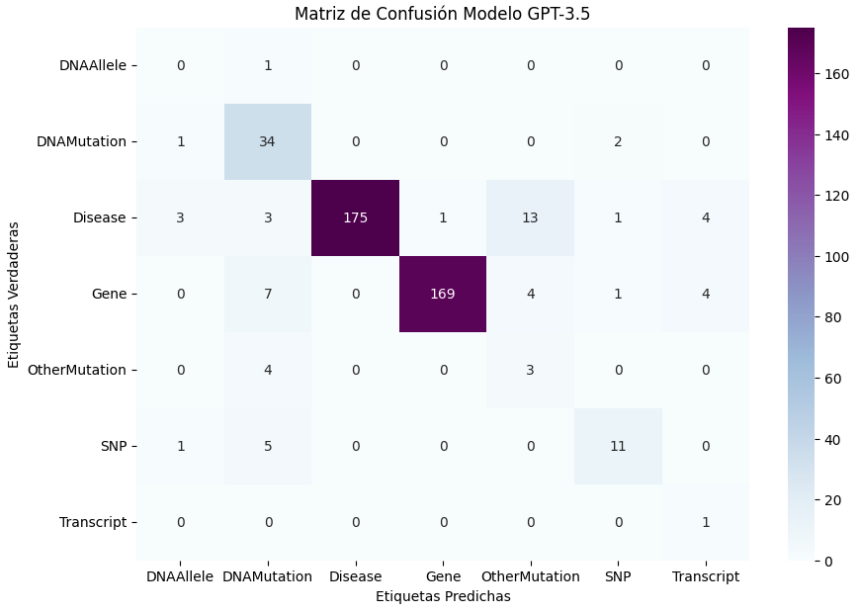

Figura 8.1: Matriz de Confusión de GPT-3.5

Así, es su diagonal la que nos va a proporcionar información de cuántas entidades han sido identificadas correctamente. Para este modelo vemos que, en general, los valores más altos se encuentran en ella. Esto puede parecer algo realmente positivo pues implicaría que la cantidad de falsos positivos y falsos negativos, correspondientes al resto de valores de la matriz, son poco comunes. De hecho, el modelo GPT parece que tiene más dificultades a la hora de clasificar enfermedades (Disease), pues es donde presenta mas fallos al identificarla como otra mutación (OtherMutation) con un total de 13 errores. Por tanto, estos resultados parecen no ir de la mano con las métricas ya que el F1-Score es bastante bajo.

Esto se debe a que estamos trabajando con modelos NER, y las matrices de confusión por sí mismas no siempre proporcionan toda la información que el modelo puede ofrecer. Para ello, se propone complementarla con la siguiente gráfica.

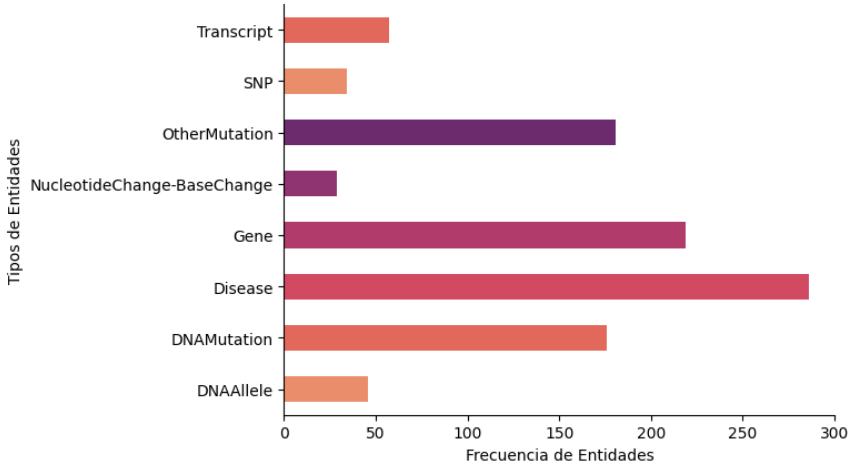

Figura 8.2: Gráfica de frecuencias de entidades de GPT-3.5

Cuadro 8.2: Frecuencias concretas de entidades predichas por GPT-3.5

Label	**Frecuencia**
Disease	286
Gene	219
OtherMutation	181
DNAMutation	176
Transcript	57
DNAAllele	46
SNP	34
NucleotideChange-BaseChange	29
Total	1028

De nuevo, mostramos una gráfica de frecuencias pero en este caso de las entidades predichas por nuestro modelo. Vemos que en total se han realizado 1028 predicciones (independientemente de si se ha identificado bien su tipo o no); de modo que es lógico pensar que si sumamos el total de todos los valores de las celdas de la matriz de confusión, obtengamos esta cantidad. En un gran número de problemas de aprendizaje automático esto es así. Sin embargo, en el caso de NER no es común. Hay que tener en cuenta que el modelo puede identificar palabras en el texto que realmente no corresponden con ninguno de los tipos de entidad posibles, aunque el modelo lo detecte como tal. Estos casos son también falsos positivos, pero no los

podemos representar en la matriz de confusión pues no poseen una etiqueta real asociada.

Para saber cuál es la cantidad de falsos positivos descartados en la matriz de confusión, podemos emplear los Dataframes de Pandas. En uno almacenamos el archivo TSV con las predicciones del modelo y en otro el TSV con las anotaciones de test, *test_annotation.tsv*. Si los unimos mediante el método *merge* y conservamos las columnas label de cada uno (renombradas a label_pred y label_true, respectivamente), obtendremos un Dataframe con la cantidad de entidades detectadas correctamente ya que solo se conservarán aquellas filas que coincidan totalmente en ambos (a excepción de label). Hay que recalcar esto, pues que se haya detectado una entidad correctamente no quiere decir que se haya categorizado bien (puede detectarse la entidad 'diabetes' que sería correcta pues es tipo Disease, pero clasificarse como OtherMutation).

El resultado de esto es que de 1028 predicciones que se han realizado, únicamente 453 han detectado bien las entidades, que son las representadas en la matriz de confusión, y de donde 393 de ellas han categorizado bien el tipo, que son la diagonal. Por consiguiente, 575 son falsos positivos que han de ser descartados, el 55.93 % del total de predicciones. Una cantidad bastante considerable.

Esto no es todo, pues según nuestro archivo de anotaciones de test, un resultado ideal correspondería a la detección de 2101 entidades. Algo más del doble de las identificadas por el modelo.

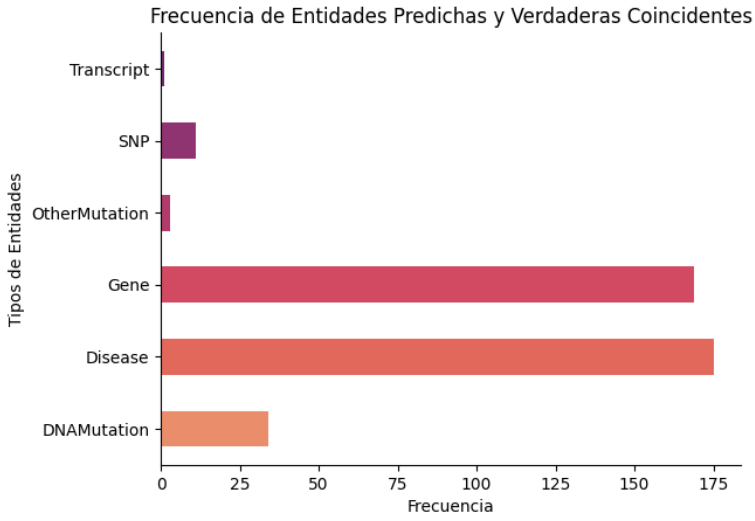

Figura 8.3: Gráfica de frecuencias de entidades predichas correctamente de GPT-3.5

Esta sería la gráfica final con la cantidad de entidades predichas y clasificadas correctamente por nuestro modelo GPT. Si la comparamos con Figura 8.2, podemos ver como etiquetas como DNAAllele y NucleotideChange-BaseChange han desaparecido completamente, pues eran erróneas. Al igual que la frecuencia de entidades por cada tipo a disminuido drásticamente. Esta gráfica es equivalente a la diagonal de la matriz de confusión.

En definitiva, tras ver todas estas particularidades, queda más que claro el porqué las métricas presentan valores tan bajos.

La precisión es la que posee el valor más alto, y nos indica que un 40 % de entidades detectadas por el modelo son realmente correctas. Por contraposición, el recall es bastante bajo y significa que el modelo solo logra detectar el 18.66 % de todas las entidades reales presentes en el texto, hecho que acabamos de corroborar con las demás gráficas. La F1-score es la media armónica de ambas, y con un valor del 25.44 % nos indica que el rendimiento global es muy limitado.

Si tuviéramos que sacar algún punto más positivo, podríamos destacar que obtiene mejores resultados al detectar entidades de longitud corta que larga, como podemos ver en la siguiente gráfica:

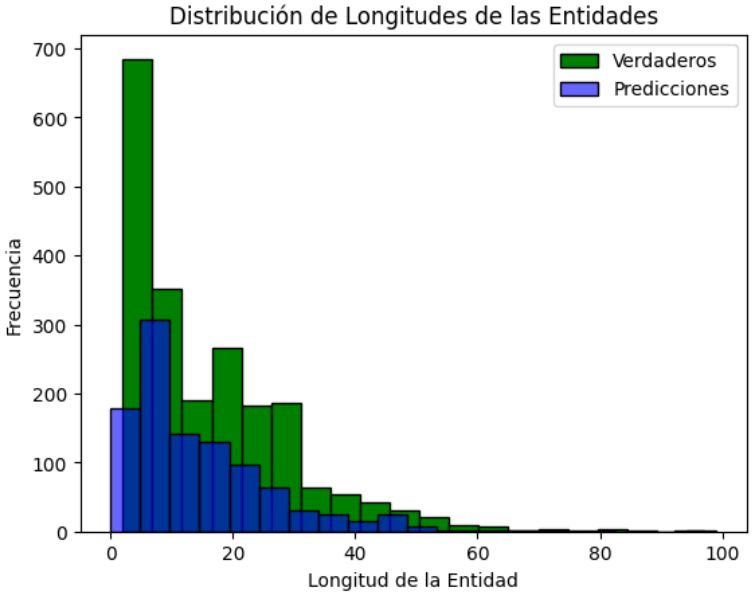

Figura 8.4: Gráfica de distribución de longitudes de GPT-3.5

En ella se puede ver como el modelo es más exitoso cuando categoriza entidades de corta longitud, pues las entidades verdaderas presentan esa característica. Aunque, de nuevo, apreciamos que identifica una cantidad muy baja en comparación con lo que sería ideal.

A partir de este análisis, podemos concluir que los modelos generativos, en concreto GPT, no son una buena solución para problemas NER de cierta complejidad. Es bastante probable que si el conjunto de tipos de entidades a detectar fuera más simple (nombres de empresas, localizaciones, nombres propios, comida...) los resultados serían bastante mejores, con una gran diferencia. Sin embargo, nuestros tipos no solo abarcan un campo más complejo como el clínico, sino que además se enfocan en variantes y enfermedades, que podemos considerar un tópico de nicho mucho más específico, complicado y extenso.

Previamente también hablamos de la posibilidad de mejorar el prompt, ya sea añadiendo detalles o aplicando técnicas más complejas de ingeniería de prompt. Sin embargo, haciendo solo esto no podemos esperar dar solución al problema. Existe un cuello de botella originado por el propósito de estos modelos, generación y comprensión del lenguaje, por lo que no están tan optimizados para tareas específicas como NER.

También es relevante considerar el uso de modelos GPT más avanzados, como GPT-4 o el más reciente, GPT-4o. Es probable que ofrezcan mejoras en los resultados; sin embargo, partiendo de un F1-score del 25.44 % con GPT-3.5, no podemos esperar una mejora tan significativa que permita a estos modelos alcanzar un rendimiento competitivo en la actualidad.

8.2. Evaluación de RoBERTa

Como dijimos, la evaluación del modelo RoBERTa consta de dos partes. Una donde usamos el archivo *dev_text.tsv* como conjunto test y otra donde lo usamos como conjunto de validación. Para comentar sus resultados, realizaremos un análisis conjunto de ambos dado que, a pesar de ser distintos, ambos presentan una muy buena distribución de clases, siendo la única diferencia que este último posee una cantidad algo mayor de ejemplos.

Comenzaremos viendo como ha sido la evolución de su aprendizaje durante las 20 épocas que duró su entrenamiento en el fine-tuning [Tabla 8.3 y Tabla 8.4]. Además, para proporcionar una mejor visión de ello, emplearemos gráficas de curvas de aprendizaje en donde se representa una comparación de la evolución de la pérdida en validación y entrenamiento. Son bastante comunes en problemas de aprendizaje automático y dado que el modelo proporciona esa información, merece la pena discutirla.

Cuando hablamos de pérdidas, nos referimos a aquellas medidas que nos indican si nuestro modelo está o no aprendiendo. Así, la pérdida en entrenamiento mide cuán bien lo hace durante el entrenamiento; mientras que la pérdida en validación, mide qué tan bien el modelo generaliza con datos nuevos.

Época	Training Loss	Validation Loss	Precisión	Recall	F1 Score	Accuracy
1	1.511400	0.665041	0.000000	0.000000	0.000000	0.871055
2	0.630600	0.450983	0.472941	0.258687	0.334443	0.886588
3	0.437800	0.315294	0.587886	0.637066	0.611489	0.922051
4	0.344100	0.244024	0.591966	0.720721	0.650029	0.940421
5	0.222900	0.206363	0.654420	0.733591	0.691748	0.944961
6	0.197600	0.194973	0.661435	0.759331	0.707010	0.948933
7	0.173300	0.184059	0.694282	0.765766	0.728274	0.951628
8	0.149900	0.184401	0.672626	0.774775	0.720096	0.950777
9	0.140600	0.185036	0.698026	0.773488	0.733822	0.949926
10	0.123500	0.191279	0.685714	0.803089	0.739775	0.947727
11	0.114600	0.185815	0.698305	0.795367	0.743682	0.951770
12	0.109100	0.182168	0.705950	0.794080	0.747426	0.952479
13	0.098100	0.188601	0.704545	0.797941	0.748340	0.952621
14	0.095500	0.183993	0.721198	0.805663	0.761094	0.954961
15	0.086300	0.194102	0.704646	0.819820	0.757882	0.950777
16	0.083400	0.196870	0.705882	0.818533	0.758045	0.950422
17	0.082100	0.194591	0.718502	0.814672	0.763571	0.952337
18	0.080900	0.195982	0.707589	0.815959	0.757920	0.951486
19	0.074000	0.195931	0.717367	0.813385	0.762364	0.952834
20	0.077100	0.197207	0.710762	0.815959	0.759736	0.951770

Cuadro 8.3: Evolución del aprendizaje del modelo RoBERTa con *dev_text* para test.

Época	Training Loss	Validation Loss	Precisión	Recall	F1 Score	Accuracy
1	1.407300	0.625543	0.000000	0.000000	0.000000	0.877693
2	0.614700	0.374606	0.681333	0.394900	0.500000	0.902404
3	0.313900	0.266682	0.594726	0.662287	0.626691	0.932033
4	0.245100	0.230608	0.634535	0.707110	0.668860	0.934700
5	0.219200	0.213494	0.656099	0.735703	0.693625	0.940326
6	0.165600	0.207407	0.662293	0.741113	0.699489	0.941826
7	0.152300	0.203427	0.700962	0.731839	0.716068	0.944493
8	0.136600	0.204733	0.688112	0.760433	0.722467	0.943326
9	0.119800	0.201074	0.681943	0.770479	0.723512	0.943618
10	0.115500	0.197284	0.702914	0.764297	0.732321	0.947160
11	0.107800	0.204496	0.690426	0.763524	0.725138	0.945827
12	0.091300	0.210696	0.685832	0.774343	0.727405	0.944076
13	0.090100	0.206691	0.700284	0.761978	0.729830	0.946743
14	0.090300	0.213762	0.692629	0.769706	0.729136	0.945201
15	0.078700	0.214202	0.698601	0.772025	0.733480	0.946493
16	0.076100	0.214365	0.702381	0.775116	0.736958	0.946868
17	0.075000	0.214048	0.698258	0.774343	0.734335	0.946827
18	0.072400	0.218142	0.704225	0.772798	0.736920	0.946535
19	0.070500	0.216484	0.697982	0.775116	0.734529	0.946660
20	0.066800	0.216410	0.703289	0.776662	0.738156	0.946785

Cuadro 8.4: Evolución del aprendizaje del modelo RoBERTa con *dev_text* para validación.

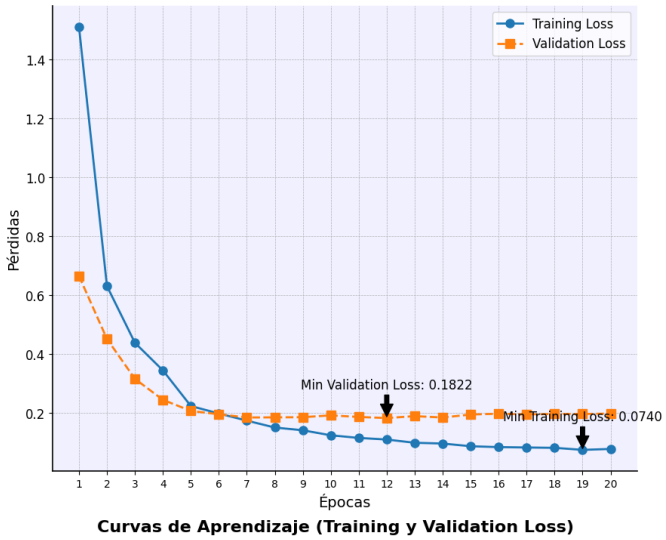

Figura 8.5: Curvas de aprendizaje para RoBERTa usando *dev_text* como test.

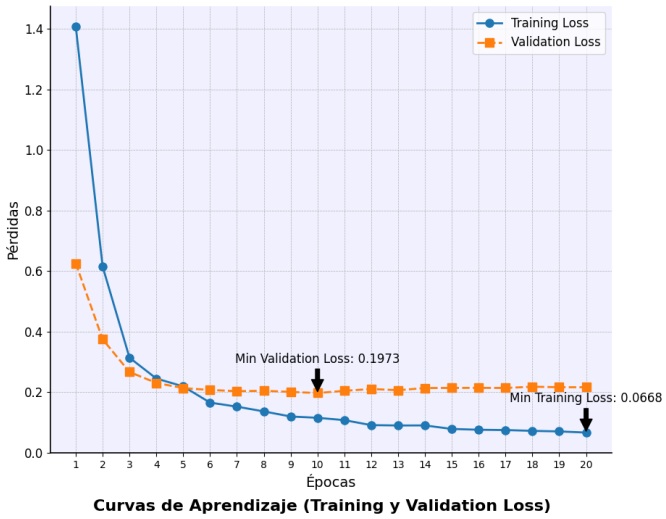

Figura 8.6: Curvas de aprendizaje para RoBERTa usando *dev_text* como validación.

Si prestamos atención a las gráficas de curvas presentadas [Figura 8.7 y Figura 8.8] vemos que son realmente parecidas. En ambas vemos aspectos positivos como una disminución rápida de la pérdida, lo que nos está indicando que el modelo aprende rápidamente, y en especial al inicio. Sin embargo, llega un punto (aproximadamente entre las 10-12 épocas) donde la pérdida comienza a estabilizarse. Esto implica que nuestro modelo ha al-

canzado un punto donde la mejora es realmente baja y lo más probable es que no pueda aprender más, al menos con la configuración que empleamos.

Aún así, es importante destacar que dicha estabilidad es relativa, pues la pérdida en entrenamiento va disminuyendo, aunque muy ligeramente, pero no es capaz de generalizar mejor. Esto quiere decir que existe un pequeño *overfitting*. Aún así, este es muy leve.

Este hecho ocurre para ambas implementaciones. De modo que, si tenemos en cuenta que cuando usamos *dev_text* para validación estamos empleando un conjunto más representativo, pues contiene una mejor distribución de clases, podríamos comenzar a suponer que será esta implementación la que ofrecerá mejores resultados. Aunque no con grandes diferencias.

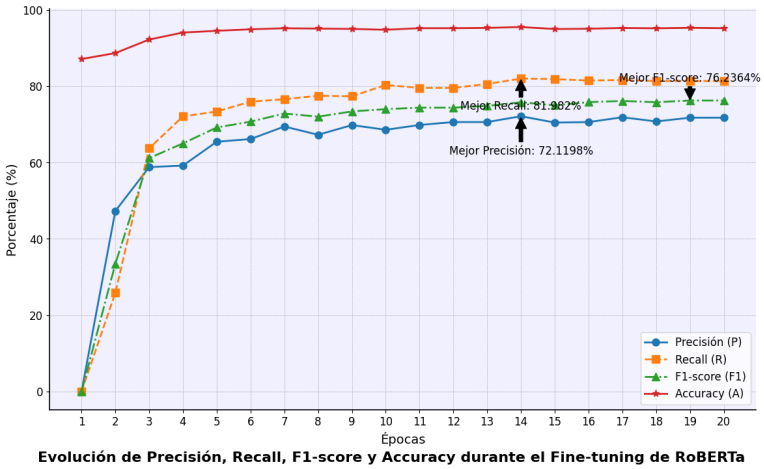

Figura 8.7: Evolución de métricas de RoBERTa usando *dev_text* como test.

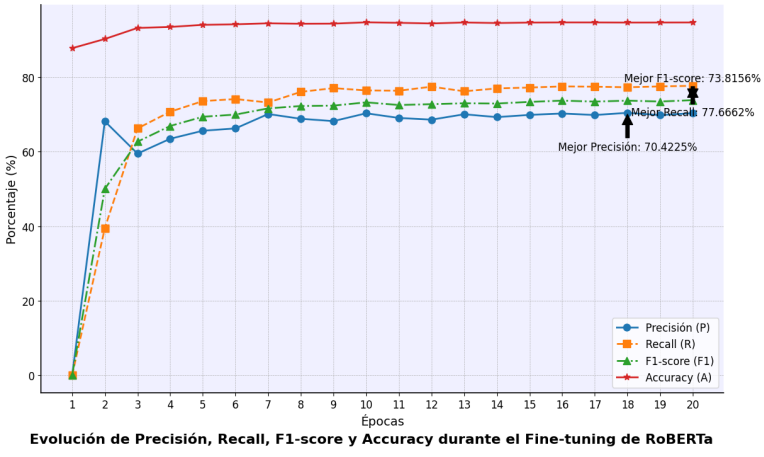

Figura 8.8: Evolución de métricas de RoBERTa usando *dev_text* como valid.

A continuación, analizaremos el resto de métricas. Lo primero a destacar es que todas ellas, exceptuando el accuracy, comienzan con valor cero en la primera época. El motivo es que el modelo no ha sido aún capaz de aprender lo suficiente para realizar predicciones, probablemente debido al desbalanceo de clases o que el corpus no contiene ejemplos suficientes. Sin embargo, no es mayor problema ya que a partir de las siguientes comienza a mostrar resultados.

Si comparamos las dos, de nuevo, vemos que son muy parecidas. El F1-score, al ser la media armónica de la precisión y el recall, siempre se encuentra entre ambas métricas. Si hablamos de su convergencia, vemos que es rápida y buena dado que alcanza valores que podemos considerar altos alrededor de la cuarta época aproximadamente. En ese punto los valores se vuelven a estabilizar, en algunos casos aumentando muy ligeramente.

Si nos fijamos en los mejores resultados, vemos que el mayor F1-score es 76.23 %, a pesar de emplear un conjunto de validación aleatorio del 10 %. Cuando este conjunto es el archivo *dev_text*, el mejor F1-score es del 73.81 %. Esto contradice un poco la idea anterior de que este último caso pudiera ser el que generalizase mejor. Sin embargo, no olvidemos que el modelo poseía un ligero *overfitting*, que podría verse reflejado en los resultados del F1-score, haciendo que no sean del todo fiables.

Antes de ver los resultados de la evaluación con test, merece la pena hablar del accuracy. En las gráficas esta métrica siempre posee valores bastante altos, alrededor del 90 %. Sin embargo, es una métrica engañosa al emplearla sobre nuestro conjunto de datos. Recordemos que durante la etapa de exploración vimos que nuestras clases están desbalanceadas, siendo los tipos de entidades Disease y Gene los más predominantes con diferencia. Esto significa que el accuracy, en este caso, nos indica que el modelo es capaz de predecir correctamente las clases mayoritarias, lo cual no reflejará un rendimiento real de este. Por tanto, no es buena práctica emplearla para problemas multiclase con esta característica.

Así, tras haber empleado ambas implementaciones del modelo sobre sus archivos test correspondientes y generar sus respectivos archivos TSV, Codalab nos proporcionó los siguientes resultados:

Modelo	Precisión	Recall	F1-Score
RoBERTa validación 10 %	0.570952	0.488339	0.526424
RoBERTa validación *dev_text*	0.611556	0.654926	0.632498

Cuadro 8.5: Métricas de evaluación test de RoBERTa

La realidad es que los resultados presentan diferencias significativas. Cuando usamos el archivo *dev_text* como conjunto de validación, el modelo es capaz de generalizar algo mejor. Algo lógico, pues posee más ejemplos y

una distribución de clases más equilibrada que al tomar como validación un 10 % aleatorio de los datos de entrenamiento.

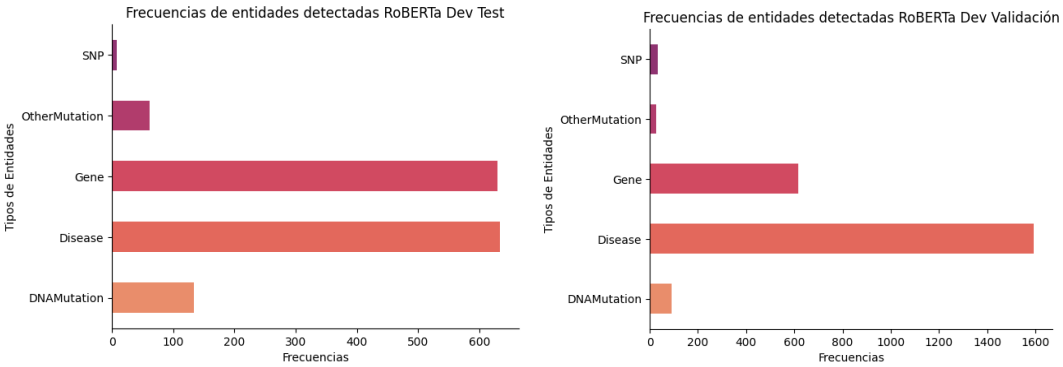

Figura 8.9: Frecuencias de tipos de entidades detectadas por ambas implementaciones de RoBERTa.

A continuación, mostramos las frecuencias de tipos de entidades detectados. Como era de esperar, las entidades más comunes pertenecen a la categoría Disease o Gene, pues son los tipos con más ejemplos en nuestro corpus y, por ende, los que más va a aprender a localizar el modelo. Otros como Transcript o DNAAllele, menos comunes, podemos ver que no han sido detectados nunca.

En total, el modelo correspondiente a la primera gráfica detectó 1466 entidades, de las cuales 581 han sido falsos positivos (un 39.63 %). Así, podemos representar el resto, 885, en la siguiente matriz de confusión:

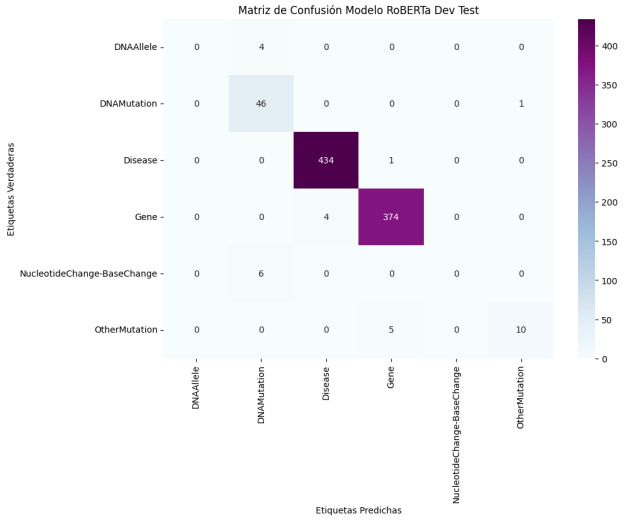

Figura 8.10: Matriz de confusión de RoBERTa usando *dev_text* como test.

En cuanto a la segunda gráfica, hizo 2354 predicciones, bastantes más. De las cuales 958 fueron falsos positivos (el 40.70 %). Por tanto, el resto de casos, 1396, pueden ser representados en la siguiente matriz de confusión:

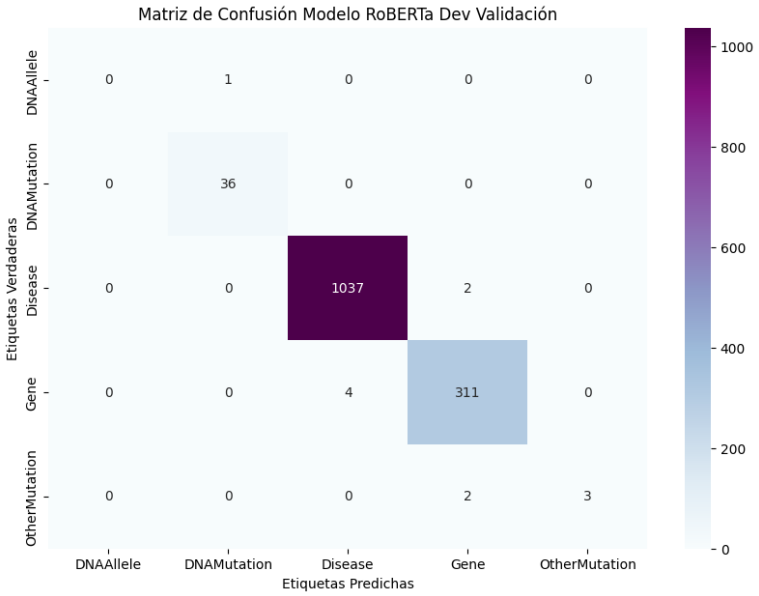

Figura 8.11: Matriz de confusión de RoBERTa usando *dev_text* como validación.

Ambas matrices muestran un gran número de buenas clasificaciones, pues los valores más altos se encuentran en su diagonal con una amplia diferencia.

En base a esto, ya podemos entender mucho mejor los resultados. Este último modelo ha realizado una cantidad mucho mayor de predicciones, a pesar de que la cantidad de falsos positivos a nivel porcentual sea prácticamente la misma. Esto significa que presenta una mayor sensibilidad para detectar entidades.

Aún así, a nivel de precisión ambos son similares. Cuando usamos *dev_text* como validación esta es del 61.15 %; en caso contrario, del 57.09 %. Este es el porcentaje de predicciones positivas hechas por el modelo que son realmente correctas. En cuanto al recall, aquí si que encontramos grandes diferencias, en donde el primer modelo mencionado presenta uno del 65.49 %, significativamente más grande al 48.83 % del segundo. Este porcentaje indica la cantidad de instancias positivas que el modelo es capaz de detectar. Como consecuencia de este bajo recall, el F1-score se ve drásticamente disminuido. Por tanto, podemos concluir que el conjunto de validación empleado cobra una importancia vital a la hora de realizar una detección adecuada.

A continuación, vamos a realizar una comparativa de la distribución de

las longitudes entre las predichas por el mejor modelo y las verdaderas.

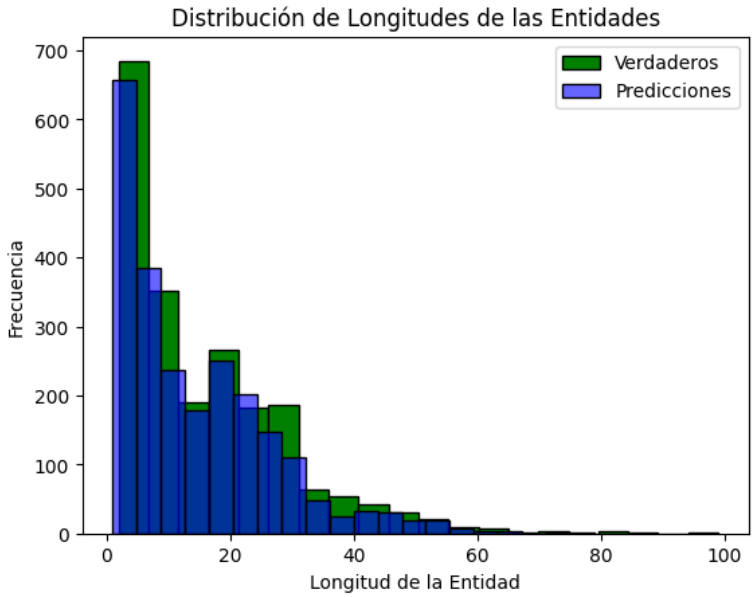

Figura 8.12: Gráfica de distribución de longitudes de RoBERTa.

Estos resultados son bastante buenos, pues significa que el modelo a aprendido a identificar muy bien aquellas entidades de corta longitud que, como vemos, realmente son las predominantes. No solo con eso, también es capaz de detectar aquellas más largas, a pesar de que su frecuencia sea menor. Llegando a ofrecer una distribución muy similar a la real.

Por último, mencionar que los tiempos de ejecución (únicamente el entrenamiento) de RoBERTa son muy bajos. En concreto, la ejecución de las 20 épocas tomó 15 minutos en completarse. Aproximadamente 45 segundos por cada una.

Por todo ello, podemos concluir que el rendimiento alcanzado por el modelo RoBERTa en el contexto biomédico, con un F1-score del 63.24 %, es altamente satisfactorio. Más aún dada la naturaleza específica y compleja de las entidades a detectar. En el propio paper de GLiNER [5], del que hablamos en capítulos anteriores, se dedica una parte a comparar el funcionamiento de diversos modelos BERT, RoBERTa entre ellos. Concretamente, compara el rendimiento zero-shot de estos modelos entre un conjunto de 20 datasets de tipo NER y 20 datasets de tipo OOD NER[1], algunos de ellos con datos biomédicos. Los resultados fueron los siguientes:

[1]Un dataset OOD NER, Out-of-Domain NER, es aquel formado por datos de un dominio diferente al que los datos de un modelo fueron entrenados. Su uso es bastante común cuando se aplican técnicas zero-shot.

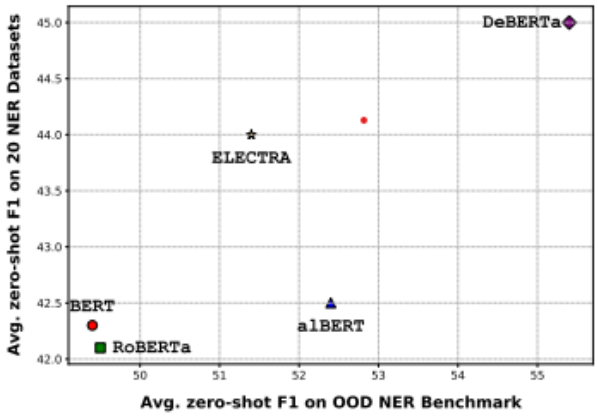

Figura 8.13: Comparación de rendimiento de distintos modelos BERT.

Como vemos, los modelos RoBERTa son los que obtienen resultados más bajos para su F1-Score, no llegando a alcanzar el 50 % en ninguno de los dos tipos de datasets. Así, podemos respaldar que nuestro modelo ha obtenido unos resultados favorables sobre nuestro corpus. Obviamente, no debemos olvidar que hemos empleado un modelo RoBERTa preentrenado con datos biomédicos al que le hemos realizado un fine-tuning para aplicarlo al dominio NER, lo cual no hay duda que también ha favorecido a la mejora de la generalización del modelo resultante.

8.3. Evaluación de GLiNER

Para evaluar GLiNER, emplearemos todas las implementaciones que mencionamos previamente y las compararemos. Así, comenzaremos examinando los resultados obtenidos por su versión Small y Medium en su versión más reciente, la 2.1, para cuando ambos emplean el archivo *dev_text* para test. O lo que es lo mismo, toman un 10 % aleatorio del conjunto de entrenamiento para validación.

A continuación, tomaremos el modelo que mejor resultados haya dado y esta vez usaremos *dev_text* como validación y *text_text* para test. Análogamente, se comparará con la versión que emplea el archivo trainer.py para el fine-tuning.

A diferencia del modelo RoBERTa, durante el análisis de curvas de aprendizaje no se ha podido tener en cuenta el error de validación, pues no fue proporcionado durante la implementación recomendada de GLiNER para fine-tuning. Comenzamos comparando entre GLiNER Small y Medium:

Épocas	Training Loss	Precisión	Recall	F1-Score
1	241.65	47.16	32.25	38.31
2	210.94	46.72	36.01	40.67
3	284.08	44.44	43.01	43.71
4	118.82	41.01	47.28	43.92
5	160.32	38.21	49.74	43.22
6	86.48	38.41	49.35	43.20
7	147.15	39.67	46.50	42.81
8	199.67	43.36	41.84	42.58
9	128.56	47.82	36.92	41.67
10	130.57	52.12	34.97	41.86
11	156.60	54.25	34.72	42.34
12	44.24	57.20	34.97	43.41
13	72.90	57.92	34.59	43.31
14	155.00	58.62	35.23	44.01
15	343.12	58.69	35.88	44.53
16	86.14	57.96	36.79	45.01
17	65.51	57.95	37.31	45.39
18	148.36	58.28	37.82	45.88
19	110.99	58.50	38.34	46.32
20	34.88	58.50	38.34	46.32

Cuadro 8.6: Evolución del aprendizaje del modelo GLiNER Small con *dev_text* para test.

Épocas	Training Loss	Precisión	Recall	F1-Score
1	75.93	55.18	42.75	48.18
2	247.08	53.05	45.08	48.74
3	66.39	50.63	51.81	51.22
4	165.75	49.09	55.83	52.24
5	77.87	49.50	57.38	53.15
6	183.35	51.15	57.51	54.15
7	138.93	50.93	56.48	53.56
8	211.08	54.03	57.25	55.60
9	53.52	56.05	56.99	56.52
10	50.11	57.88	56.61	57.24
11	75.25	59.18	56.35	57.73
12	80.41	60.58	56.35	58.39
13	67.62	61.13	56.22	58.57
14	93.97	61.22	55.83	58.40
15	59.78	61.68	56.09	58.75
16	107.86	61.42	56.09	58.63
17	173.86	61.29	56.61	58.86
18	82.55	61.29	56.61	58.86
19	98.00	61.29	56.61	58.86
20	213.05	61.29	56.61	58.86

Cuadro 8.7: Evolución del aprendizaje del modelo GLiNER Medium con *dev_text* para test.

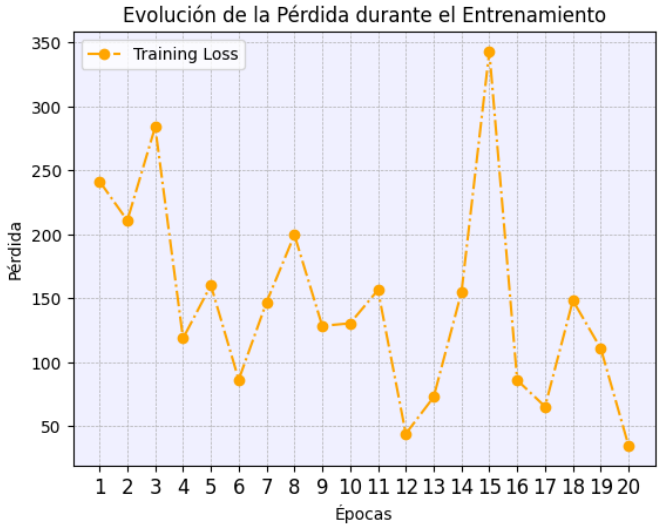

Figura 8.14: Curvas de aprendizaje para GLiNER Small usando *dev_text* como test.

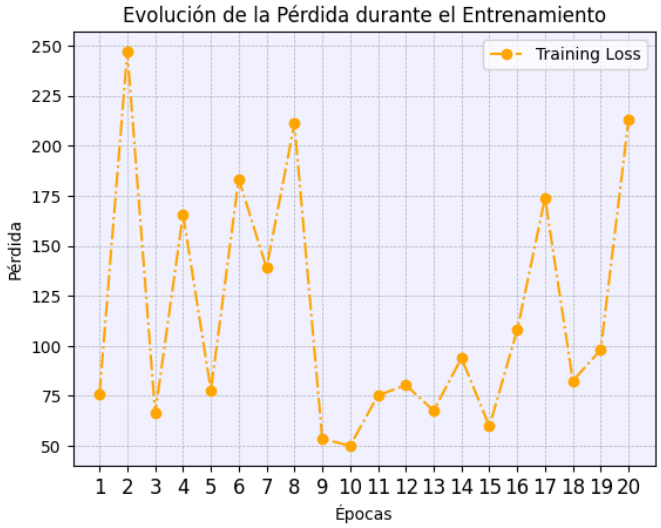

Figura 8.15: Curvas de aprendizaje para GLiNER Medium usando *dev_text* como test.

En este caso solo podemos evaluar la curva de aprendizaje correspondiente a la pérdida de entrenamiento. A primera vista, las curvas presentan una alta variabilidad que nos indica que es posible que el modelo necesite algunas mejoras, aunque no necesariamente tiene por que ser malo. El problema se podría deber a una tasa de aprendizaje demasiado alta, poca

optimización en la regularización, ruido en los datos de entrenamiento... Este último caso es más improbable pues el mayor problema que tenemos es el desbalanceo de clases.

Así, ambas curvas no nos proporcionan demasiada información ya que a simple vista podría parecer que el modelo no está aprendiendo lo suficiente. En este caso, si tuviéramos que decantarnos por un modelo, da la sensación de que el Small podría dar mejores resultados ya que a largo plazo converge mejor que el Medium. Este último, tiene una disminución significativa en la época 10, pero continúa aumentando durante el resto de épocas.

A continuación, pasaremos a evaluar lás métricas a partir de las siguientes gráficas [Figura 8.16 y Figura 8.17].

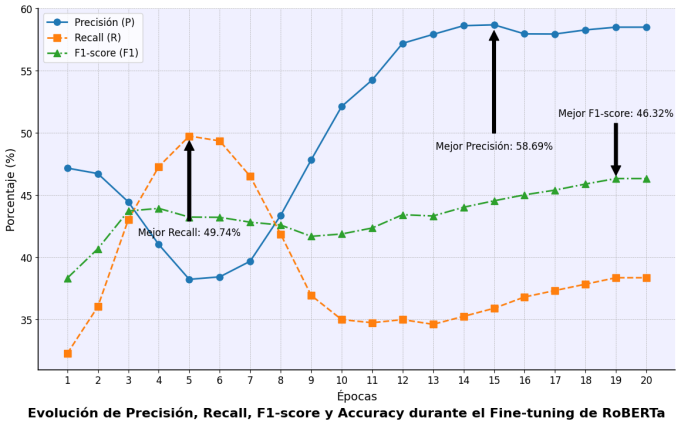

Figura 8.16: Evolución de métricas del modelo GLiNER Small con *dev_text* para test.

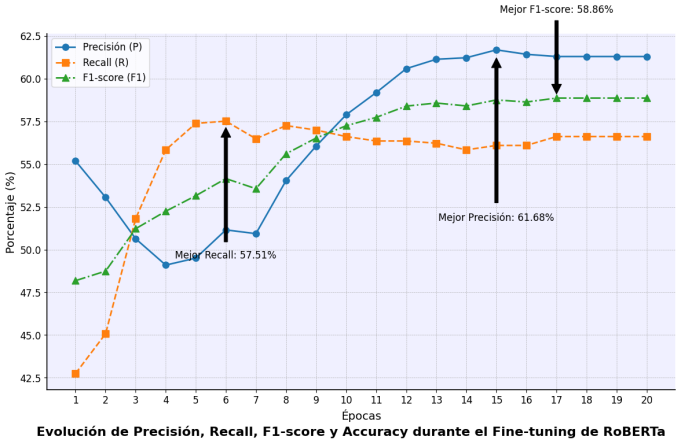

Figura 8.17: Evolución de métricas del modelo GLiNER Medium con *dev_text* para test.

En ambas gráficas podemos encontrar ciertas similitudes. La principal es que encontramos una tendencia general al aumento del rendimiento. Aunque con el modelo Small esta no es tan notable como con el Medium. En este último llegamos a alcanzar un pico de 58.86 % de F1-Score, mientras que en el primer mencionado este se encuentra en 46.32 %. Además, con el Medium este valor se encuentra mucho antes, tras seis épocas.

En cuanto a precisión y recall, de nuevo, el Small presenta valores más reducidos, con un 58.69 % y 49.74 %, respectivamente, en comparación el Medium, de 61.68 % y 57.51 %. Por tanto, este último es el que alcanza una mayor proporción de predicciones correctas de entre todas las positivas, así como también identifica una mayor proporción de entidades reales.

En cuanto a estabilidad, con el modelo Small encontramos muchas más fluctuaciones. La más significativa es que tras alcanzar el máximo valor de recall, disminuye considerablemente. Con Medium, la tendencia a la mejora es más estable y consistente a lo largo del tiempo, sobre todo a partir de la época 10.

Por tanto, en base a estas métricas, parece indicar que el modelo con más parámetros es el que mejores resultados puede llegar a presentar. A diferencia de lo que en un principio daba a entender su gráfica de curvas de aprendizaje [Figura 8.15].

De este modo, tras evaluar los modelos con el archivo *dev_text.tsv* y generar el TSV correspondiente, Codalab nos devolvió los siguientes resultados:

Modelo	Precisión	Recall	F1-Score
GLiNER Small	0.593487	0.423856	0.494530
GLiNER Medium	0.709351	0.631658	0.668254

Cuadro 8.8: Métricas de evaluación test de GLiNER

En consonancia con el análisis de métricas que acabamos de realizar, la versión Medium obtiene mejores resultados que la versión Small. Además, con una diferencia más que considerable.

A continuación, analizaremos los tipos y cantidad de entidades detectadas por ambos modelos [Figura 8.26].

Lo más fácil de ver, y de suponer, es que los tipos de entidades más detectados han sido Disease y Gene. Algo obvio pues son las categorías más predominantes y pueden aprender más a identificarlas.

Por otro lado, también apreciamos que de los ocho tipos de entidades disponibles, la versión Small solo ha podido detectar tres y la Medium, cinco. Es curioso que los demás tipos identificados, aparte de los dos mencionados, han sido aquellos cuya frecuencia en el conjunto de entrenamiento era bastante baja, sobre todo Transcript. Por ello, ahora comprobaremos si estas

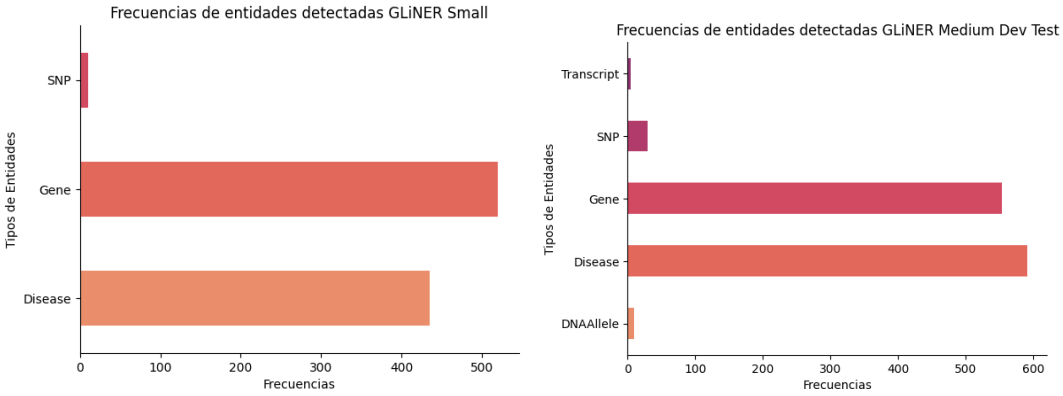

Figura 8.18: Frecuencias de tipos de entidades detectadas por ambas versiones de GLiNER.

predicciones han sido correctas.

En total, el modelo GLiNER Small ha detectado 965 entidades, de las cuales 368 han sido falsos positivos (el 38.13 %). El resto, 597, están representados en su matriz de confusión:

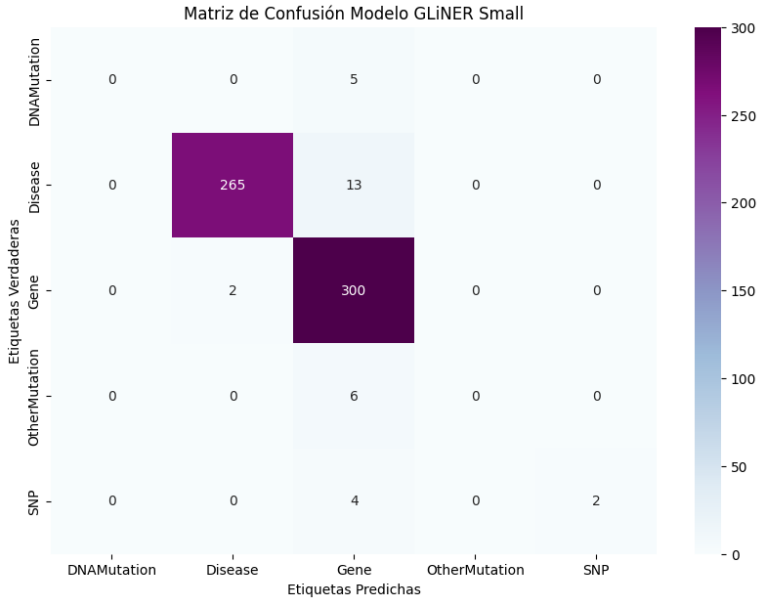

Figura 8.19: Matriz de confusión de GLiNER Small

Uno de los tipos de entidades detectados por esta versión fue SNP, el tercero con menos ejemplos. En concreto, el modelo clasificó a 10 entidades como tal. Si nos fijamos en la matriz de confusión, la diagonal correspon-

diente a SNP presenta dos entidades, lo que implica que el 20 % de esas predicciones han sido correctas. Dado que el resto de elementos de dicha columna toman valor cero, quiere decir que el resto de predicciones fueron falsos positivos. No es un valor relativamente alto, pero merece la pena destacarlo ya que debemos tener en cuenta que el modelo Small ha presentado resultados no demasiado buenos y que esta etiqueta es díficil de aprender.

En cuanto al modelo Medium, ha realizado 1188 predicciones, de las cuales 326 son falsos positivos (el 27.44 %). Esto nos deja con 862 entidades detectadas correctamente:

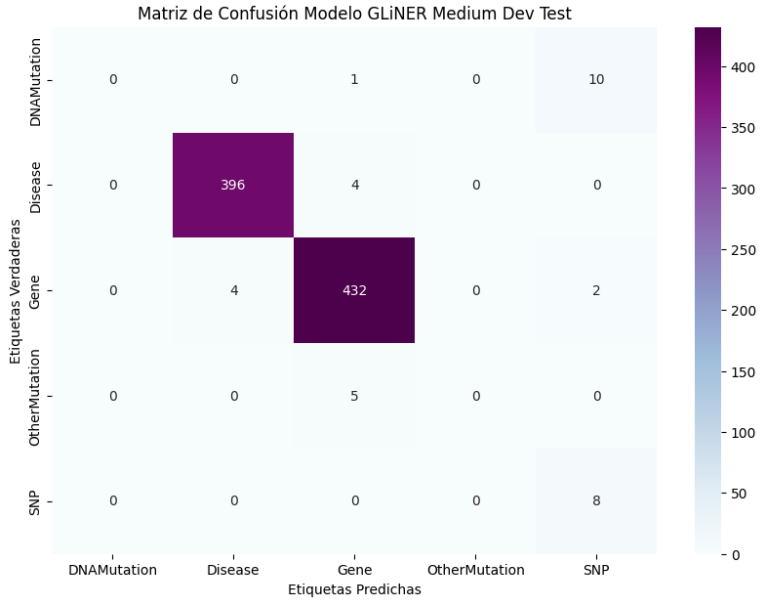

Figura 8.20: Matriz de confusión de GLiNER Medium usando *dev_text* como test.

Este modelo es el que sorprendentemente detectó el tipo de entidad Transcript (en concreto, 4 casos) y DNAAllele (con 9 casos). Sin embargo, si nos fijamos en la matriz, en las etiquetas predichas no aparecen ninguna de ellas. Esto significa que fueron falsos positivos, pues fueron detectados en entidades que no correspondían a ningún tipo. Aún así, los resultados son bastante buenos ya que ha clasificado correctamente muchas entidades, como vemos en su diagonal.

Como dato a destacar, podemos ver que el modelo tiende a confundir las entidades DNAMutation con SNP.

Por tanto, la comparación de los modelos Small y Medium de GLiNER aplicados sobre nuestro corpus ha mostrado que este último es el que ha obtenido los mejores resultados. En cierto modo podemos considerarlo nor-

mal. El número de parámetros, la cantidad de capas, su mayor capacidad para generar embeddings, mayor resiliencia al ruido debido a su mayor complejidad, entre otros factores, han dado lugar a que sea capaz de aprender mejor. Los resultados de test [Tabla 8.13] son una prueba fehaciente de ello, pues tanto en precisión, recall y F1-Score, hemos obtenido resultados con una diferencia de más del 10 %.

Aún así, ha merecido la pena el comparar estos dos modelos. De no hacerlo, no hubiéramos sabido a ciencia cierta si la diferencia entre ambos es significativa; ya que en caso de que los resultados fueran muy similares, la mejor opción sería emplear la versión Small debido a que conseguiríamos lo mismo con menos capacidad computacional. Ya hablamos de estas implicaciones en el capítulo del impacto medioambiental.

A nivel de tiempos, GLiNER Small ha tardado una hora y seis minutos en completar las 20 épocas. Por otro lado, GLiNER Medium ha tomado dos horas de entrenamiento, prácticamente el doble. Es lógico que tarde más pues tiene 43 millones de parámetros adicionales.

Por último, mostraremos una gráfica de distribución de longitudes de entidad de las predicciones de este modelo.

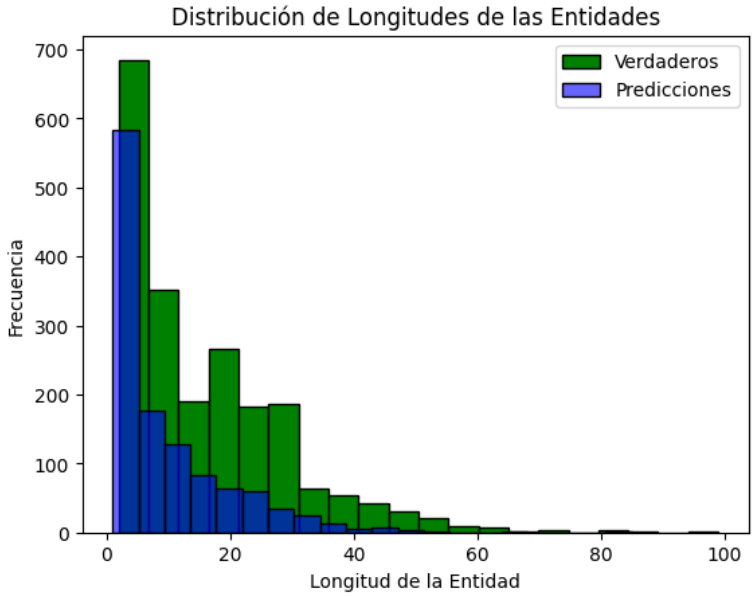

Figura 8.21: Gráfica de distribución de longitudes de GLiNER Medium Dev Test.

Como ya se ha visto, las etiquetas verdaderas suelen tener longitudes cortas. GLiNER Medium ha sido capaz de detectarlas y parece alinearse bien con ellas. Aunque puede que haya posibles subpredicciones debido a

que la frecuencia es ligeramente menor.

Conforme las entidades verdaderas comienzan a presentar más caracteres, alrededor de los 20, comenzamos a ver cierta discrepancia, lo cual sugiere que el modelo tiene más dificultad para identificarlas.

A continuación, vamos a realizar el último análisis. El mejor modelo GLiNER hasta ahora ha sido el Medium, por lo que lo volveremos a emplear. Sin embargo, usaremos el *dev_text* como el conjunto de validación. En principio, los resultados deberían ser mejores debido a las las ventajas de usar dichos datos al validar, como ya se vió.

Al mismo tiempo, realizaremos una comparación con GLiNER Medium y *dev_text* para validación pero empleando el archivo trainer.py durante el fine-tuning. En vez de la implementación de la que nos hemos servido hasta ahora. Comenzaremos explorando cual ha sido su evolucion durante el entrenamiento. Podemos verlo detalladamente [Tabla 8.9 y Tabla 8.10] así como gráficamente [Figura 8.22 y Figura 8.23].

Épocas	Training Loss	Precisión	Recall	F1-Score
1	47.03	56.37	42.83	48.67
2	132.14	55.63	45.61	50.12
3	124.34	56.63	50.83	53.58
4	92.80	56.59	50.61	53.44
5	106.25	58.49	52.50	55.33
6	125.62	58.57	53.95	56.17
7	156.42	59.09	56.40	57.71
8	97.44	63.26	57.84	60.43
9	141.27	67.38	59.29	63.08
10	99.15	70.84	59.18	64.48
11	73.22	73.48	59.18	65.56
12	123.53	74.37	59.07	65.84
13	83.66	74.20	59.18	65.84
14	58.75	74.48	59.73	66.30
15	98.15	73.93	59.29	65.80
16	68.41	74.10	59.51	66.01
17	80.67	73.90	59.51	65.93
18	59.87	73.69	59.51	65.85
19	91.52	73.49	59.51	65.77
20	59.57	73.49	59.51	65.77

Cuadro 8.9: Evolución del aprendizaje del modelo GLiNER Medium con *dev_text* para validación.

Épocas	Training Loss	Precisión	Recall	F1-Score
1	104.49	68.34	55.95	61.53
2	79.23	68.43	61.96	65.03
3	64.81	69.35	65.18	67.20
4	56.12	71.58	68.08	69.78
5	49.48	71.81	68.85	70.30
6	45.35	70.94	69.52	70.22
7	41.75	70.34	70.97	70.65
8	39.05	72.16	70.63	71.39
9	36.48	73.37	71.41	72.38
10	34.41	74.06	72.41	73.23
11	33.41	73.26	73.75	73.50
12	32.57	74.01	72.86	73.43
13	31.12	74.01	72.53	73.26
14	30.49	75.47	72.19	73.79
15	29.84	75.73	72.53	74.09
16	28.47	75.11	73.53	74.31
17	29.20	75.89	73.19	74.52
18	28.73	75.93	72.97	74.42
19	28.31	75.60	73.08	74.32
20	28.77	75.60	73.08	74.32

Cuadro 8.10: Evolución del aprendizaje del modelo GLiNER Medium con Trainer.py.

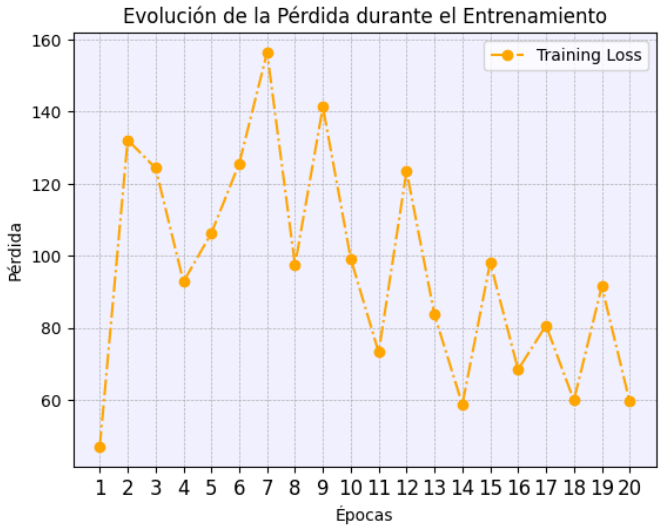

Figura 8.22: Curvas de aprendizaje para GLiNER Medium usando *dev_text* como validación.

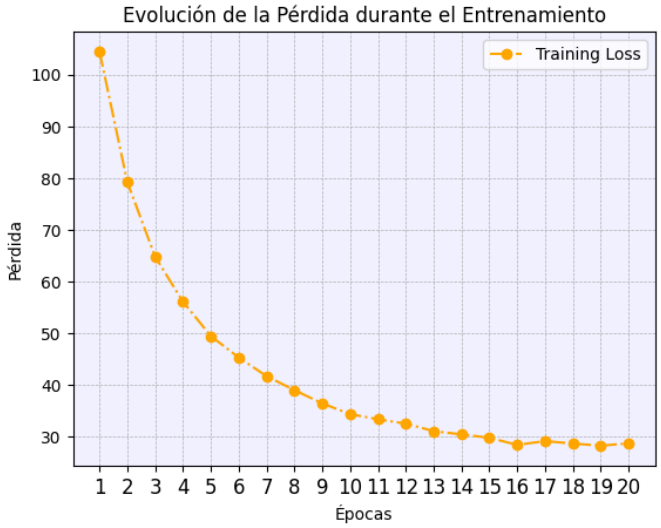

Figura 8.23: Curvas de aprendizaje para GLiNER Medium usando Trainer.py.

De nuevo, el primer modelo GLiNER Medium [Figura 8.22] presenta una alta variabilidad, así como presenta varios picos altos de pérdidas durante las primeras épocas. Los motivos con mucha probabilidad son los mismos que mencionamos previamente con las demás gráficas. Sin embargo, si la comparamos con la curva de aprendizaje de GLiNER Medium anterior [Figura 8.15], donde tomaba un conjunto de validación aleatorio, vemos que en este caso sí que encontramos un descenso general hacia las últimas épocas. Así, el modelo está aprendiendo mejor, aunque de manera algo inestable. Esto nos parece indicar que, de nuevo, el conjunto de validación empleado cobra mucha relevancia.

Por otro lado, si ahora nos fijamos en la curva de aprendizaje de GLiNER Medium junto con Trainer.py, vemos una diferencia abismal. La tendencia es, con diferencia, mucho más estable y consistente en el descenso. Este muestra una disminución gradual y constante a lo largo de las épocas, indicando que el modelo está aprendiendo progresivamente y ajustando los datos correctamente. Además, en las últimas épocas esta pérdida comienza a estabilizarse, lo que sugiere que el modelo se está acercando a una convergencia. Estos indicativos son bastante buenos, aunque al no disponer de la curva de validación, no sabemos si se está produciendo un ligero *overfitting* en las últimas épocas.

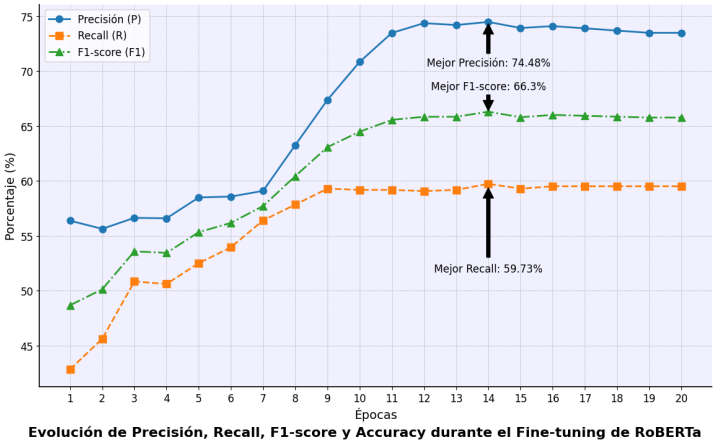

Figura 8.24: Evolución de métricas del modelo GLiNER Medium con *dev_text* para validación.

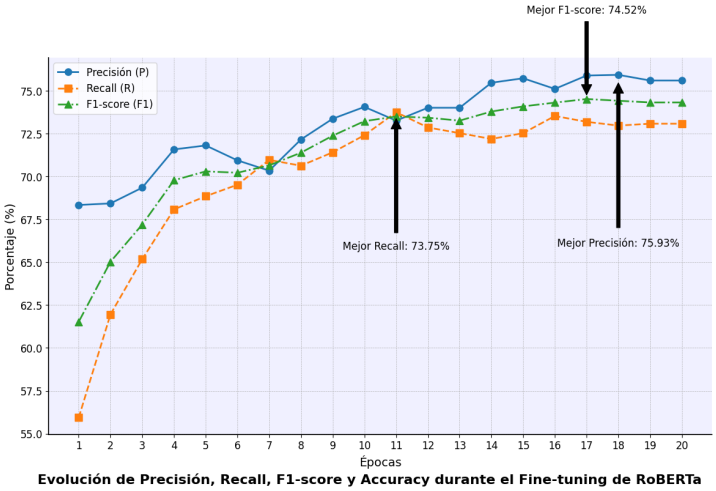

Figura 8.25: Evolución de métricas del modelo GLiNER Medium con Trainer.py.

Si comparamos la evolución de métricas de ambos modelos, es fácilmente apreciable como los resultados son bastante mejores cuando empleamos Trainer.py. Las tres métricas presentan una evolución más rápida y consistente hasta el punto de que para la época 3, el recall y el F1-Score presentan valores más elevados que los más altos obtenidos por el primer modelo, el cual necesitó 14 épocas para obtenerlos.

También encontramos más diferencias, pues el primer modelo presenta más fluctuaciones y tras comenzar a estabilizarse tras la 11ª época, los valores de precisión y recall no son cercanos, actuando este último como cuello de

botella para el F1-Score. Esto implica que el modelo es ciertamente conservador en las predicciones, pues solo detectará entidades si está muy seguro de ellas.

En cuanto al segundo, es bastante estable durante todo el entrenamiento. Con valores de precisión y recall similares que conllevan a una mejora significativa de los resultados. Además, cuando comienza a estabilizarse, ya posee valores elevados.

De este modo, tras evaluar los modelos con el archivo *test_text.tsv* y generar el TSV correspondiente, Codalab nos devolvió los siguientes resultados:

Modelo	Precisión	Recall	F1-Score
GLiNER Medium	0.822957	0.603998	0.696679
GLiNER Medium con Trainer.py	0.790643	0.796287	0.793455

Cuadro 8.11: Métricas de evaluación test de ambos versiones de GLiNER Medium

En consonancia con el análisis de métricas que acabamos de realizar, la versión Medium con Trainer.py obtiene mucho mejores resultados. Además, con una diferencia más que considerable.

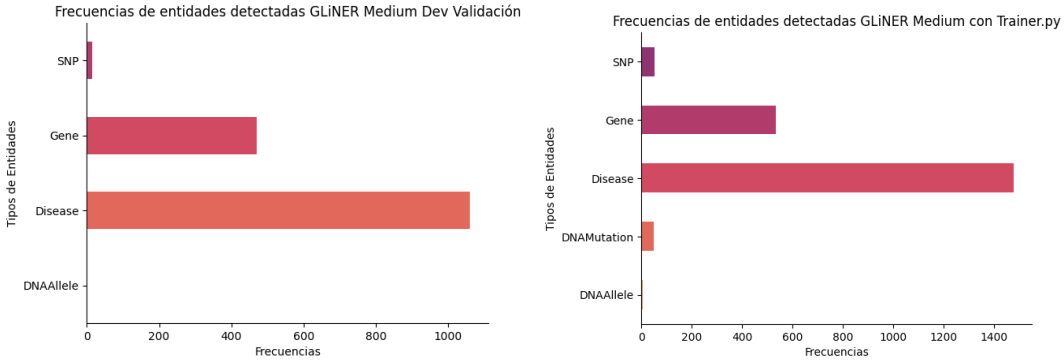

Figura 8.26: Frecuencias de tipos de entidades detectadas por ambas versiones de GLiNER Medium.

Ambos modelos vuelven a destacar por detectar en mayor medida aquellos tipos de entidades Disease y Gene, ya que son los que más ejemplos ofrecen para aprender y porque son las más comunes en los textos del corpus. De nuevo, vuelven a detectar entidades más complejas como SNP o DNAAllele. Aunque en muy poca cantidad.

El modelo respectivo a la primera gráfica realizó 1546 predicciones de las cuales 263 fueron falsos positivos (el 17.01 %). Así, se detectaron correctamente 1283 entidades:

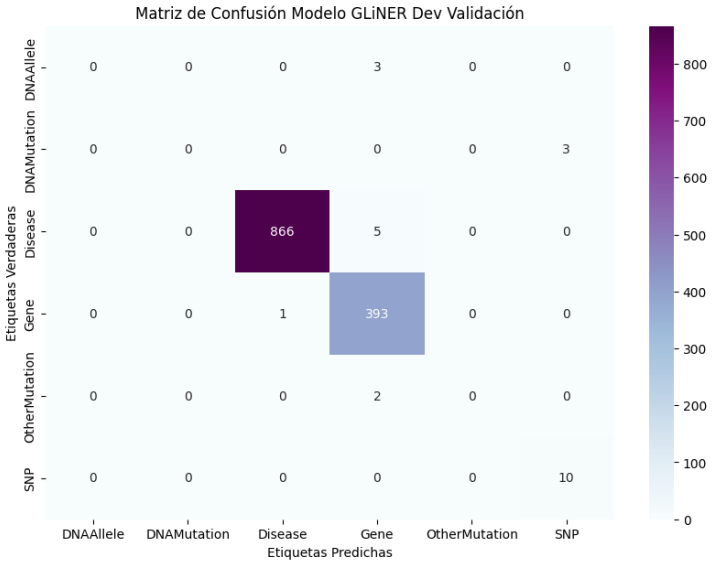

Figura 8.27: Matriz de confusión de GLiNER Medium usando *dev_text* como validación.

Los resultados de la matriz son bastante buenos, pues la mayoría de las 1283 entidades detectadas fueron clasificadas correctamente. En cuanto al tipo DNAAllele, podemos ver que la matriz no ofrece resultados, indicando que fueron falsos positivos. Por parte de SNP, se detectaron 14 entidades, 13 aparecen en la matriz y 10 fueron correctos por lo que el modelo ha sido capaz de aprender a diferenciarlos en cierto modo correctamente.

En cuanto al modelo Medium con Trainer.py, detectó 2118 entidades. Algo realmente bueno pues es el primer modelo GLiNER en detectar tal cantidad, muy similar a las 2101 entidades reales del conjunto test. De ellas, 426 han sido falsos positivos (el 20.11 %). Por tanto, 1692 fueron clasificadas correctamente. Representados en la siguiente matriz de confusióm [Figura 8.28].

Los resultados son realmente buenos pues si nos fijamos en la diagonal, encontramos unos valores realmente altos. Al igual que con el modelo anterior, ha sido capaz de aprender muy bien el tipo SNP y ha fallado con DNAAllele. Por otro lado, este modelo también presentó varias predicciones de DNAMutation (concretamente 78), una etiqueta más común, de las cuales vemos que 41 han sido correctas. Así, podemos decir que ha aprendido a distinguirlas de una manera relativamente correcta.

En cuanto a tiempos de ejecución de entrenamiento, el modelo GLiNER Medium con *dev_text* para validación ha empleado 1 hora y 40 minutos en ejecutarse. Sorprendentemente menos que la versión GLiNER Medium que lo empleó como test. El motivo principal sugiere ser que el aprendizaje

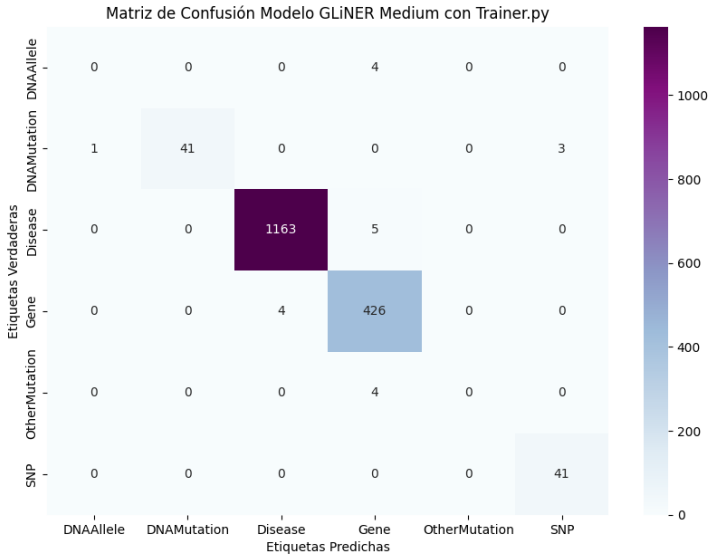

Figura 8.28: Matriz de confusión de GLiNER Medium usando Trainer.py.

fue más fácil, tal y como vimos anteriormente en su curva de aprendizaje [Figura 8.22]. En cuanto al modelo GLiNER Medium con Trainer.py, 40 minutos. Con una media de dos minutos por época.

Por último, mostraremos la gráfica de distribución de longitudes de entidad predichas:

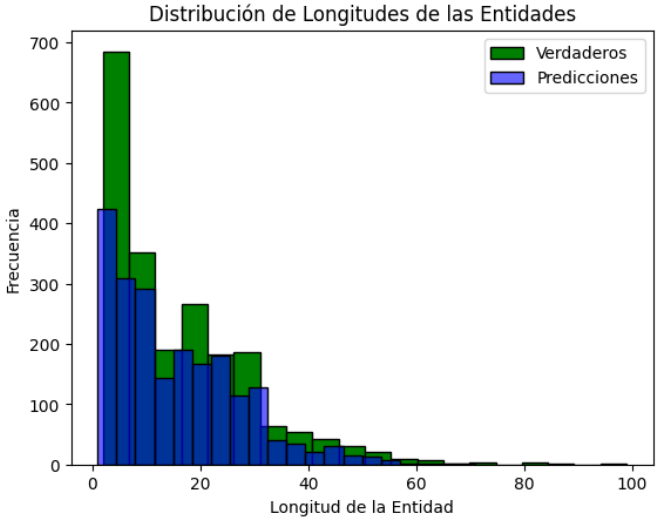

Figura 8.29: Gráfica de distribución de longitudes de GLiNER Medium con Trainer.py.

Brevemente, encontramos un caso similar al visto previamente. El modelo es capaz de identificar muy bien aquellas entidades cuya longitud es más corta y, además, en este caso muestra mejores resultados al detectar aquellas más largas.

Por tanto, de todos los modelos GLiNER que hemos analizado, podemos concluir que el mejor de todos ellos, con una amplia diferencia, ha sido GLiNER Medium empleando Trainer.py para el fine-tuning. Además, para ello ha empleado el menor tiempo de todos pues como ya mencionamos durante la implementación, se priorizaba mucho la eficiencia del mismo. Empleando técnicas de ejecución distribuida y paralela, entre otras. Un desglose de los resultados finales de todos ellos es el siguiente:

Modelo	Precisión	Recall	F1-Score
GLiNER Small con validación aleatoria del 10 %	0.593487	0.423856	0.494530
GLiNER Medium con validación aleatoria del 10 %	0.709351	0.631658	0.668254
GLiNER Medium con *dev_text* para validación	0.822957	0.603998	0.696679
GLiNER Medium con Trainer.py	0.790643	0.796287	0.793455

Cuadro 8.12: Rendimiento de los diversos modelos GLiNER implementados

Por tanto, con el mejor modelo obtenemos que el 79.06 % de las palabras etiquetadas como entidades, lo son realmente; el 79.62 % de las entidades reales en un caso clínico son capaces de ser detectadas correctamente; y como consecuencia, obtenemos un F1-Score del 79.34 % que indica un rendimiento robusto y un gran equilibrio entre las dos métricas previas.

Para justificar que obtener un valor tan cercano al 80 % de F1-Score es realmente prometedor cuando trabajamos en la disciplina médica y con entidades tan concretas como las nuestras, de nuevo, nos ayudaremos del paper de GLiNER [5]. En uno de sus apartados, mide cual es el rendimiento zero-shot de su versión Large, ChatGPT y UniNER-7B sobre un conjunto de 20 datasets para NER y los compara [Figura 8.30]:

En concreto, los que nos interesan son los datasets AnatEM, bc2gm, bc2chemd, bc5cdr, GENIA y ncbi. Todos están formados por datos de literatura biomédica. De estos, los datasets bc2gm, GENIA y ncbi presentan las mayores similitudes con nuestros datos. El dataset bc2gm se centra en la anotación de genes y proteínas, GENIA contiene anotaciones sobre términos relacionados con biología molecular, y ncbi está orientado hacia las enfermedades. Los mejores resultados para estos tres conjuntos fueron un F1-Score de 47.9 %, 55.5 %, y 61.9 %, respectivamente. De hecho, fueron obtenidos por GLiNER.

No hay que olvidar que estos conjuntos presentan mayor complejidad que el nuestro, de ahí que las métricas sean menores. El objetivo es mostrar que, a medida que el dominio se vuelve más específico y complejo, la detección

Dataset	ChatGPT	UniNER-7B	GLiNER-L
ACE05	26.6	**36.9**	27.3
AnatEM	30.7	25.1	**33.3**
bc2gm	40.2	46.2	**47.9**
bc4chemd	35.5	**47.9**	43.1
bc5cdr	52.4	**68.0**	66.4
Broad Tweeter	61.8	**67.9**	61.2
CoNLL03	52.5	**72.2**	64.6
FabNER	15.3	**24.8**	23.6
FindVehicle	10.5	22.2	**41.9**
GENIA	41.6	54.1	**55.5**
HarveyNER	11.6	18.2	**22.7**
MIT Movie	5.3	42.4	**57.2**
MIT Restaurant	32.8	31.7	**42.9**
MultiNERD	58.1	59.3	**59.7**
ncbi	42.1	60.4	**61.9**
OntoNotes	29.7	27.8	**32.2**
PolyglotNER	33.6	41.8	**42.9**
TweetNER7	40.1	**42.7**	41.4
WikiANN	52.0	55.4	**58.9**
WikiNeural	57.7	69.2	**71.8**
Average	36.5	45.7	**47.8**

Figura 8.30: Rendimiento de GLiNER, ChatGPT y UniNER sobre un conjunto de 20 datasets para NER.

de entidades se convierte en una tarea significativamente más exigente. Por ello, conseguir valores altos en métricas de rendimiento, como hemos logrado en nuestro caso, no siempre es posible y subraya la dificultad inherente a esta tarea.

8.4. Análisis comparativo de los modelos

En este punto ya tenemos una idea clara de cual ha sido el rendimiento de todos los modelos implementados en el proyecto. Por tanto, en este apartado realizaremos un breve análisis comparativo de todos ellos. Partiremos de aquellas implementaciones que mejores resultados hayan dado de cada uno de los tres tipos de modelos.

Modelo	Precisión	Recall	F1-Score
GPT-3.5	0.400000	0.186578	0.254463
RoBERTa	0.611556	0.654926	0.632498
GLiNER Medium	0.790643	0.796287	0.793455

Cuadro 8.13: Rendimiento de GPT, RoBERTa y GLiNER.

El peor modelo de todos ha sido GPT-3.5 con una amplia diferencia. Como se ha venido comentando a lo largo del proyecto, es normal dado

que no fue diseñado para tareas NER. Recordemos que es un modelo auto-regresivo, no auto-codificador (que son los empleados en NER). Si, además, la tarea implica detectar tipos de entidades tan específicos en el campo médico, la complejidad aumenta drásticamente.

Por otro lado, RoBERTa mejora significativamente el rendimiento con el anterior mencionado en todas las métricas. Esta arquitectura fue preentrenada sobre textos médicos y, aunque su tarea principal no era NER, se podía adaptar a partir de un fine-tuning. Los resultados han sido bastante mejores, mostrando que ha sabido capturar mejor las relaciones de contexto y las características lingüísticas de los casos clínicos.

GLiNER Medium ha sido el mejor de todos ellos, mostrando el mejor rendimiento en todas las métricas. Su arquitectura fue diseñada y entrenada específicamente para NER, motivo por el cual le permite capturar de manera mucho más efectiva las características más relevantes al detectar entidades.

Aún así, si hablamos de tiempos de ejecución, el más rápido ha sido GPT dado que no necesita un entrenamiento previo para comenzar la detección de entidades. En segundo lugar, RoBERTa con 45 segundos por época y, por último, GLiNER con 2 minutos por época. Estas características dependen del diseño interno de cada modelo, de su optimización, tamaño del conjunto de datos, hiperparámetros empleados, etc. Aunque en nuestros casos ya mencionamos que hemos intentados ser lo más justos posibles, por lo que el diseño de cada uno es la característica que más relevancia cobra.

A nivel de aprendizaje, GLiNER supera a RoBERTa. Como pudimos ver durante el análisis de curvas de aprendizaje. De hecho, RoBERTa no presentaba resultados de métricas durante la primera época [Tabla 8.3], mientras que GLiNER sí [Tabla 8.10]. Esto demuestra que este último modelo presenta más facilidad para aprender y, por lo tanto, para una posible convergencia en el futuro.

Por último, destacar que aquellos tipos de entidades cuya presencia en el corpus era limitada, como Transcript, DNAAllele y NucleotideChange-BaseChange, son los que más dificultad han presentado para la detección. Hasta el punto de haberse detectado o muy pocas veces o ninguna. Es normal ya que de por sí, no son tipos de entidades que aparezcan comúnmente en textos clínicos reales, por lo que si además el modelo no posee varios ejemplos para su aprendizaje, los resultados van a ser pobres en ese sentido.

Capítulo 9

Conclusiones

9.1. Conclusiones

En conclusión, este proyecto ha alcanzado los objetivos establecidos al inicio de su desarrollo. Para ello, se ha realizado una exhaustiva exploración en los campos de Procesamiento del Lenguaje Natural (NLP) y Reconocimiento de Entidades Nombradas (NER), destacando su prominencia y avances significativos en la actualidad.

En particular, se ha profundizado en el ámbito biomédico debido a su relevancia y aplicaciones prácticas, así como a los desafíos intrínsecos a su complejidad. La selección de un conjunto de datos reciente, caracterizado por etiquetas detalladas y específicas, ha enriquecido aún más el estudio.

Se procedió con la implementación y evaluación de tres modelos. Uno de ellos comúnmente conocido en el dominio NER, como lo es RoBERTa; y los otros dos más recientes, GPT-3.5 y GLiNER. En donde este último ni siquiera hace un año que fue lanzado, a fecha de junio de 2024.

Tras su entreno, se realizaron diversos experimentos y análisis detallados en donde el que mejor desempeño obtuvo fue GLiNER pues superó a los otros modelos en todas las métricas y con una amplia diferencia de más del 15 % con el segundo mejor, RoBERTa, lo cual es bastante cuando trabajamos en el campo clínico. Esto va en consonancia con el análisis que se realizó al comienzo del proyecto, confirmando que es un modelo realmente competitivo y prometedor al que deberemos prestar atención.

Además, se ha realizado una comparación aclaratoria con modelos generativos, evidenciando sus limitaciones cuando son aplicados en dominios para los cuales no fueron entrenados. Este análisis ha sido crucial para comprender mejor las capacidades y restricciones de estos enfoques, las cuales a nivel popular pareciera a día de hoy que no tienen.

En resumen, este proyecto ha contribuido significativamente al conocimiento y comprensión de la aplicación de modelos NER en un contexto real y que incumbe a todos como lo es la medicina. Además, los resultados obtenidos proporcionan información valiosa para la mejora de la identificación y extracción de entidades médicas de cara a realizar diagnósticos lo más precisos posibles y consecuentemente, aplicar los tratamientos adecuados.

9.2. Trabajos futuros

En el marco de este proyecto, se han identificado varias direcciones prometedoras para futuras investigaciones y desarrollos. En primer lugar, se podría proponer el uso de modelos GPT más recientes, como GPT-4 o GPT-4o, y comparar sus resultados con los obtenidos por los demás modelos para comprobar si podrían llegar a ser más competitivos. Asimismo, también se podría realizar una investigación más profunda en el campo de Prompt Engineering, empleando técnicas más avanzadas y específicas.

Con respecto a RoBERTa, este es solo uno de los múltiples modelos basados en BERT. Sería beneficioso explorar modelos más avanzados, o aquellos diseñados específicamente para el dominio médico, como BioBERT, ClinicalBERT o PubMedBERT, que no se utilizaron en este proyecto para facilitar una comparación con un modelo BERT más clásico.

En cuanto a GLiNER, podría plantearse el uso de su versión Medium Multilingüe o incluso la Large, la cual no hemos podido emplear por limitaciones de Colab. Es probable que esta última obtenga resultados superiores a los actuales. Además, dado que GLiNER está estableciendo nuevos estándares en el dominio NER, es recomendable seguir de cerca sus futuras mejoras y actualizaciones, así como la posible aparición de modelos derivados.

Otra opción sería continuar trabajando con estos mismos modelos, profundizando en la optimización de hiperparámetros para encontrar configuraciones que pudieran proporcionar mejores resultados.

Bibliografía

[1] Saswat Sarangi and Pankaj Sharma. *Artificial intelligence: evolution, ethics and public policy*. Routledge India, 2018.

[2] Anne Kao and Steve R Poteet. *Natural language processing and text mining*. Springer Science & Business Media, 2007.

[3] Juan Dempere, Kennedy Modugu, Allam Hesham, and Lakshmana Kumar Ramasamy. The impact of chatgpt on higher education. In *Frontiers in Education*, volume 8, page 1206936. Frontiers Media SA, 2023.

[4] Behrang Mohit. Named entity recognition. In *Natural language processing of semitic languages*, pages 221–245. Springer, 2014.

[5] Urchade Zaratiana, Nadi Tomeh, Pierre Holat, and Thierry Charnois. Gliner: Generalist model for named entity recognition using bidirectional transformer, 2023.

[6] A Vaswani, N Shazeer, N Parmar, J Uszkoreit, L Jones, AN Gomez, and I Polosukhin. Attention is all you need. paper presented at: Advances in neural information processing systems. In *30 (nips 2017)*. 2017.

[7] Diksha Khurana, Aditya Koli, Kiran Khatter, and Sukhdev Singh. Natural language processing: State of the art, current trends and challenges. *Multimedia tools and applications*, 82(3):3713–3744, 2023.

[8] Carol Friedman, Thomas C Rindflesch, and Milton Corn. Natural language processing: state of the art and prospects for significant progress, a workshop sponsored by the national library of medicine. *Journal of biomedical informatics*, 46(5):765–773, 2013.

[9] Bo Guo, Huaming Liu, and Lei Niu. Integration of natural and deep artificial cognitive models in medical images: Bert-based ner and relation extraction for electronic medical records. *Frontiers in Neuroscience*, 17:1266771, 2023.

[10] Nalini Chintalapudi, Gopi Battineni, Marzio Di Canio, Getu Gamo Sagaro, and Francesco Amenta. Text mining with sentiment analysis on

seafarers' medical documents. *International Journal of Information Management Data Insights*, 1(1):100005, 2021.

[11] Chung-Chi Huang and Zhiyong Lu. Community challenges in biomedical text mining over 10 years: success, failure and the future. *Briefings in bioinformatics*, 17(1):132–144, 2016.

[12] Kannan Nova. Generative ai in healthcare: advancements in electronic health records, facilitating medical languages, and personalized patient care. *Journal of Advanced Analytics in Healthcare Management*, 7(1):115–131, 2023.

[13] Yanis Labrak, Adrien Bazoge, Emmanuel Morin, Pierre-Antoine Gourraud, Mickael Rouvier, and Richard Dufour. Biomistral: A collection of open-source pretrained large language models for medical domains, 2024.

[14] Jasmine Chiat Ling Ong, Shelley Yin-Hsi Chang, Wasswa William, Atul J Butte, Nigam H Shah, Lita Sui Tjien Chew, Nan Liu, Finale Doshi-Velez, Wei Lu, Julian Savulescu, et al. Ethical and regulatory challenges of large language models in medicine. *The Lancet Digital Health*, 2024.

[15] N. D. Andreev. The intermediary language as the focal point of machine translation. In A. D. Booth, editor, *Machine translation*, pages 3–27. North Holland Publishing Company, Amsterdam, 1967.

[16] W. J. Hutchins. *Machine translation: past, present, future.* Ellis Horwood, Chichester, 1986.

[17] B. F. Green Jr, A. K. Wolf, C. Chomsky, and K. Laughery. Baseball: an automatic question-answerer. In *Papers presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, pages 219–224. ACM, 1961.

[18] William A. Woods. Semantics and quantification in natural language question answering. *Advances in Computers*, 17:1–87, 1978.

[19] Hiyan Alshawi. *The Core Language Engine.* MIT Press, 1992.

[20] Hans Kamp and Uwe Reyle. Tense and aspect. In *From Discourse to Logic*, pages 483–689. Springer Netherlands, 1993.

[21] Philip R. Cohen, Jerry Morgan, and Andrew M. Ramsay. Intention in communication. *American Journal of Psychology*, 104(4), 2002.

[22] Kathleen R. McKeown. *Text Generation.* Cambridge University Press, Cambridge, 1985.

[23] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*, volume 999. MIT Press, Cambridge, 1999.

[24] Inderjeet Mani and Mark T. Maybury, editors. *Advances in Automatic Text Summarization*, volume 293. MIT Press, Cambridge, MA, 1999.

[25] Yoshua Bengio, Rejean Ducharme, and Pascal Vincent. A neural probabilistic language model. In *Proceedings of the Neural Information Processing Systems (NIPS)*, 2001.

[26] Ronan Collobert and Jason Weston. A unified architecture for natural language processing. In *Proceedings of the 25th International Conference on Machine Learning*, pages 160–167, 2008.

[27] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, 2013.

[28] Charles Thomas. Recurrent neural networks and natural language processing, 2019.

[29] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[30] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[31] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[32] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[33] David W. Otter, John R. Medina, and Jugal K. Kalita. A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32(2):604–624, 2020.

[34] I. Zeroual, A. Lakhouaja, and R. Belahbib. Towards a standard part of speech tagset for the arabic language. *Journal of King Saud University-Computer and Information Sciences*, 29(2):171–178, 2017.

[35] N. Tapaswi and S. Jain. Treebank based deep grammar acquisition and part-of-speech tagging for sanskrit sentences. In *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*, pages 1–4. IEEE, 2012.

[36] P. Ranjan and H. V. S. S. A. Basu. Part of speech tagging and local word grouping techniques for natural language parsing in hindi. In *Proceedings of the 1st International Conference on Natural Language Processing (ICON 2003)*, 2003.

[37] Ryan McDonald, Koby Crammer, and Fernando Pereira. Flexible text segmentation with structured multilabel classification. In *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 987–994. Association for Computational Linguistics, 2005.

[38] Fei Sha and Fernando Pereira. Shallow parsing with conditional random fields. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 134–141. Association for Computational Linguistics, 2003.

[39] Xia Sun, Louis-Philippe Morency, Daisuke Okanohara, and Jun'ichi Tsujii. Modeling latent-dynamic in shallow parsing: a latent conditional model with improved inference. In *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*, pages 841–848. Association for Computational Linguistics, 2008.

[40] Alan Ritter, Sam Clark, and Oren Etzioni. Named entity recognition in tweets: an experimental study. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1524–1534. Association for Computational Linguistics, 2011.

[41] Martha Palmer, Daniel Gildea, and Paul Kingsbury. The proposition bank: an annotated corpus of semantic roles. *Computational Linguistics*, 31(1):71–106, 2005.

[42] Min Zhang and Juntao Li. A commentary of gpt-3 in mit technology review 2021. *Fundamental Research*, 1(6):831–833, 2021.

[43] Alexander Turchin, Stanislav Masharsky, and Marinka Zitnik. Comparison of bert implementations for natural language processing of narrative medical documents. *Informatics in Medicine Unlocked*, 36:101139, 2023.

[44] Cong Sun, Zhihao Yang, Lei Wang, Yin Zhang, Hongfei Lin, and Jian Wang. Biomedical named entity recognition using bert in the machine reading comprehension framework. *Journal of Biomedical Informatics*, 118:103799, 2021.

[45] Xi Yang, Jiang Bian, William R Hogan, and Yonghui Wu. Clinical concept extraction using transformers. *Journal of the American Medical Informatics Association*, 27(12):1935–1942, 2020.

[46] Yuqi Si, Jingqi Wang, Hua Xu, and Kirk Roberts. Enhancing clinical concept extraction with contextual embeddings. *Journal of the American Medical Informatics Association*, 26(11):1297–1304, 2019.

[47] Cheng Peng, Xi Yang, Kaleb E Smith, Zehao Yu, Aokun Chen, Jiang Bian, and Yonghui Wu. Model tuning or prompt tuning? a study of large language models for clinical concept and relation extraction. *Journal of Biomedical Informatics*, 153:104630, 2024.

[48] Peter Norvig and Stuart Russell. *Inteligencia Artificial: Un enfoque moderno*. Pearson, 2015.

[49] Douglas Appelt and et. al. Sri international fastus system muc-6 test results and analysis. In *Proceedings of the Sixth Message Understanding Conference (MUC-6)*, Columbia, 1995. NIST, Morgan Kaufmann.

[50] Lucja Iwanska, Marjorie Croll, Tae Yoon, and Michael Adams. Wayne state university: Description of the uno processing system as used for muc-6. In *Proceedings of the Sixth Message Understanding Conference (MUC-6)*, Columbia, 1995. NIST, Morgan Kaufmann.

[51] R. Morgan and et. al. University of durham: Description of the lolita system as used for muc-6. In *Proceedings of the Sixth Message Understanding Conference (MUC-6)*, Columbia, 1995. NIST, Morgan Kaufmann.

[52] Patrick Lewis, Myle Ott, Jingfei Du, and Veselin Stoyanov. Pretrained language models for biomedical and clinical tasks: understanding and extending the state-of-the-art. In *Proceedings of the 3rd clinical natural language processing workshop*, pages 146–157, 2020.

[53] Chih-Hsuan Wei, Alexis Allot, Kevin Riehle, Aleksandar Milosavljevic, and Zhiyong Lu. tmvar 3.0: an improved variant concept recognition and normalization tool. *Bioinformatics*, 38(18):4449–4455, September 2022.

[54] Mujeen Sung, Minbyul Jeong, Yonghwa Choi, Donghyeon Kim, Jinhyuk Lee, and Jaewoo Kang. Bern2: an advanced neural biomedical named entity recognition and normalization tool. *Bioinformatics*, 38(20):4837–4839, October 2022.

[55] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

[56] Alexandra Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power hungry processing: Watts driving the cost of ai deployment? *arXiv e-prints*, pages arXiv–2311, 2023.