Trabajo Fin de Máster: Generación Automática de Texto Fantástico en Español

Trabajo Fin de Máster

Francisco José Morón Reyes

Trabajo de investigación para el

Máster en Lenguajes y Sistemas Informáticos

Universidad Nacional de Educación a Distancia

Dirigido por el

Prof. Dr. Anselmo Peñas Padilla

Septiembre 2022

Agradecimientos

Al Prof. Dr. Anselmo Peñas Padilla por su implicación en el proyecto y su buen hacer a la hora de dirigir esta memoria.

Resumen

La generación automática de texto es un problema desafiante que plantea fuertes bases en las disciplinas del procesamiento del lenguaje natural e inteligencia artificial. En este trabajo analizaremos sus fortalezas, debilidades y desafíos así como su desarrollo en la actualidad. Existen multitud de enfoques para la generación automática de texto pero a día de hoy sigue siendo una tarea compleja de abordar, ya que cuenta con grandes dificultades aún por solucionar. A lo largo de estas páginas construiremos TolkienGen, un *dataset* en español que nos permitirá cotejar la efectividad de diferentes tipos de modelos de generación basados en redes neuronales. Evaluaremos y analizaremos los resultados con el objetivo de comparar los resultados y establecer con ellos un análisis de la situación actual.

Abstract

Automatic text generation is a challenging problem that has strong foundations in the disciplines of natural language processing and artificial intelligence. In this work we will analyze its strengths, weaknesses and challenges, as well as its current status. There are many approaches to automatic text generation, but today it is still a complex task to approach, since it has great difficulties yet to be solved. Throughout these pages we will build TolkienGen, a dataset in spanish that will allows us to compare the effectiveness of different types of generation models based on neural networks. We will evaluate and analyze the results in order to compare the results and establish an analysis of the current situation.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

La creación y narración de historias es un aspecto casi tan antiguo como el origen del propio Ser Humano. Desde tiempos inmemoriales hemos utilizado nuestra imaginación y buena parte de la realidad que nos rodeaba para componer diferentes crónicas y relatos, con tanto éxito que muchos nos han acompañado desde su creación.

A día de hoy, buena parte de nuestro entretenimiento se basa en el consumo de diferentes historias, ya sea vía literatura, cine, televisión, videojuegos u otros contenidos. Es imposible ignorar la importancia de la narración en nuestro día a día. De hecho, en muchos sentidos se ha convertido en símbolo de diferentes aspectos culturales de nuestra sociedad y algunos medios, como el cine, se asocian a diferentes industrias, como Hollywood y su estilo característico.

Sin embargo, a pesar de la obvia importancia que tienen las historias en nuestro día a día, es bastante complicado para nuestros ordenadores entenderse con ellas. El campo de la Inteligencia Computacional Narrativa no es algo nuevo, aunque sí ha experimentado un fuerte crecimiento en la última década debido a las importantes mejoras computacionales que han ido surgiendo. Aún así, a día de hoy todavía estamos lejos de lograr una comunicación fluida entre máquina y hombre que permita la creación de una historia coherente y cohesionada de considerable extensión.

Definamos; la Inteligencia Narrativa Computacional es la habilidad para crear, contar, entender y responder de forma adecuada a diferentes historias [1]. Aplicar esta inteligencia a un ordenador lo convierte en un mejor comunicador, pero sobre todo en un mejor entendedor, de tal forma que la interacción con la máquina mejora exponencialmente.

Dentro de las diferentes aplicaciones que podría tener un sistema de Narrativa Computacional el tema central de estas páginas será la Generación Automática de Texto, y más concretamente un campo específico dentro de este tema: La Generación Automática de Historias.

Es importante definir bien nuestra área de investigación, ya que nuestro tipo de generación textual tiene un objetivo diferente a los que podríamos encontrar en otras áreas de la Generación Automática de Texto.

Los humanos no necesitan entrenamiento para entender una historia, o para descifrar el objetivo de estas. Sin embargo, para las máquinas es una tarea difícil de asimilar, ya que en ella están presentes procesos mentales que para nosotros se realizan de forma automática e innata desde que somos muy pequeños. Una Historia escrita contiene eventos, acciones, personajes… y la máquina debe ser capaz de interpretar, crear y entrelazar estos elementos de forma correcta para generar una historia consistente.

1.1. Motivación

La motivación de estas páginas no es otra que la de estudiar en profundidad la generación de texto automática en el campo de la narrativa fantástica. El procesamiento del lenguaje natural y la inteligencia artificial están en continuo avance gracias a las nuevas tecnologías que se van actualizando. Gracias a esto cada vez se están haciendo mejoras en el campo de la generación textual y esto conduce a resultados impresionantes que poco a poco parecen prácticamente escritos por una persona real.

Desde el principio tuve claro que quería explicar y construir un proyecto que permitiese sobrevolar el estado de la situación actualmente, así como evolución y, con el apoyo de código escrito en Python, poder comprobar de primera mano cómo funciona la generación textual utilizando diferentes tecnologías de fácil acceso.

Por supuesto y como veremos en las páginas de este TFM, todo el proceso de estudio de generación textual que ha motivado este proyecto está fuertemente influenciado por el concepto de 'historia' ligado a la fantasía y, por supuesto, a la generación automática de texto.

Afrontar la creación de algo tan humano como una historia, escrito mediante texto, con la inteligencia artificial, es un tema apasionante. La escritura tiene mucho de ciencia pero más aún de arte, y siempre que concebimos el arte lo alejamos todo lo posible de la computación. La realidad es que los avances en computación hacen que la frontera entre lo que un humano puede hacer y lo que una máquina puede hacer quede cada vez mas difuminada. Crear una historia con su contexto, sus personajes, su evolución, tramas etc, es una obra más desafiante para una máquina que componer una partitura o pintar un cuadro y existen varias razones por las que esto es así. Intentaremos arrojar luz sobre estas razones mientras explicamos el estado actual del campo y nuestras indagaciones a nivel práctico y teórico.

Ya hemos ido adelantando algunas nociones y entraremos en detalles en profundidad sobre algunos conceptos mencionados más adelante, pero quería citarlo aquí para dejar claro que

era una motivación importante entender qué debería hacer una máquina para conseguir llegar a un nivel narrativo similar al de un humano. Es atrayente la idea de observar si el estado del arte es capaz de alcanzar un nivel óptimo de desarrollo en esta tarea, tanto como lo ha alcanzado en otras (véase clasificación, extracción de entidades…).

Por todo esto, pensemos en este trabajo como en un proyecto motivado por conocer y dar a conocer el estado del arte sobre la generación textual, sus entresijos y dificultades, y su pasado y su futuro.

1.2. Propuesta y objetivos

Nuestra propuesta consiste en construir un marco teórico y práctico de experimentación enfocado en la generación textual dentro de la narrativa fantástica.

Para construir el marco teórico se ha explorado el estado del arte en lo que a generación de texto se refiere, mencionando algunas particularidades de la escritura narrativa en general. Analizaremos brevemente la evolución del género, enfocándonos en diferentes algoritmos de aprendizaje automático y sus modelos neuronales asociados. Hablaremos de las ventajas y problemas observados durante todos estos años mediante el estudio de diferentes artículos enfocados a la generación textual y a la narrativa computacional, centrándonos también en diferentes modelos de generación centrados en esta tarea.

Para construir el marco práctico se ha elaborado un repositorio de Github que contará con todo el código necesario para construir un sistema *end-to-end* que se alimente de un conjunto de datos propios, genere un modelo y dicho modelo pueda servir para generar texto. Para ello nos serviremos de un conjunto de datos propio (TolkienGen) creado por nosotros y que consistirá en cinco libros del autor J.R.R. Tolkien (El Hobbit, El Señor de los Anillos, Las Dos Torres, El Retorno del Rey y Cuentos inconclusos de Númenor y la Tierra Media).

Nos centraremos en todas las fases del proceso por lo que hablaremos de la descarga de estos archivos, su almacenamiento en formato PDF y todas las fases propias del procesamiento del lenguaje natural: extracción del texto, pre-procesamiento de los datos, ingesta en los diferentes modelos neuronales y predicción.

El objetivo del trabajo será abordar la situación actual de la generación textual en el campo de la inteligencia artificial y su pasado, presente y su futuro apoyándose en dichos marcos. Para esto, abordaremos de forma práctica su pasado y su presente, valiéndonos de diferentes arquitecturas de aprendizaje automático que iremos viendo a lo largo de estas páginas. Para

el futuro esbozaremos un marco puramente teórico que nos permitirá hacernos una idea del mañana de la generación textual.

Nuestro objetivo se centrará en comprobar la coherencia y la cohesión del texto generado por diferentes modelos de generación, utilizando desde redes neuronales recurrentes en sus diferentes versiones (LSTM, BiLSTM, GRU) hasta la nueva y rompedora arquitectura de *Transformers* sirviéndonos de los modelos GPT.

Valiéndonos del conjunto de datos construido para este TFM y que también detallaremos mas adelante, compararemos la calidad del texto de un autor tan renombrado como J.R.R. Tolkien, con el que generarán nuestras diferentes redes neuronales. Con todo esto, concluiremos y esbozaremos la situación actual, sirviéndonos de los resultados obtenidos y las conclusiones halladas mediante el desarrollo de este trabajo.

Los objetivos de este trabajo son:

1. Construir TolkienGen, un *dataset* en español de libre disposición que permita evaluar la generación de lenguaje.
2. Utilizar este conjunto de datos para comparar la efectividad de diferentes tipos de redes neuronales en la generación de texto en español. Publicar datos y algoritmos en un repositorio de Github que nos permita visualizar los resultados de forma práctica.
3. Establecer un protocolo de evaluación y seleccionar las métricas que permitan cuantificar la calidad del texto generado.
4. Analizar los resultados producto de las diferentes alternativas propuestas y emitir conclusiones.

En particular, las preguntas de investigación que queremos responder son:

1. ¿Se confirma que también en español las aproximaciones basadas en transformers obtienen mejores resultados?
2. ¿Se corresponde la evaluación cuantitativa propuesta en el marco de evaluación con una exploración cualitativa de los resultados?
3. ¿En qué grado un fine-tuning con TolkienGen permite mejorar los resultados en términos de perplejidad?

Así pues, centraremos estas páginas en la creación de un sistema de aprendizaje automático que pretende construir texto coherente y cohesionado a fin de comprobar cuál ha sido la evolución de esta rama de la Inteligencia Artificial a lo largo de los años.

Nuestro trabajo también pretende servir de ayuda a futuras investigaciones en todo lo que se pueda. El conjunto de datos que crearemos se almacenará en el repositorio creado para este trabajo, así como también se mantendrá y mejorará el código que ha sido utilizado para cada una de las fases del proyecto.

Como todo el proyecto ha sido ideado en Español, y dada la poca información que se encuentra actualmente sobre hacer ajuste de modelos de *Transformers* al Español y lo confuso de los posibles conjuntos de datos a utilizar, el repositorio también incluirá toda la información recopilada sobre estas etapas, a fin de que pueda servir de apoyo en futuros proyectos.

1.3. Estructura del documento

A continuación describiremos brevemente los capítulos de este trabajo.

Capítulo 1. Introducción. Capítulo que servirá como puerta de entrada al proyecto. Explicaremos las principales motivaciones detrás de este TFM, los diferentes objetivos y la estructura que hemos seguido para elaborarlo.

Capítulo 2. Estado del arte. Describiremos el estado en cuestión de la rama del procesamiento del lenguaje natural y la inteligencia artificial que nos ocupa; la generación textual. Explicaremos los diferentes enfoques para la generación textual llegando hasta los novedosos mecanismos de atención, haciendo especial énfasis en los *embeddings contextuales* y en la tecnología de *Transformers*.

Capítulo 3. Creación de una colección de datos para la evaluación de la generación automática de texto en Español: TolkienGen. Profundizaremos en el conjunto de datos que hemos creado y los procesos que se han llevado a cabo con él. Explicaremos algunas de sus particularidades así como su creación, almacenamiento y otros aspectos de la misma.

Capítulo 4. Experimentación. Describiremos la metodología utilizada para evaluar el caso de estudio propuesto y hablaremos en profundidad de la implementación que hemos llevado a cabo a nivel de código. También presentaremos los resultados que hemos obtenido con nuestros experimentos.

Capítulo 5. Conclusiones y trabajo futuro. Hablaremos de las dificultades experimentadas durante el proceso de elaboración del trabajo. Aprovecharemos para analizar en profundidad los resultados obtenidos en el apartado anterior y retomar los temas de coherencia y cohesión

textual. Recopilaremos la información obtenida a lo largo del TFM y concluiremos el trabajo.

Capítulo 2

Estado del arte

Antes de entrar a la generación automática de historias de forma específica vamos a introducir brevemente el problema de la generación textual. Pensemos en esta rama del procesamiento del lenguaje natural como aquella que se basa en, dada un determinada entrada textual, mediante el uso de un modelo de generación, devolver una salida textual que consiste en una serie de palabras generadas de forma automática por un modelo entrenado [3].

La secuencia con la que trabajamos siempre estará compuesta de una serie de palabras de entrada que podemos denominar como *tokens* de entrada, a los que se le añadirá al final una palabra predicha por el modelo, que denominaremos *token* predicho. La nueva entrada será formada por esta estructura de *tokens* de entrada mas *token* predicho. De esta forma, la secuencia que va recibiendo la red neuronal siempre estará formada por el conjunto de *tokens* de entrada más el *token* predicho, haciendo que cada vez la secuencia de entrada aumente de tamaño al estar siempre compuesta por la combinación resultante del *input* textual más el *token* predicho.

Utilizando esta estructura es posible aplicar la generación textual a multitud de posibilidades. Es posible utilizarla para sugerir búsquedas en un buscador web, para generar texto de respuesta en un *chatbot* o simplemente para generación textual con algún motivo concreto. Por ejemplo, es posible utilizar la generación textual para generar recetas a partir de una lista de ingredientes [20].

Existen muchas consideraciones a tener en cuenta a la hora de generar texto, y estas consideraciones varían en función de la tarea en la que nos estemos centrando [20] [24]. Por ejemplo, para generar texto de respuesta en un asistente virtual debemos tener en cuenta el texto que nos escribe el usuario y, en base a este, generar la respuesta más adecuada para solucionar su problema.

Imaginemos también que queremos crear un modelo de generación textual para un videojuego de tal forma que cada vez que alguien pregunte algo a alguno de los personajes no jugables, este responda algo generado de forma automática. En este sentido deberíamos tener en cuenta no sólo el escribir una respuesta correcta desde el punto de vista gramatical, sino también escribir una respuesta correcta que tenga sentido dentro del contexto que se está jugando [24].

En nuestro caso, pondremos el enfoque sobre la Generación automática de historias, proceso que utiliza un sistema computacional para crear historias escritas mediante un modelo de aprendizaje automático [1] [3]. Dicho proceso requiere un entendimiento bastante profundo de los conceptos, eventos y entidades a representar. Conocimiento del que normalmente se encarga el procesamiento del lenguaje natural [2]. Dentro de este último es una tarea catalogada dentro de la generación de lenguaje natural (NLG, por sus siglas en inglés *Natural Language Generation*).

Uno de estos sistemas se define como capaz de comprender una historia si, dado una determinada entrada textual es capaz de leer y entender preguntas sobre el texto [2]. No es algo que debamos confundir con un sistema de Pregunta-Respuesta. Interactuar con preguntas y respuestas sobre un contexto narrativo se considera una tarea más desafiante que responder a preguntas basadas en hechos. De un Sistema de Generación Textual se espera una mejor comprensión y representación del conocimiento contenido en una narración determinada.

Generar una historia es más que simplemente predecir palabras de forma automática. Cuando estamos generando una historia no solo estamos prediciendo *tokens*, sino que también debemos construir eventos, actores y palabras relacionadas e interconectadas entre sí. Este es el aspecto más desafiante de nuestro problema y, como veremos más adelante, también uno de los más problemáticos a la hora de intentar configurarlos.

Un aspecto importante a tener en cuenta es el objetivo de nuestro sistema; entretener. Con la investigación sobre la narrativa computacional y sus aplicaciones buscamos la creación de un generador de historias cuyo objetivo final es componer una obra textual con sentido, que sea comprensible para un determinado público y que, además, les entretenga.

La naturaleza del objeto que tenemos por objetivo crear (una historia) tiene una serie de características que debemos tener en cuenta. Una historia suele generar una reacción en las personas que la reciben, y muchas veces dicha reacción no está exenta de emociones fuertes. Es importante analizar cómo va a interpretar el humano lo que estamos generando, ya no solo para crear un contenido lógico, también para no crear un contenido ofensivo que pueda ser perjudicial para el usuario [1].

Tengamos en cuenta otra serie de consideraciones previas para diseñar un Generador de Historias.

Los humanos escriben o cuentan historias para que sean consumidas por otros humanos. Este proceso involucra una serie de procesos mentales que son bastante complicados de hacer

entender a una máquina. Lo que para nosotros puede resultar obvio es todo un desafío para la computación en este contexto. La inteligencia narrativa tiene mucho de razonamiento, sentido común y experiencias que el narrador intenta plasmar en su historia con, generalmente, el objetivo de contar algo o hacer llegar una idea.

Este tipo de finalidad es algo que deberíamos de poder imprimir a nuestra máquina, a fin de que la historia quede como algo natural.

Otra consideración importante es la inclusión de figuras retóricas como la metáfora o la ironía. Este tipo de sentidos son a veces incluso difíciles de detectar por un humano. A día de hoy, la comprensión por parte de las máquinas no es la mejor, pero se han realizado intentos de usar metáforas y otros elementos en generación automática de historias [1]. A pesar de no estar demasiado pulido, este tipo de elementos cada vez son más frecuentes y se están haciendo grandes avances en este sentido.

Otro concepto importante es el de la creatividad, y a día de hoy pocas cosas se nos suelen ocurrir más alejadas de la creatividad que una máquina.

La creatividad es un concepto asociado al Ser Humano y su cultura, un concepto que muchas veces nos es difícil de definir. Las historias son creativas, prácticamente no hay dos iguales y buscan entretener. Aquí la creatividad es un concepto muy importante.

Sin embargo, es imposible pensar en la generación textual sin unirla al concepto de creatividad porque como comentábamos anteriormente, la literatura y la creatividad son dos conceptos que van de la mano. Más adelante exploraremos este tema enlazándolo con la generación automática de música o imágenes, sin embargo merece aquí una mención importante la generación automática de poesía, por la asociación entre ambas disciplinas. Existen modelos enfocados a la generación automática de poesía [25] que tienen en cuenta la métrica y la rima, guardando la forma poética de tal forma que no queda un texto simple, sino una poesía construida con sentido. Es algo fundamentalmente artístico que intenta ser imitado por un modelo de generación textual, lo que nos da una idea del rumbo que puede tomar la disciplina y de los esfuerzos y avances que se van detectando en ella.

2.1. Enfoques para la Generación Automática de Texto

El problema de la generación de texto es desafiante. Tengamos en cuenta que un sistema de narración computacional debe componer un texto con sentido, y no una simple palabra o frase como respuesta. El resultado debe tener un sentido global y no perder de vista el contexto del texto generado en todo su conjunto.

La gramática es bastante compleja incluso para una persona, acostumbrada a lidiar con ella. Por esto han existido multitud de enfoques destinados a facilitar la tarea de generación textual. Primero encontramos enfoques basados en reglas, que con el paso del tiempo han evolucionado para llegar a las redes neuronales recurrentes [3] y estas han acabado dando paso a la revolucionaria tecnología de *Transformers*.

En la actualidad existen muchos y variados enfoques a la hora de construir una historia. Definir el tipo de tecnología y enfoque a utilizar es una tarea igualmente desafiante pues han sido muchas las alternativas propuestas para generar un texto de forma automática mediante la computación.

Para escribir las líneas de este documento hemos decidido prestar atención principalmente a aquellos mecanismos de generación de texto que usen Atención [4]. Sin embargo, también centraremos nuestra atención en el uso de modelos de generación de texto que emplean arquitecturas de aprendizaje automático centradas en redes neuronales recurrentes ya que consideramos que es imposible entender la evolución del género y su estado actual sin prestar atención a este tipo de modelos neuronales.

Es posible pensar que la llegada de los mecanismos de atención ha significado el fin del uso de este tipo de algoritmos de aprendizaje automático, sin embargo es posible encontrar enfoques actuales para generación textual que utilizan redes neuronales recurrentes [22] y que tienen buenos resultados. De hecho, este tipo de redes siguen siendo bastante populares en problemas de traducción, sistemas de pregunta-respuesta, tareas de clasificación etc.

Tenemos la suerte de vivir una era dónde el avance tecnológico está a la orden del día. Este avance lleva aparejado un aumento de la potencia de computación que nos permite explorar nuevas soluciones aplicadas al procesamiento del lenguaje natural y a la inteligencia artificial que de otra forma no serían accesibles para todos los públicos. Decimos esto porque las redes neuronales recurrentes que más adelante detallaremos necesitan una menor capacidad computacional y esto tiene sus ventajas dependiendo del problema al que vayamos a enfrentarnos.

En contraposición, los modelos basados en la tecnología de atención utilizan potentes entrenamientos, de tal forma que el usuario descarga un modelo maduro que puede utilizar directamente y que, como comprobaremos a lo largo de estas páginas, a veces no es ni siquiera necesario entrenarlo, ya que puede servir con usarlo directamente. Estos modelos han sido entrenados con un coste computacional mucho mayor que el de los modelos de generación basados en redes neuronales.

Es importante poner el foco en cada aproximación, y como decíamos nosotros en esta memoria nos centraremos en dos: redes neuronales recurrentes y mecanismos de atención mediante el uso de *Transformers*. Cada una de estas aproximaciones tiene sus ventajas y desventajas e incluso en 2022 es posible encontrar una solución mejor que otra en función del problema a trabajar.

A pesar de lo dicho, este trabajo se centra principalmente en la generación textual aplicada a historias e intentaremos encontrar una explicación a por qué actualmente los modelos de generación de texto basados en mecanismos de atención suelen presentar mejores resultados que aquellos basados en aproximaciones de redes neuronales recurrentes.

2.2. Análisis y generación narrativa a través de los Embeddings Contextuales.

El análisis narrativo es un paso fundamental para un gran número de tareas del procesamiento del lenguaje natural. A través de estas páginas exploraremos el estado del arte en lo que a representación textual se refiere a través de los denominados como *embeddings contextuales* que nos ha traído la tecnología de Atención. También nos centraremos en otros aspectos influyentes a la hora de representar el texto en toda su extensión para que nuestros modelos de generación puedan mejorar su rendimiento.

La generación de historias incluye comúnmente el uso de secuencias de eventos, como por ejemplo el "esquema del restaurante" [10]. El uso de una secuencia nos permite predecir eventos posteriores. Al fin y al cabo el lector cuando lee una historia va interpretando los eventos que se producen en ella y entendiendo lo que va ocurriendo.

Podríamos decir pues que una historia se compone de una serie de eventos y que la representación de dichos eventos es una parte fundamental para este campo. La tarea adquiere una nueva dimensión a la hora de representar estos eventos en el texto manteniendo la cohesión entre ellos. Pensemos que no sólo tenemos que representar el sentido de cada palabra, sino que también debemos representar de alguna forma el mensaje de la historia en su totalidad. La naturaleza no estructurada de los datos textuales requiere de diferentes enfoques a la hora de realizar técnicas de aprendizaje no supervisado en lo que a representación semántica se refiere.

Esto enlaza directamente con una idea que iremos desarrollando poco a poco, y es que generar el texto añadiendo una estructura narrativa es una tarea realmente desafiante. El

'esquema del restaurante' que acabamos de mencionar puede servir como ejemplo para intentar clarificar este asunto.

Pensemos en una historia en la que una determinada persona llega a un restaurante para pedir comida, ya que tiene hambre. Este simple hecho desencadena una serie de eventos que surgirán prácticamente con obligatoriedad en la historia y que para nosotros ya están claros en nuestra mente.

En una secuencia lógica y muy probable de los acontecimientos esta persona primero se sentará en una mesa. Luego hablará con un camarero para pedirle su bebida y su comida y posteriormente le traerán esa bebida y esa comida. Finalmente, la persona comerá lo que ha pedido y tras ello saldrá del restaurante.

Esta cadena de eventos sería la lógica para una historia como la que hemos descrito, pero claro, el modelo de generación textual no es capaz de entender esta secuencia de eventos porque no tiene el mismo contexto que nosotros.

Esta situación tan cotidiana se da de forma continuada en nuestras vidas por lo que para nosotros es intuitivo imaginar lo que va a pasar una vez que definimos la secuencia 'una persona entra a un restaurante con hambre' [10]. Sin embargo, lo que puede parecernos simple entraña un gran problema de representación en la actualidad de esta disciplina.

Además existen otros inconvenientes que funcionan a un nivel más reducido y que los *embeddings contextuales* no pueden solucionar. Hablamos, por ejemplo, de la resolución de la correferencia [6] [7] [23].

Dentro de la secuencia que hemos descrito anteriormente existen una serie de relaciones que para nosotros vuelven a ser intuitivas pero que no lo son para la máquina. Por ejemplo, cuando decimos que una persona entra a un restaurante con hambre, se sienta a una mesa y un camarero viene a tomar nota de su comida, estamos estableciendo una serie de relaciones entre cada uno de los elementos del texto.

El hambre es de la persona en cuestión que entra al restaurante, la mesa en la que se ha sentado es del restaurante pero no del hombre, el camarero trabaja para ese restaurante y la comida que viene a apuntar será preparada por el restaurante para el hombre. Toda esta serie de relaciones que una vez mas son sencillas de establecer para nosotros se vuelven muy complicadas de representar a la hora de generar texto.

El problema de la falta de coherencia y cohesión en los textos generados automáticamente surge de una combinación de todo lo descrito anteriormente. Sobre todo cuando estamos generando texto en grandes cantidades e intentando mantener un hilo.

Si el texto que se está generando no tiene en cuenta la existencia de un hilo narrativo que conlleva una serie de eventos lógicos y, además, no es capaz de establecer relaciones entre los diferentes actores de la historia, el resultado puede ser correcto desde el punto de vista gramatical, pero nunca será un texto con sentido que alguien pueda leer y entender.

El objetivo central de este trabajo no pone el foco sobre la solución de estos problemas, aunque algunos de los artículos consultados sí que centran su atención en soluciones que podrían funcionar [23]. Sería difícil abordar este problema en este trabajo ya que a día de hoy no existe solución que permita contemplar todos estos elementos anteriormente descritos. Para nosotros queda más como reflexión que nos permitirá entender los resultados que mostraremos en el apartado de experimentación más adelante.

La evolución natural del género llegará a estos problemas de forma directa. Los *embeddings contextuales* han significado una revolución en lo que a representación por *token* o frase se refiere y también debemos resaltar sus múltiples virtudes en lo que a generación textual se refiere. Poder entender las relaciones entre los diferentes tokens del texto está a un paso muy cercano de poder entender las relaciones entre los diferentes actores del mismo.

2.3. Modelos Neuronales

Aunque hasta ahora no habíamos realizado una definición oficial, entendemos por aprendizaje automático a una de las ramas de la inteligencia artificial dedicada a la creación de sistemas capaces de aprender por sí mismos a realizar una tarea de forma automática.

Dentro de las muchas soluciones que existen en aprendizaje automático están las redes neuronales, uno de sus algoritmos clásicos. Este tipo de algoritmos toman como entrada una gran cantidad de datos, en nuestro caso texto, y aprende sobre ellos para generar una salida. Los elementos básicos de una red neuronal son las neuronas y las capas [5] [7].

Con el objetivo de analizar la evolución de esta disciplina presentamos a continuación un resumen de algunas de las soluciones mas comúnmente usadas en este campo. Para las tres primeras soluciones entraremos en el campo de las redes neuronales recurrentes, prestando especial atención a las LSTM, BiLSTM y GRU. En el segundo apartado nos centraremos en *Transformers* y mas concretamente en el modelo GPT-2.

2.3.1. Modelos de generación basados en Redes Neuronales Recurrentes

En apartados anteriores hemos detallado la importancia del contexto a la hora de trabajar con textos y el carácter secuencial del mismo. Las redes neuronales recurrentes se diseñaron específicamente para trabajar con secuencias de datos como por ejemplo textos o series temporales.

Funcionan de tal forma que, además de la entrada, hacen uso de los valores que se han emitido previamente, todo esto teniendo en cuenta el componente temporal. Por ejemplo, si estamos en el momento actual t, nuestra red neuronal recurrente utilizará datos anteriores pertenecientes a salidas en t - i, es decir, t-1, t-2, t-3, t-4 y así sucesivamente (esto es algo que se puede aplicar también a momentos posteriores, es decir t + i).

Es precisamente en esta forma de trabajo donde reside el carácter contextual de este tipo de redes neuronales. Se puede ver cómo que guardan la memoria sobre entradas anteriores, de tal forma que permite considerar lo visto anteriormente y así tomar decisiones sobre datos actuales o posteriores [7].

Dentro de la generación textual destacan diferentes tipos de algoritmos, pero nosotros nos centraremos en tres tipos de redes neuronales recurrentes: LSTM, BiLSTM y GRU.

Para entender por completo las LSTM debemos hablar del problema de desvanecimiento de gradiente. Las LSTM (*Long Short-Term Memory*) utilizan una celda de memoria que puede mantener dicha memoria o actualizarla, por lo que puede almacenar información a corto y largo plazo. Para esto utiliza tres puertas: una que puede controlar si mantiene u olvida con respecto a la celda anterior, otra que controla la información nueva a añadir a la celda y otra que decide qué información de la celda se utiliza para generar el estado oculto del instante actual.

Así, estas redes neuronales pueden mejorar los resultados de las redes neuronales recurrentes simples y solventar algunos de sus principales problemas, la pérdida de memoria a largo plazo.

Cabe mencionar su variante bidireccional (*Bidirectional Long Short-Term Memory*). Las redes neuronales recurrentes trabajan con unos datos de entrada que se leen de izquierda a derecha, orden de la secuencia que también funciona especialmente bien para el texto, al leerse este de izquierda a derecha. Pues bien, las versiones bidireccionales tienen en cuenta la secuencia de izquierda a derecha y de derecha a izquierda, en ambos sentidos [14]. Esto nos

permite aprovechar todos los datos de la secuencia, independientemente del carácter secuencial anteriormente descrito del texto.

Por último llegamos a las GRUs (*Gated Recurrent Units*), que utilizan la estructura de las LSTM anteriormente descritas pero simplificándolas sin incorporar una celda de memoria. Esta red neuronal se centra en mantener información sobre el contexto que pueda estar más alejado y utiliza dos puertas: una de actualización que nos permite controlar las partes del estado oculto que se van actualizando o manteniendo y otra de reseteo que controlará las partes del estado oculto previo utilizadas para el nuevo estado oculto.

2.3.2. Modelos de generación basados en *Transformers*

Las redes neuronales recurrentes presentan una serie de problemas que pretendían ser solucionados con el surgimiento de *Transformers* en 2017 [4]. En su origen, esta arquitectura pretendía enfocarse en tareas de traducción pero acabó demostrándose de gran utilidad en otras tareas, como por ejemplo la generación textual que nos ocupa.

La arquitectura completa de *Transformers* cuenta con un *encoder* y un *decoder.* El primero es el encargado de recibir la secuencia de entrada de datos, que en nuestro caso es un texto, y generará para ella una representación que será usada por el *decoder.* Este último utiliza la salida que le proporciona el *encoder* y otras entradas para generar una salida, que puede ser una predicción.

Todo esto va unido a los mecanismos de atención que permiten que cada *token* tenga una representación que no depende sólo de él mismo, sino de los demás *tokens* en la secuencia. Esto permite dar peso a los elementos de una secuencia textual en función de la importancia que queramos darle, y esta importancia quedará determinada por la tarea a realizar, pues no es lo mismo clasificación que generación de texto.

Además, una de las fortalezas de esta tecnología es su capacidad para ajustarse a la tarea a realizar y nuestro trabajo es testigo de ello. Por ejemplo, nosotros hemos utilizado dos modelos de GPT-2: *datificate/gpt-2-small-spanish* [15] y *DeepESP/gpt2-spanish* [16], que son versiones de GPT-2 pero ajustadas con la wikipedia en Español. Nosotros hemos hecho *fine tuning* sobre el modelo de *datificate* y sobre el modelo de *deepesp*, lo que significa que hemos entrenado de nuevo estos modelos pero utilizando nuestros datos para nuestra tarea concreta.

Como hemos mencionado anteriormente, el texto tiene un carácter eminentemente secuencial y en una frase una palabra ocupa una posición determinada, ordenándose unas tras otras de izquierda a derecha.

Retomemos un momento el tema de las redes neuronales recurrentes. Este tipo de redes utilizan una estrategia consistente en utilizar como entrada a la red neuronal la primera palabra, que internamente se procesará multiplicándose capa tras capa con los parámetros aprendidos de la red. La novedad con las redes neuronales recurrentes consiste en que la información procesada por la red será agregada a la nueva información que se tomará en cuenta en el siguiente paso de nuestra secuencia a la siguiente palabra. De esta forma hacemos un proceso que encadena el *output* de la red con el *input* del siguiente paso de tal forma que al analizar todas las palabras tendremos toda la información de nuestra secuencia procesada y analizada [5].

El procesamiento de una red neuronal recurrente tiene mucho sentido aplicado al ámbito textual ya que cuando leemos un texto procesamos cada palabra individualmente pero apoyándonos en el contexto de la frase y párrafo actuales. Leemos de manera secuencial, no damos un vistazo a todas las palabras y a partir de ahí extraemos las conclusiones sino que procesamos cada palabra, cada frase y cada párrafo para dar coherencia y cohesión al texto que estamos leyendo.

Precisamente esta lógica que tan bien funciona aplicada al texto es con la que se construían la mayoría de generadores de texto.

Sin embargo las redes neuronales recurrentes no están exentas de inconvenientes en lo que a generación textual se refiere. Uno de sus principales problemas se ve reflejado en el peso que la red neuronal concede a las primeras palabras respecto a las últimas agregadas, concediendo un peso menor a las primeras. Esto es un problema que se maximiza cuando tenemos en cuenta el carácter sintáctico de las frases.

Mi padre no ha podido venir a recogerme ya que ha perdido las llaves de su coche.

Recordemos que el procesamiento utilizado por las redes neuronales recurrentes es fundamentalmente secuencial y procesará cada palabra una detrás de otra. En la frase de arriba observamos una construcción sintáctica normal en castellano.

El sujeto se sitúa al principio de la frase y el predicado va a continuación. En este caso, la relación evidente entre *Mi padre* y *su coche* podría verse afectada por los problemas de memoria que hemos estado comentando anteriormente. Esto es lo que se conoce como problema de desvanecimiento de gradiente que comentábamos anteriormente. El gradiente de

los instantes de tiempo más lejanos se reduce en gran medida, de tal forma que no se modifican los pesos de estas palabras, y es difícil entrenar la red para establecer dependencias con las palabras más lejanas [14].

Este tipo de inconvenientes puede paliarse con el uso de los mecanismos de Atención [4] que hemos comentado anteriormente. Para entrar a este detalle primero debemos definir qué entiende la red neuronal por Palabra. Además, estos mecanismos mejoran inmensamente la calidad de la representación textual que usamos para entrenar los modelos de generación.

La computación procesa mejor una entrada de datos numéricos que una serie de letras por lo que para mejorar la comprensión textual de las redes neuronales las palabras suelen representarse como vectores multidimensionales que han ido evolucionando y perfeccionándose para capturar buena parte de la información semántica y sintáctica que alberga cada palabra y que de otra forma perderíamos. De hecho, la calidad de la representación de la información que introducimos a la red neuronal es determinante para producir un buen resultado, motivo por el cuál los vectores multidimensionales han sustituido paulatinamente a otro tipo de representaciones en procesamiento del lenguaje natural.

Entender el texto y por ende cada una de sus palabras es de vital importancia a la hora de generar un texto estructurado y con sentido. Estos vectores permiten entre sí operaciones matemáticas dentro de su espacio multidimensional. Por ejemplo, dos vectores con una dirección parecida deben representar conceptos cuyas palabras también sean parecidas mientras que, por el contrario, si dos vectores están muy alejados entre sí representarán conceptos cuyas palabras no tengan mucho que ver entre sí [7].

Retomando el ejemplo anterior y mediante los mecanismos de Atención seremos capaces de conectar cada palabra en la frase con todas las demás, permitiendo así situar cada una en su contexto de forma individual y colectiva.

Mi padre no ha podido venir a recogerme ya que ha perdido las llaves de su coche.

En esencia estaremos buscando la relación de todas las palabras con todas las palabras. *Mi* se relacionará con: *mi, padre, no, ha,* etc… *padre* se relacionará con: *mi, padre, no, ha* etc… permitiendo a la red neuronal establecer relaciones que de otra forma no seríamos capaces de establecer utilizando un procesamiento secuencial más simple como el de las redes neuronales recurrentes. *Padre* tiene una relación con *coche* pero ambas tienen también una fuerte relación con *llaves* y con el verbo *perdido*.

Las relaciones que se establecen entre cada una de las palabras de una frase y su comprensión es el punto fuerte de los mecanismos de atención que se han convertido rápidamente en el estado del arte en lo que a generación de texto se refiere [2].

Con el mecanismo de atención que usa esta arquitectura, la representación de cada *token* depende de los demás *tokens*, de tal forma que obtenemos muy buenas representaciones vectoriales del texto.

Esta gran mejora se consigue usando dos redes neuronales diferentes que con las palabras mencionadas arriba como entrada de datos aprenden a generar dos vectores distintos: un vector que nos servirá para identificar las propiedades de cada palabra y otro vector que servirá para describir las propiedades interesantes de cada palabra. Mediante el entrelazado y comparación de estos dos vectores es posible buscar coincidencias que nos permitan esclarecer las relaciones entre cada una de las palabras que va procesando nuestra red neuronal.

Estos conceptos quedan definidos como *vector query* y *vector key.* Cada palabra obtendrá su *vector query* y su *vector key* y se calculará la compatibilidad de un determinado *vector query* para todos los *vectores key* de una frase. El cálculo de compatibilidad se realiza utilizando el producto escalar de ambos vectores y es lo que se conoce como *attention vector* y será mayor cuanto más relevantes sea la relación entre las palabras de la frase [4].

Es con este enfoque con el que se crea una solución al problema que comentábamos al principio cuando explicamos que las redes neuronales recurrentes no eran capaces de establecer relaciones claras entre todos los elementos de una frase. La arquitectura descrita anteriormente es capaz de prestar atención adaptativa en función de cada palabra que va procesando, permitiéndonos calcular el vector de atención y utilizar un *output* que nos dará como resultado unos vectores de palabra cuya transformación recoge el contexto del resto de la frase.

Existen muchos modelos que utilizan la tecnología de *Transformers* pero en este apartado nos centraremos en los modelos GPT de *OpenAI.* Estos modelos de pre-entrenamiento generativo utilizan un modelado de lenguaje de probabilidad condicional basado en la arquitectura descrita anteriormente y los mecanismos de atención que tan importante son para el desarrollo de esta memoria.

OpenAI es la compañía detrás de estos modelos de lenguaje. Fue fundada a finales de 2015 por un grupo de emprendedores y empresas con el fin de asegurar un correcto desarrollo de la inteligencia artificial que beneficie a toda la humanidad.

No sería hasta Febrero de 2019 con la llegada de GPT-2 (*Generative Pre-Trained Transformers*) que *OpenAI* pasaría a estar en el centro del panorama del procesamiento del Lenguaje Natural. Llegaba hasta nosotros un modelo de lenguaje basado en *transformers* y entrenado con un 1.5B de parámetros en 10B de *tokens* [11]. El resultado fue un muy bueno a la hora de predecir la siguiente palabra en un texto basado en el contexto anterior, con resultados muy coherentes y plausibles.

Pero aún quedaba algo de mejora y esta llegaría en Junio de 2020 con el anuncio de GPT-3, un modelo de lenguaje 100 veces más grande que GPT-2, alcanzando los 175B de parámetros y 499B de *tokens* convirtiéndose en uno de los modelos de lenguaje más grandes construidos en la actualidad [11].

GPT-3 es un enorme modelo de lenguaje entrenado por *OpenAI* y especializado en generación de texto. El modelo salió a la luz en 2020 como evolución del GPT-2, modelo anterior de 2019 que a su vez era una versión mejorada de GPT.

La llegada de los modelos de *OpenAI* revolucionó la lingüística computacional tradicional al aportar nuevos modelos estadísticos de generación de lenguaje que ya se alejaban mucho de los antiguos modelos de generación de lenguaje basados en reglas [8].

Por desgracia, no podremos contar de primera mano con el uso de modelos de GPT-3 ya que no ha sido posible encontrar modelos que hayan sido ajustados en Español y, por problemas de capacidad computacional que mencionaremos más adelante, no ha sido posible para nosotros hacer este ajuste personalmente ni disponer de GPT-3 en toda su extensión.

A última hora se ha decidido incluir y mencionar el modelo de *Big Science* conocido como BLOOM [13] que ha salido recientemente y se presenta con una arquitectura similar a GPT-3 pero entrenado directamente en 46 lenguajes.

BLOOM cuenta con 176B de parámetros enfrentados todos de forma multilingüe. Un factor determinante de este modelo es su carácter *open source*, algo que lo diferencia sustancialmente de GPT-3. Por desgracia, BLOOM ha salido recientemente y su calidad no ha podido ser comprobada personalmente por lo que queda fuera del alcance de esta memoria.

Como comentábamos, los modelos de GPT no están exentos de problemas. Es cierto que la tecnología de atención dota de una capacidad de comprensión nunca vista a la red neuronal, permitiéndole contextualizar el texto que se va generando de una forma bastante coherente.

Sin embargo los modelos de *OpenAI* seguían y siguen distando de ser perfectos, aunque sí es cierto que GPT-3 ha traído una mejora considerable en lo que a la producción de escritura de calidad se refiere. Con esto nos referimos a que genera mejor texto con mayor frecuencia, aunque sigue presentando errores clamorosos en algunos apartados.

GPT hace cosas realmente bien y cosas mal. Es, por ejemplo, bastante bueno creando tramas realistas, recreando rasgos estilísticos de escritura y copiando los elementos temáticos claves de un autor en cortas cantidades de texto. Es además capaz de escribir en una variedad de géneros bastante amplia y tiene la capacidad de manejar grandes cantidades de información y sintetizarla con coherencia a la hora de generar texto.

Sin embargo, a pesar de generar un texto de calidad bastante asumible, los modelos de GPT han demostrado tener serios inconvenientes a la hora de generar un texto coherente y cohesionado a largo plazo. Nos referimos a un texto de grandes proporciones, ya que aunque en pequeñas frases y párrafos sí que se mantiene un sentido entre todos sus componentes, a medida que el texto generado aumenta de tamaño el resultado es peor en lo que a esto se refiere [21]. El modelo es bastante pobre a la hora de mantener argumentos coherentes o hilos narrativos durante cantidades más grandes de texto, mostrando graves problemas a la hora de mantener elementos naturales de la coherencia textual (como son por ejemplo la concordancia en género y número del sujeto con el predicado). Carece de la capacidad de escribir estructuras textuales complejas, prefiriendo casi siempre la utilización de reglas gramaticales simples y razonamientos básicos [8].

Ya poníamos el foco anteriormente en lo difícil que es solventar estos problemas teniendo en cuenta lo complejo que puede ser para el modelo entender cómo nosotros entendemos una cadena de eventos o una serie de relaciones en un texto. Sin embargo, una de las críticas que se le suele hacer a GPT-3 es su falta de comprensión. El modelo en sí no entiende lo que está escribiendo y eso dificulta enormemente que sea capaz de relacionar los elementos intrínsecos en un texto que hemos ido comentando.

A menudo el resultado presentado por estos modelos carece de sentido al no poder mantener una estructura sintáctica de calidad. Tarea que se vuelve aún más complicada a medida que el tamaño aumenta.

Recordemos que generar texto con sentido no completa la tarea de generar historias, ya que una historia debe tener coherencia y cohesión en todas sus partes; principio, nudo y desenlace. Es precisamente esto a lo que nos referimos. Para el modelo es difícil entender una estructura narrativa y, al carecer de ésta, el texto puede estar perfectamente escrito en todos sus aspectos pero no guardar ningún sentido en su resultado final.

Por supuesto y cómo comentábamos anteriormente esto no es un problema sencillo de responder. Es cierto que los *embeddings contextuales* de *Transformers* permiten representar con gran exactitud semántica el mapa de una oración o párrafo, interconectando cada uno de sus componentes (*tokens*) y dándole a la unidad un sentido que hasta ahora era difícil de alcanzar.

Sin embargo, una cosa es conseguir esta representación para cada palabra o frase, y otra muy diferente es dotar de sentido a todo el texto en sí. Esto último requiere de diferentes aproximaciones que buscan añadir contexto a la generación textual y que muchas veces usan incluso LSTM u otras medidas como TF-IDF para ayudar a la red neuronal a generar texto siempre dentro de unos patrones [21] [22].

2.4. Métricas para la evaluación en Generación de Lenguaje Natural

Existen multitud de métricas para analizar los modelos de *machine learning* y sus resultados. Es además un paso muy importante puesto que es necesario algún indicador que permita medir la calidad de la salida generada por nuestro modelo. Para un problema de aprendizaje automático es tan importante definir una métrica adecuada como cualquiera de sus otros componentes.

A continuación vamos a definir una serie de métricas usualmente utilizadas en procesamiento del lenguaje natural (y principalmente en generación de texto) a fin de obtener una visión general del tema.

Podemos clasificar las métricas de evaluación en dos apartados principales: evaluación extrínseca e intrínseca.

La evaluación extrínseca implica evaluar los modelos empleándolos en una tarea real, como por ejemplo sería la traducción automática, y observar sus resultados finales. Es una opción muy buena ya que nos permite ver directamente cómo se relaciona el modelo con la tarea que nos interesa.

La evaluación intrínseca se lleva a cabo utilizando una métrica que nos permita evaluar el modelo de lenguaje en sí, sin tener en cuenta las tareas específicas para las que se usará. En nuestro caso y como definiremos más en profundidad próximamente, hemos optado por la perplejidad, que es un método de evaluación intrínseco.

Es común utilizar la evaluación extrínseca, ya que con su uso se puede evaluar la calidad de un modelo para resolver un problema comercial determinado.

Este tipo de evaluaciones suelen ser más generales, sin embargo, los equipos de inteligencia artificial tienden a utilizar evaluaciones intrínsecas, que pueden ser utilizadas para determinar la calidad del resultado teniendo en cuenta los datos con los que se cuentan.

Dentro del procesamiento del lenguaje natural, la generación de texto tiene varios problemas a la hora de decidirse por una métrica adecuada para sus modelos. Debemos tener en cuenta que cada tarea de generación textual es diferente.

Por ejemplo, no es lo mismo medir la calidad de un sistema de diálogo que medir la de un sistema de resumen, al igual que tampoco podríamos aplicar sus criterios a un problema de traducción automática.

Aunque son muchas las métricas que existen para procesamiento del lenguaje natural, nosotros definiremos a continuación aquellas centradas en la generación textual.

2.4.1. BLEU (*Bilingual Evaluation Understudy Score*)

Es junto con Rouge una de las métricas de evaluación mas populares para analizar resultados de generación textual. BLEU se centra en la precisión que calcula la superposición de n-gramas entre el valor de referencia y los textos generados.

Un punto importante a tener en cuenta con BLEU es que penaliza bastante la brevedad, es decir, cuando el texto generado es pequeño en comparación con el texto de destino, los resultados son penalizados [26].

Se calcula generalmente a nivel de frases y se suele utilizar frecuentemente en tareas de traducción automáticas.

BLEU es una métrica que se puede calcular de forma rápida y práctica, al basarse principalmente en la superposición de n-gramas, por lo que su fácil implementación es una ventaja. Además, el algoritmo no tiene en cuenta qué idiomas se están utilizando, lo que le aporta gran versatilidad.

En contraposición, BLEU tiene problemas a la hora de interpretar correctamente traducciones. Al basarse en la comparación de oraciones generadas con oraciones de

referencia, BLEU puede pasar por alto traducciones que sean correctas aunque no empleen exactamente las mismas palabras que la frase original. Precisamente, este carácter comparativo que requiere de un corpus de referencia es uno de los motivos por los que no se la considera una buena métrica para un caso de uso enfocado puramente en generación textual sin referencias.

2.4.2. Rouge (*Recall Oriented Understudy for Gisting Evaluation)*

Rouge es otra de las métricas más utilizadas. Es bastante parecida a BLEU en su definición pero la diferencia radica en que BLEU está centrada en la precisión y Rouge más en la cobertura [27]. Existen diferentes métricas de evaluación disponibles dentro de Rouge:

- ROUGE-N (superposición de n-gramas, unigramas como ROUGE-1 o bigramas como ROUGE-2)
- ROUGE-L: Basada en la sub-secuencia común mas larga, teniendo en cuenta la similitud de estructura a nivel de oración de forma natural.
- ROUGE-S: Se centra en skip-ngramas dentro de las oraciones.

La mas comúnmente usada es ROUGE-N. Es bastante popular a la hora de evaluar traducciones automáticas y tareas de resumen automático.

Al igual que ocurre con BLEU, Rouge se basa en la coincidencia de n-gramas entre el texto generado por nuestro modelo y una referencia. Por lo tanto también tiene problemas a la hora de captar diferentes palabras con el mismo significado, debido a que mide coincidencias desde el punto sintáctico en lugar de semántico. Así pues, si tuviéramos dos secuencias que tuvieran el mismo significado pero usaran palabras diferentes, tendrán una puntuación Rouge bajo

2.4.2. METEOR (Metric for Evaluation of Translation with Explicit Ordering)

Si BLEU se basaba principalmente en la precisión léxica y REOUGE se centraba en la cobertura léxica, METEOR se basa en la medida F [29].

También se basa en el concepto de coincidencia de unigramas entre la generación producida por nuestro modelo y un conjunto de frases de referencia. METEOR calcula el *mapeo* uno a uno de las palabras entre textos generados y de referencia. Utiliza WordNet o

porterStemmer. Una vez que se han encontrado todas las coincidencias de unigramas generalizadas entre las dos cadenas de texto que se están comparando, METEOR calcula una puntuación para esta coincidencia usando la medida F. Usar WordNet o *porterStemmer* le permite tener en cuenta los sinónimos de las palabras.

2.4.3. Perplexity

La perplejidad es otra de las medidas comúnmente utilizadas para evaluar la eficacia de un modelo de generación textual. Para aplicar BLEU y ROUGE, por ejemplo, se suelen utilizar comparaciones con textos de referencia. Perplejidad, sin embargo, permite medir la calidad de lo escrito sin tener que comparar con textos de referencia.

Pensemos en un modelo de lenguaje de generación textual como en un modelo estadístico que va a asignar probabilidades a palabras y oraciones. En general lo que estamos intentando hacer es adivinar la siguiente palabra, en una oración, dadas todas las palabras anteriores, que podríamos denominar historial.

Pongamos un ejemplo. Pensemos en la siguiente frase:

Voy al supermercado a comprar ___

Si nos detenemos a pensar, nuestra mente asigna diferentes probabilidades a la siguiente palabra que pensamos que ocuparía el lugar inmediatamente a la derecha de comprar y, automáticamente, nos ofrece resultados lógicos, descartando aquellos que no tienen sentido. Así pues, es más común para nosotros pensar en la palabra comida que en la palabra helicópteros.

Es decir, P(comida | voy al supermercado a comprar) > P(helicópteros | voy al supermercado a comprar).

Esto también es aplicable a la probabilidad que nuestro modelo asigna a una oración completa W formada por la secuencia de palabras (w_1, w_2,...,w_N).

$$P(W) = P(w_1, w_2, \ldots, w_N)$$

De esta forma es posible crear una fórmula que permite asignar probabilidades más altas a oraciones que son reales y sintácticamente correctas. Sin embargo, un modelo de unigramas

como el propuesto sólo funcionaría a nivel de palabras individuales.

$$P(W) = P(w_1)P(w_2)\dots P(w_N)$$

Dónde las probabilidades individuales P(w_i) podrían, por ejemplo, ser estimadas en función de la frecuencia de las palabras en el corpus de entrenamiento.

Sin embargo, un modelo de n-gramas analizaría las palabras anteriores (n-1) para estimar la siguiente [7]. Por ejemplo, un modelo enfocado en trigramas miraría a las dos palabras anteriores, de tal forma que:

$$P(W) = P(w_1)P(w_2|w_1)P(w_3|w_2,w_1)\dots P(w_N|w_{N-1},w_{N-2})$$

Para nosotros, la definición de probabilidad que utilizaremos será la siguiente:

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1,w_2,\dots,w_N)}}$$

Queremos que nuestro modelo asigne probabilidades altas a oraciones que son reales y tienen un sentido semántico y sintáctico, de igual forma que querremos probabilidades bajas en oraciones falsas y que no tengan mucho sentido.

Teniendo esto en cuenta, podemos asumir que nuestro conjunto de datos contará con oraciones que son reales y correctas, de tal forma que nuestro modelo no se sorprenda al verlo (no quede perplejo), lo que significa que existe una buena comprensión del lenguaje y que tiene sentido lo que está escrito. Esto será usado como medida para entender qué es una frase correcta y qué no.

Un factor importante a tener en cuenta será la cantidad variable de oraciones y palabras que puede tener un conjunto de datos. No sólo cada conjunto de datos tendrá un número diferente de oraciones, sino que cada oración tendrá un número diferente de palabras.

Agregar más oraciones y palabras repercutirá en nuestra perplejidad ya que un conjunto de pruebas más grande tiende a dar más problemas que uno pequeño.

Esto puede controlarse mediante la normalización, es decir, normalizando el número total de palabras haciendo una media.

Teniendo esto en cuenta, podemos interpretar la ecuación de la perplejidad como la probabilidad inversa del conjunto de prueba, normalizada por el número de palabras en dicho conjunto. Dado que estamos tomando la probabilidad inversa, una menor perplejidad indicará un mejor modelo.

Capítulo 3

Creación de una colección de datos para la evaluación de la generación automática de texto en español

A lo largo del apartado anterior hemos explorado diferentes tipos de redes neuronales y su aplicación a la generación automática de texto, poniendo especial interés en los *embeddings* y su evolución a lo largo de los años.

Este apartado servirá para concretar algunos de los objetivos de este trabajo de investigación y explicar la propuesta de investigación que llevaremos a cabo utilizando las principales tecnologías asociadas al aprendizaje automático.

Nuestra propuesta incluye la creación y utilización de TolkienGen, un conjunto de datos común para los diferentes modelos de generación que utilizaremos. Con este *dataset* se llevarán a cabo diferentes experimentos que consistirán en la utilización de diferentes tipos de redes neuronales y *embeddings* que nos permitirán hacer un estudio de la evolución de la generación de texto y el panorama actual.

Desde un principio se consideró como un objetivo fundamental de esta memoria la creación de este conjunto de datos propio que sirviese como punto de partida para elaborar las diferentes soluciones de aprendizaje automático que hemos definido y que observaremos a continuación.

Pensamos en la inclusión de diferentes libros como fuente de datos ya que son un formato relativamente fácil de obtener y que contiene mucha información textual de carácter secuencial sin demasiado ruido. De hecho, es común ver entrenamientos de modelos utilizando grandes corpora textuales que beben de bases de datos llenas de libros o de páginas webs que contienen artículos [13] [15] [16].

Para nuestro objetivo siempre fue importante contar con gran cantidad de datos, y que estos tuviesen una calidad adecuada a fin de facilitar lo máximo posible las tareas de pre-procesamiento de los datos. Teniendo esto en mente se decidió optar por utilizar libros directamente. Son varias las razones que apoyan esta decisión.

Primero, utilizar libros nos permitía descargarlos en formato PDF, pasarlos a texto plano y guardarlos en nuestro repositorio de tal forma que pudiésemos consumirlos directamente y trabajar con ellos sin inconveniente alguno.

Segundo, los libros son fuentes de datos textuales secuenciales que no suelen contar con demasiadas faltas ortográficas, errores estructurales o problemas que dificulten su comprensión, por lo que también esto fue una razón importante para decidirse por ellos.

En tercer lugar hay que tener en cuenta la facilidad de disponibilización de estos recursos. Entraremos más en profundidad sobre el autor más adelante pero en general es fácil encontrar libros en diferentes formatos en la web, de tal forma que es sencillo descargarlos y almacenarlos sin tener que contar con un sistema que se conecte a una API o algo que tengamos que mantener actualizado.

Una consideración importante que también detallaremos a continuación pero que mencionaremos brevemente en este punto es que no hemos utilizado base de datos para almacenar TolkienGen.

Esto se debe principalmente al carácter de los datos que hemos utilizado, diferentes archivos en formato PDF que son fácilmente almacenables y no requieren de importantes tareas de recuperación de la información para acceder a ellos. Cuando entremos a los detalles en el apartado de Implementación se explicará cómo y por qué estos libros se están almacenando directamente en el repositorio.

Con nuestras búsquedas no hemos encontrado buenos *datasets* manejables en Español que se suelan utilizar para problemas de generación textual.

Los grandes modelos de *Hugging Face* utilizan principalmente la wikipedia en Español o corpora con grandes cantidades de libros para alimentar y ajustar sus modelos, ya sean versiones primero monolingües y luego convertidas a multilingües con este ejercicio o directamente creadas en una versión multilingüe. Estos conjuntos de datos son titánicos y requieren de gran potencia computacional tanto para usarlos como para almacenarlos.

TolkienGen tiene como objetivo servir como punto de partida para otras personas que realicen un estudio en el mismo área que nosotros y que no tengan la capacidad para usar conjuntos de datos demasiado grandes. Tiene la intención de servir como guía a fin de que cada uno pueda adaptar cada problema a sus necesidades y encontrar diferentes soluciones desde el punto de vista computacional que puedan ayudarle a la hora de decidir qué tipo de datos son mejores o peores en función de cada situación.

Además, no debemos olvidar que una de las características más importante de esta colección es el idioma. Todos los libros que estamos utilizando están en Español. El idioma ha sido una característica limitante durante todo el tiempo que hemos estado realizando la memoria, ya que no sólo hay poca bibliografía en español sobre este tema, también pocos recursos a disposición del usuario.

3.1. *Dataset*: Tolkien y la narrativa fantástica.

Es muy importante tener en cuenta las características de nuestra colección a la hora de utilizarla. Cada conjunto de datos es único, y tiene una serie de factores que determinan en buena medida el tipo de limpieza a aplicar, la selección de párrafos, la aplicación de diferentes técnicas…

En nuestro caso estamos trabajando con libros. Es importante matizar esto ya que no estamos trabajando con el contenido de los libros en sí, sino con los libros directamente. Con esto nos referimos a que TolkienGen contendrá elementos típicos de ellos como, por ejemplo, diálogos, lo que nos obliga a tener en cuenta los símbolos utilizados en ellos para poder limpiar adecuadamente el conjunto de datos.

Además, otra característica común que suelen tener los libros son las imágenes, y para el caso particular de nuestro autor seleccionado suelen ser frecuentes. En TolkienGen es posible encontrar mapas e ilustraciones que dificultan en muchos casos la extracción de texto.

Mención importante merecen algunas características particulares del libro en sí y de Tolkien en general. Nos referimos al sindarin, lenguaje élfico inventado por el autor que está presente en los libros de forma escrita y a través de imágenes escaneadas. Ha sido especialmente difícil lidiar con esto y como observaremos más adelante en el apartado de evaluación, ha condicionado en cierta forma la generación de texto de algunos de nuestros modelos.

Comentábamos en páginas anteriores que es bastante difícil acceder a conjuntos de datos especializados en generación textual, algo que además aumenta su dificultad cuando intentamos hacerlo en Español. Así pues, uno de los objetivos principales de este trabajo siempre fue crear un *dataset* que se pudiese adaptar a nuestros objetivos y servir como punto de partida para proyectos futuros y que además estuviese disponible en español de forma rápida y práctica.

Para crear el conjunto de datos orientado a la generación automática de historias estamos creando un *dataset* basado principalmente en obras de autores de un género literario concreto. Este tipo de conjuntos de datos han sido creados y usados en otros experimentos

relacionados con la temática, por lo que sabemos que su utilidad está probada. Además será interesante explorar cómo utilizando libros de una temática concreta y de autores concretos el texto a generar presentará elementos comunes a dicha temática y autores [12].

Nos hemos decantado por la narrativa fantástica y por Tolkien por encuadrarse dentro de un género concreto. La idea de centrar todo el *dataset* en un autor buscaba unificar criterios sintácticos, gramaticales…

Es importante tener en cuenta que, como Tolkien tiene como lengua nativa el inglés, sus obras en español son resultado de una traducción que puede variar en función del traductor que haya tratado con la obra original. Esto siempre genera una pérdida de la información original que es asumible para nuestra tarea, ya que los traductores encargados de estas obras suelen mantener el estilo original del autor y el significado general de sus textos. Para no caer en incoherencias, todos los libros utilizados para la construcción de TolkienGen tienen el mismo formato, traducción y editorial.

Es interesante analizar no sólo la generación textual, sino también el estilo del autor y sus particularidades. Esto es importante porque de las arquitecturas que propondremos, muchas de ellas utilizarán para entrenarse las frases de Tolkien tratadas de una forma u otra dependiendo de la situación. Esto también implica que estarán aprendiendo a escribir directamente del autor, y no de un conjunto de datos azarosos como podría ser los seleccionados de wikipedia. Esta aproximación que hemos intentado se ha realizado con la idea de unificar criterios.

La narrativa fantástica, al igual que cualquier otro tipo de narrativa, cuenta con una serie de elementos recurrentes que encontraremos a lo largo del texto. La presencia de estos elementos de forma repetida también ayudará a la red neuronal a encontrar patrones de clasificación que, en teoría, deberían facilitar su trabajo, permitiéndole adecuarse al estilo de una forma más óptima.

En este punto conviene recordar los problemas comentados anteriormente sobre la gestión de eventos y el control de actores en la generación textual.

Tengamos en cuenta que TolkienGen está compuesto de cinco libros que contienen cada uno de ellos una historia general, numerosas historias entrelazadas y un elenco de personajes común que pueden participar o no de estas historias. Para el ejercicio de generación textual que nosotros propondremos no es importante la supervisión de estos eventos y vigilar si se cumple o no una correcta creación de los mismos.

Cabe destacar que no estamos diciendo que al utilizar una temática común, como puede ser el género fantástico, y al utilizar un autor común como es Tolkien, el algoritmo lo tenga más fácil para generar texto con sentido. Nos referimos al hecho de que, al utilizar cinco libros del mismo autor, de la misma editorial y con el mismo traductor, obtendremos palabras y oraciones en común que nos permitirán reducir parámetros tan importantes como el tamaño del vocabulario y facilitar la comprensión del conjunto de datos.

3.2. Creación de la colección.

Ya hemos mencionado varias veces lo difícil que es encontrar un *dataset* preparado para la generación de texto en Español que se adaptase a nuestro problema. Desde un principio se tuvo en consideración que el conjunto de datos tenía que ser fácilmente recuperable en caso de que se perdiese alguno de los archivos originales o el proceso de extracción de texto diese algún problema.

También se buscaba homogeneidad en TolkienGen, ya que como hemos mencionado anteriormente, es importante tener en cuenta que dependiendo de la editorial y del traductor, el libro resultante puede variar.

Los libros de Tokien seleccionados (El Hobbit, El Señor de los Anillos, Las Dos Torres y El Retorno del Rey y Cuentos inconclusos de Númenor y la Tierra Media) han sido todos descargados de la misma fuente, a fin de unificar el formato, traducción y estilo [17].

Para obtener estos libros se realizó una búsqueda en la web intentando localizar algún repositorio donde pudieran descargarse varias de las obras del autor en un mismo formato de forma gratuita. Aunque encontramos varias opciones, al final nos decantamos por la página web *debeleer.com* [17] que almacena diferentes libros de literatura y ficción en formato PDF, prometiendo una descarga gratuita y sin límites del contenido.

La página web además permite realizar búsquedas por etiquetas, haciendo que localizar al autor y todas sus obras sea especialmente sencillo. También es destacable el hecho de que sólo tenga los libros de Tolkien citados en un único formato y en una única edición, facilitando el acceso y evitando cualquier tipo de confusión.

Escribiendo los títulos deseados en el buscador web es posible navegar hasta ellos sin dificultad. Después tan sólo tenemos que descargarlos y ya tendremos a nuestra disposición el documento en formato PDF para nuestro uso particular. Estos documentos se subieron al repositorio en cuestión y se encuentran disponibles para todo aquél que quiera consultarlos.

Sin embargo, todas las operaciones de extracción de datos de estos documentos se realizan sobre su versión en texto plano y no sobre su versión PDF. Esto es así por dos razones: es más sencillo operar sobre los documentos en texto plano y así evitamos cualquier tipo de alteración de los originales, que siempre quedarán guardados e inalterables en el formato original.

Cabe destacar que en un principio se optó por realizar nosotros mismos la extracción de texto de los PDF pero finalmente se optó por usar un *software* externo. Tras consultar diferentes OCR y experimentar con su uso, se llegó a la conclusión de que era mas sencillo convertirlos usando alguna herramienta web especializada [30].

Los archivos PDF simplemente se subieron a la herramienta web en cuestión y se descargaron los archivos TXT resultantes, que posteriormente se almacenaron también en el repositorio.

Este paso, a pesar de hacerse de forma breve y simple, interviene en gran medida en la calidad final de la conversión. Extraer texto de un PDF siempre conlleva una serie de inconvenientes de las TolkienGen no está exento. Así pues, a pesar de realizarse diferentes pruebas para extraer el texto de la mejor manera posible, nuestra extracción tiene algunos errores fácilmente observables si se abrieran y leyesen los archivos en texto plano.

Por ejemplo, muchos de los diálogos se juntan con el anterior o el próximo párrafo interfiriendo en la correcta estructura del texto. Así mismo, muchos inicios y finales de párrafo se desordenan, juntándose también con otros párrafos. Muchas veces las imágenes generan saltos de línea no deseados que también influyen en la capa texto, haciendo que diferentes frases se separen de sus respectivos párrafos.

La mayoría de los errores descritos afectan principalmente a la estructura de párrafos del texto, lo cual podría pasar por alto ya que la extracción de texto en sí es buena. Sin embargo, para medir la calidad de TolkienGen nos centraremos principalmente en los párrafos de nuestro *dataset* por lo que su alteración y desorden influyen directamente sobre los resultados finales.

Para este punto cabe hacer una importante distinción en nuestro *dataset*. Tenemos un total de cinco libros, de los cuáles vamos a utilizar cuatro para entrenar y uno para predecir a fin de que los conjuntos de entrenamiento y predicción nunca se junten. Por lo tanto, tendremos un conjunto de entrenamiento compuesto por los títulos de El Hobbit, El Señor de los Anillos, Las Dos Torres y El Retorno del Rey y un conjunto de test compuesto por el título Cuentos inconclusos de Númenor y la Tierra Media. Es muy importante hacer esta distinción ya que de otra forma el rendimiento de nuestros modelos no sería objetivo y no podríamos

realizar un análisis de los resultados real. De aquí en adelante y para aclarar cualquier duda, haremos referencia a los cuatro primeros títulos cuando hablemos del conjunto de entrenamiento, y haremos referencia al último título cuando hablemos del conjunto de test. Cuando mencionemos el conjunto de validación estaremos hablando del porcentaje del conjunto de entrenamiento destinado a hacer la validación para las métricas de *keras* o *transformers*.

Así pues, todas las menciones que encontremos a los párrafos de TolkienGen cuando hablemos de predicciones hacen siempre referencia a aquellos extraídos del título Cuentos inconclusos de Númenor y la Tierra Media, ya que es nuestro conjunto de test.

Para la extracción de párrafos se decidió no utilizar ninguna librería externa ya que la mayoría de los párrafos están separados por doble salto de línea. Así pues, realizando una búsqueda sobre el texto y extrayendo aquellos separados por doble salto de línea es como conseguimos almacenar todos los párrafos de la colección.

Luego, con ayuda del *tokenizer* de NLTK contamos los *tokens* presentes en cada párrafo y calculamos la media sabiendo el número de párrafos totales con el que contamos.

Finalmente, para extraer las frases de inicio de cada párrafo simplemente hemos optado por calcular la media de tamaño de las frases y obtener de cada párrafo esa media en *tokens*. Esta última operación sólo se ha realizado sobre los párrafos del conjunto de test, ya que las frases resultantes serán las utilizadas como entrada a la red neuronal.

3.3. Caracterización de la colección.

División de datos del *dataset* TolkienGen

	Conjunto de entrenamiento	Conjunto de test
Tokens	1.250.000	457.000
Frases	92.000	22.000
Párrafos	5.900	2.200

Tabla 1: División de datos TolkienGen.

TolkienGen cuenta con 1.600.000 palabras. El tamaño medio de las frases es de 20 palabras, y el de los párrafos de 300. Contamos con un total de 110.000 frases y un total de 8.000 párrafos. Estas características se han tenido en cuenta a la hora de crear toda la *pipeline* de pre-procesado que describiremos en apartados posteriores.

Para la evaluación solo trabajaremos con los aproximadamente mil párrafos de más de treinta tokens que hemos extraído de la obra Cuentos Inconclusos de Númenor y la Tierra Media. Se ha tomado esta decisión con la intención de generar párrafos con una longitud considerable que permitan hacer una buena evaluación.

El valor original de la perplejidad de nuestra colección es 3.46 para todos los párrafos en total de toda la colección y el valor de perplejidad para los párrafos del conjunto de test es de 3.25. Para calcular la perplejidad hemos utilizado el módulo de python *flair* [28], una biblioteca especializada en procesamiento del lenguaje natural que contiene diferentes modelos de *embeddings* [31]. Especialmente interesante ha sido su versión en español, que nos ha permitido calcular la perplejidad sin tener que convertir ningún modelo del inglés a nuestro idioma.

Nosotros usaremos este valor (3.25) como referencia para calcular la calidad del texto generado por nuestros diferentes modelos. En el apartado de implementación describiremos el código utilizado para hacer este cálculo de perplejidad con la biblioteca *flair*.

Capítulo 4

Experimentación

A lo largo del siguiente punto explicaremos la metodología y el proceso de implementación seguido para trabajar con nuestra colección y generar con ella los párrafos que posteriormente serán evaluados con la métrica seleccionada.

4.1. Implementación

A fin de entrar de primera mano en el mundo de la generación textual y comprobar de la forma más fiel posible la evolución del género y las posibilidades comentadas en páginas anteriores, se ha realizado para esta memoria un repositorio de Github que contiene todos los archivos necesarios para extraer información de un archivo en texto plano, pre-procesarlo y alimentar diferentes algoritmos de redes neuronales.

El repositorio ha sido creado en GitHub, aunque se ha trabajado fundamentalmente en local y sólo se han hecho contribuciones desde una única cuenta. También cabe mencionar la existencia de un *notebook* en *Goole Colab* que contiene la estructura básica y el flujo de trabajo central que luego se plasmaría en el repositorio, aunque este último está más completo y será el que mostraremos en esta memoria [18].

El repositorio recibe el nombre de *text_generation* y consta de cuatro módulos principales: *datasets*, dónde almacenaremos los libros destinados a convertirse en el conjunto de datos de nuestra colección y sus diferentes variantes, *ml_preprocessing*, enfocado a gestionar todo el apartado de ingesta y pre-procesamiento de los datos y *ml_training*, que contiene todos los algoritmos necesarios para entrenar nuestras redes neuronales.

A continuación describiremos en profundidad cada uno de los módulos presentes en nuestro repositorio.

4.1.1. datasets

A pesar de que lo hemos denominado módulo, en realidad *datasets* no es más que una carpeta dentro de nuestro repositorio donde almacenamos los archivos pre-procesados de nuestro conjunto de datos.

Recordemos que estamos trabajando con cinco títulos del autor J.R.R. Tolkien (El Hobbit, El Señor de los Anillos, Las Dos Torres, El Retorno del Rey y Cuentos inconclusos de Númenor y la Tierra Media). Como ya hemos mencionado, se descartó en un principio almacenar este *dataset* en una base de datos especializada, ya que no se considera que tenga un tamaño significativo.

Los archivos PDF están disponibles en la carpeta *pdf_files*, donde podremos observar los cinco títulos en el formato seleccionado para trabajar. Nuestro flujo de trabajo no extrae la información directamente desde *pdf_files*, sino que trabaja con *raw_files*, carpeta dónde encontramos exactamente lo mismo que en *pdf_files* pero en texto plano, utilizando para ello el *software* externo que describimos anteriormente [30].

Otras dos carpetas importantes y que guardan relación entre ellas son *sentences_files* y *sequences_files*. Ambas almacenan diferentes listas de información en formato *pickle*. Para la creación de estos archivos se ha utilizado el módulo *pickle* de python, que implementa protocolos binarios que permiten serializar y deserializar una estructura de objetos python.

En *sentences_files* encontramos un archivo por cada título de Tolkien anteriormente mencionado. Cada uno de estos archivos almacena todas las frases de cada libro correspondiente. La razón de mantener por separado esta información, técnica que veremos de forma continuada, es la trazabilidad. Así seremos capaces en todo momento de distinguir de donde proviene la información que estamos consultando o con la que estamos trabajando.

El siguiente archivo es *sequences_files* y contiene las secuencias en formato de ventana rodante descritas en el apartado de Tratamiento de los datos. Al igual que *sentences_files,* aquí encontramos un archivo *pickle* por cada libro de Tolkien utilizado a fin de mantener la trazabilidad en todo momento.

La carpeta *hf_files* contiene el *dataset* unificado en el formato necesario para entrenar los modelos de *Hugging Face*. Encontraremos tres archivos llamados *data.txt, train.txt, test.txt*. El primero consiste en el resultado de todos nuestros datos en un mismo archivo. Los otros dos son una división de la totalidad del *dataset* en dos conjuntos, el de entrenamiento y el de evaluación, con una proporción de 75% para el conjunto de entrenamiento y 25% para el de evaluación. El resto de archivos presentes en esta carpeta son simplemente producto del entrenamiento de *Hugging Face* y no añaden nada a la explicación.

La carpeta *paragraphs_files* contiene gran cantidad de información necesaria para realizar la evaluación de nuestro sistema. El archivo *tolkien_paragraphs* almacena todos los párrafos de TolkienGen. El archivo *tolkien_paragraphs_numenor* almacena tan sólo los párrafos de nuestro conjunto de test, mientras que el archivo *tokenized_tolkien_paragraphs_numenor* almacena en formato *tokenizado* todos los párrafos de un tamaño mayor a 30 tokens también sólo del conjunto de test. El archivo *seed_text_paragraphs_numenor* almacena el inicio de cada párrafo del conjunto de test, que se ha estimado en 20 palabras. Por último, encontramos *perplexity_per_model*, que simplemente almacena los resultados de calcular la perplejidad para cada uno de nuestros modelos.

Por supuesto es importante destacar que ninguno de los archivos de secuencias, frases o aquellos destinados a entrenar nuestros modelos GPT-2 han utilizado frases o párrafos del conjunto de test. Siempre se ha mantenido una estructura que los separa como se puede observar en el repositorio.

De esta forma mantenemos todo lo relacionado con nuestro conjunto de datos en un lugar centralizado y localizado al que podemos acudir con facilidad tanto para extraer la información como para consultarla.

4.2.2. ml_preprocessing

Este módulo es, junto con *ml_training,* el corazón de nuestro repositorio. En *ml_preprocessing* encontramos un conjunto de archivos python destinados a pre-procesar los datos de tal forma que los tengamos siempre a nuestra disposición en el mejor formato posible para el uso que queramos darles.

El módulo de *ml_preprocessing* es el responsable de generar varios de los archivos explicados en el apartado anterior de *datasets*. Dentro de él encontramos primero el archivo *cleaner.py* que contiene una de las muchas clases propias creadas para este proyecto; *Cleaner*. Esta clase se encarga de tomar un determinado conjunto de datos, *tokenizarlo*, limpiarlo, seleccionar las frases de más de un tamaño determinado y generar un texto limpio.

En la clase *Cleaner* hemos definido todo tipo de funciones destinadas a eliminar acentos, puntuaciones, símbolos y números, así como también *stop words*. Sin embargo, a pesar de tener esta capacidad no todas han sido usadas. Se tomó la decisión de no eliminar ni *stop words* ni números a fin de mantener la coherencia textual.

Recordemos que estamos buscando generar un texto que luego vamos a evaluar en función de su coherencia. Si eliminamos elementos como *stop words* reduciremos en gran medida el

tamaño del vocabulario (los artículos y preposiciones aparecen con gran frecuencia en los textos), pero haremos que la red neuronal no los tenga en cuenta a la hora de predecir. Esto se vuelve aún más importante en los algoritmos de redes neuronales recurrentes, ya que no disponemos de ningún mecanismo de atención que permita conferir más o menos peso a cada token.

Para este punto es importante comentar que todo el repositorio utiliza un sistema de rutas relativas definidas desde cada módulo, de tal forma que al instalarlos es posible importarlos y acceder a sus archivos sin necesidad de utilizar rutas absolutas, lo que facilita el trabajo.

El archivo *sequences_generator* contiene uno de los algoritmos más complejos empleado en nuestro código. Dentro de él encontraremos la clase *SequencesGenerator* y en su interior una función denominada *sentences_to_sequences*. Esta función, elaborada por nosotros personalmente en python y sin librerías externas, es la encargada de tomar una frase y construir a partir de ella la ventana rodante que comentábamos anteriormente en el tratamiento de datos. La función es además parametrizable, de tal forma que podemos definir el tamaño mínimo y máximo de la secuencia que vamos a generar.

En nuestro caso, el tamaño mínimo está configurado para ser dos y el máximo, ocho. Generar una secuencia de un token no tendría ningún sentido y no aportaría información a nuestra red neuronal recurrente. El número máximo de tokens en la secuencia sí que es algo más complicado de definir. En nuestro caso se probaron dos configuraciones, seis y ocho. Se optó por la configuración de ocho ya que en las pruebas realizadas ofreció mejores resultados.

Un punto a destacar de este algoritmo es su capacidad para adaptarse al tamaño de cada frase. Hemos mencionado que el tamaño mínimo de la secuencia será de dos y el máximo de ocho. Si por alguna razón la secuencia no pudiese construirse manteniendo estos criterios (por ejemplo, si fuese más pequeña que ocho o si el número de tokens no puede dividirse correctamente) el algoritmo es capaz de adaptarse y construir secuencias de menor tamaño o del tamaño adecuado.

A la hora de realizar este algoritmo se tuvieron en cuenta diferentes factores que detallaremos a continuación.

Hay que distinguir tres tipos de preparación de los datos; la que usaremos para las redes neuronales recurrentes implementadas en *Keras*, la que usaremos para los modelos genéricos de GPT-2 seleccionados y, por último, la que utilizaremos para realizar *fine tuning* sobre el modelo de GPT-2 que hemos seleccionado.

En este caso, este algoritmo se encuentra destinado a preparar los datos para los modelos de generación implementados basados en redes neuronales recurrentes. La preparación de los datos se realizará tomando cada frase del documento (libro) y convirtiéndola en una secuencia que servirá para alimentar la red neuronal. La secuencia se compondrá de todos los elementos de la frase menos del último, que se convertirá en el elemento a predecir. Tomar cada una de las ochenta y dos mil frases presentes en nuestro conjunto de datos puede parecer una gran cantidad de datos, lo que en teoría debería beneficiar nuestro entrenamiento.

Sin embargo, hay que tener en cuenta que esas ochenta y dos mil frases están codificadas utilizando *one hot encoding* y muchas veces son frases largas, cuya secuencia a predecir incluye multitud de elementos a tener en cuenta antes de llegar al último token.

El carácter largo de estas secuencias y la gran cantidad de información que contienen obligaba a plantear alternativas, y así es como se decidió optar por una ventana rodante que permitiese ampliar los datos e incluso simplificarlos.

Definamos qué consideramos como ventana rodante a fin de entender como crearemos la entrada de estas redes neuronales.

Dada la frase:

Ayer estuve con mis amigos

Nosotros construiremos la secuencia:

Ayer estuve

Ayer estuve con

Ayer estuve con mis

Ayer estuve con mis amigos

estuve con

estuve con mis

estuve con mis amigos

con mis

con mis amigos

mis amigos

Realizamos esta operación con la intención de mejorar la calidad del *input* de nuestra red neuronal. De otra forma y utilizando tan solo las frases planas sin construir secuencias quizás habríamos tenido un *dataset* muy poco representativo de lo que se considera un texto y no habríamos podido predecir palabras con la misma eficacia.

Este apartado de pre-procesamiento ha sido muy potente, construyendo para ello una arquitectura que pretendía limpiar lo mejor posible el texto a fin de eliminar el ruido pero sin pasarnos ya que necesitamos construir resultados coherentes. El resultado ha aumentado nuestro número de frases hasta un total de casi tres millones de secuencias.

A este respecto se han tomado decisiones como dejar las *stopwords* ya que la mayoría de ella son preposiciones o artículos. Es normal eliminar este tipo de palabras en tareas de clasificación como análisis de sentimiento ya que no aportan nada a la frase (el significado general no queda condicionado por ellas), pero a la hora de generar texto eliminarlas nos llevaría a construcciones sintácticas sin sentido alguno.

La semántica y la sintaxis tienen un tratamiento diferente en la generación de texto. Así pues, dejar las *stopwords* ha aumentado considerablemente el tamaño del vocabulario a predecir y esto perjudica nuestros resultados.

El resto de funciones de la clase están orientadas a almacenar y consultar la información. La función *create_sentences_files* lee los archivos en texto plano almacenados en *datasets*, aplica el algoritmo que acabamos de comentar y genera un dos archivo pickle por cada archivo en texto plano, uno con todas las frases y otro con todas las secuencias. Esta función llama a la clase *Cleaner* y la utiliza para limpiar el texto que va leyendo mientras va construyendo las secuencias. Cabe destacar que la clase *Cleaner* también puede llamarse de forma parametrizable, de tal forma que somos capaces de decidir si queremos que elimine las *stop words* o cuándo hacemos la llamada a la función *make_clean*.

Las funciones *get_sentences* y *get_sequences* permiten recuperar la información presente en estos archivos de forma cómoda.

A continuación encontramos el archivo *hf_dataset_generator*. Existe una relación entre este archivo y el anterior que aunque no es directa es importante resaltarla. Mientras que el

archivo *sequences_generator* se encarga de generar las secuencias que servirán como *input* a las redes neuronales recurrentes, este archivo es el encargado de generar los conjuntos de entrenamiento y validación para GPT-2.

La clase *HfDatasetGenerator* contiene tan solo tres funciones. La función *create_raw_text*, se encarga de generar el archivo *data.txt*, tomar todos los archivos en texto plano (menos aquellos referentes al set de validación) y guardar su contenido en dicho archivo. De esta forma combinamos todo TolkienGen en un archivo único. La función *get_raw_text* nos permite recuperar la información dentro del archivo *data.txt*, y la función *train_test_split_dataset* es la encargada de tomar toda la información almacenada en *data.txt*, dividirla y crear dos archivos: *train.txt* y *test.txt*, que servirán para hacer *fine tuning* sobre el modelo de GPT-2 elegido.

Esta última función también es parametrizable, de tal forma que podemos definir el tamaño del conjunto de validación sobre el total. En nuestro caso se ha elegido una proporción de 75% para el conjunto de entrenamiento y 25% para el conjunto de validación. No se han hecho más pruebas a este respecto, ya que el coste computacional de hacer *fine tuning* sobre uno de estos modelos es demasiado alto.

Antes comentábamos el archivo *SequencesGenerator* y lo complicada que fue la elaboración de su algoritmo de creación de secuencias. Sin embargo, en *HfDatasetGenerator* podemos ver otra cara de la moneda; *Hugging Face* y la comodidad que representa su repositorio y el uso de su *pipeline*. *Hugging Face* requiere tan sólo de dos archivos (*train.txt* y *test.txt*) y es la propia clase la que se encarga de realizar la ingesta de datos tomando la información de estos dos archivos, sin entrar a la necesidad de qué tipo de pre-procesamiento se necesita en la entrada textual de información para el modelo.

Pasamos ahora al archivo *paragraphs_generator*, también dentro del módulo. En este archivo encontramos la clase *ParagraphsGenerator*, que es la encargada de generar algunos de los archivos comentados en el módulo de *datasets* relacionados con los párrafos. Esta clase es bastante más simple que los anteriores y contiene una única función llamada *create_paragraphs_files* que toma toda la información presente en los archivos de texto plano, extrae los párrafos y los almacena en un archivo pickle llamado *tolkien_paragraphs* de forma única o *tolkien_paragraphs_numenor* si hace referencia a los párrafos del conjunto de test.

Es importante matizar cómo hemos extraído los párrafos. Las librerías NLTK y Spacy nos permiten segmentar un texto por párrafos. Tras varias pruebas utilizando estas librerías de diferentes formas, se decidió que la segmentación se llevaría a cabo partiendo simplemente por doble salto de línea. De todas las formas que hemos descrito anteriormente, la que nos

pareció que aportaba mejores resultados era el doble salto de línea, que además no dependía de ninguna librería para funcionar.

Todos estos archivos componen el conjunto de clases que hemos creado para pre-procesar el texto. Dentro del mismo módulo encontramos también una carpetas: *stopwords*, que contiene una lista de *stop words* en formato YAML.

4.3.3. ml_training

Dentro de los diferentes módulos que componen nuestro repositorio este es el mas complejo, ya que contiene todo lo necesario para entrenar y predecir con los diferentes algoritmos de aprendizaje automático seleccionados.

Dividimos el contenido de este módulo en dos carpetas de suma importancia, *training* e *inference*. La primera contendrá todos los archivos python con sus correspondientes algoritmos de aprendizaje automático listos para entrenar. La segunda se enfoca en leer los modelos producidos en *training* y generar el texto que posteriormente vamos a evaluar.

Primero nos centraremos en *training*, ya que es la carpeta dónde se ejecutan primero los archivos para generar los modelos que necesitaremos en *inference*.

Dentro de la carpeta *training* encontramos varios elementos importantes. Observaremos una lista de archivos python nombrados siempre con el algoritmo en cuestión seguido del *framework* que los implementa. De esta forma encontramos aquí una LSTM, una BiLSTM, una GRU implementadas en *Keras* y *Tensorflow* y dos versiones *fine tuned* de GPT-2.

Otro de los elementos importantes es la carpeta *config*. Dentro de esta carpeta encontraremos diferentes archivos YAML que guardan la configuración de hiperparámetros seleccionada para nuestra red neuronal.

Estos archivos son de vital importancia y han ido variando tras varias semanas de experimentación en las que se han lanzado diferentes entrenamientos con diferentes configuraciones hasta alcanzar las que tenían mejores resultados. Más adelante entraremos al detalle sobre algunos de estos números.

Si entramos dentro de cualquiera de estos archivos observaremos una estructura idéntica que consta de un diccionario cuya clave será el nombre del hiperparámetro en cuestión y cuyo valor será el determinado por nosotros.

La decisión de mover los hiperparámetros del código a archivos de configuración externa fue tomada únicamente por comodidad. Debido al alto número de entrenamientos lanzados, se hizo más cómodo simplemente ir cambiando los diferentes hiperparámetros en un archivo de configuración externo y guardando dicho archivo que editando el código cada vez que quisiésemos cambiar de configuración.

De esta forma, se elaboró un archivo de configuración para cada una de nuestras redes neuronales, a fin de que también pudiese visualizarse rápidamente la configuración elegida para cada una de ellas. De hecho, así podemos observar con sólo abrir los archivos como la estructura de la LSTM es diferente a la de la BiLSTM, y bastante similar a la de la GRU.

Estos archivos cobran sentido cuando entramos al detalle de cada una de las redes neuronales.

La primera de ellas se encuentra dentro del archivo *lstm_keras*. Como ya hemos comentado varias veces, hemos utilizado *Keras* para construir estas redes neuronales. *Keras* es una librería que nos proporciona acceso de alto nivel mediante una API a las funcionalidades de *Tensorfow*.

Tensorflow es una biblioteca de código abierto especializada en aprendizaje automático que fue desarrollada por Google y lanzada en 2015. Con *Keras* se hace mucho más intuitivo el uso de *Tensorflow*.

Seleccionamos *Tensorflow* y no *PyTorch* (otra de las grandes bibliotecas de aprendizaje automático) para implementar nuestras redes neuronales simplemente por costumbre. Ha sido la librería con la que hemos trabajado durante todo el máster y nos ha parecido más sencilla de manejar.

La clase *LstmKerasTrainer* es la encargada de gestionar la LSTM que hemos construido. Dentro de ella y al iniciarla se carga la configuración en su archivo YAML correspondiente. Luego, encontramos dos funciones muy importantes que también encontraremos en las otras dos redes neuronales recurrentes implementadas: *get_rolling_windows_sequence* y *keras_embeddings*.

La primera, *get_rolling_windows_sequences* lee el contenido de todos los archivos almacenados en el módulo de *datasets* dentro de la carpeta *sequences_files*. Este es un paso importante en el que me gustaría detenerme unos instantes.

No existe una conexión real entre el módulo de *ml_preprocessing* y el módulo de *ml_training*. Sí que existen algunas llamadas entre los diferentes módulos del repositorio

pero generalmente se debe ejecutar cada archivo dentro de cada módulo para generar los pasos que estamos comentando. Si ejecutásemos alguna de las redes neuronales aquí presentes sin haber ejecutado previamente los archivos de *ml_preprocessing* obtendríamos un error. Por ejemplo, la función *get_rolling_windows_sequences* podría llamar a la clase *SequencesGenerator* de *ml_preprocessing* y crear las secuencias directamente, haciendo que no sea necesario ejecutar nada previamente.

Sin embargo, hemos preferido mantener cada tarea de cada módulo independiente, de tal forma que no haya dependencias y pueda controlarse en todo momento lo que va generando cada uno de nuestros componentes. Además, de esta forma es posible observar cada paso que se ha ido realizando.

Una vez que hemos explicado el sentido de nuestra función *get_rolling_windows_sequences*, pasamos a *keras_embeddings*. Esta función también se repetirá en las otras clases. Aquí es dónde usamos el *Tokenizer* de *Tensorflow* para codificar la entrada textual que vamos a pasarle a la red neuronal. La clave está en el método *texts_to_sequences*, que transformará el texto en una secuencia de números enteros. Básicamente cogerá cada palabra (cada token, ya que le pasamos el texto tokenizado) y la reemplazará con un valor correspondiente a la posición que ocupe dentro del texto.

Recordemos que no podemos pasarle directamente el texto a nuestra red neuronal, así que esta conversión a números nos permite codificarlo de tal forma que el formato resultante (números enteros) pueda ser interpretado por la red neuronal.

Este proceso se realiza tanto para el conjunto de entrenamiento como para el conjunto de validación de forma independiente. De hecho es importante explicar este punto. Para utilizar *texts_to_sequences* previamente tenemos que haber asignado un número a cada palabra en función de su posición, y esto se hace mediante el método *fit_on_texts*. Si nos fijamos, sólo realizamos esta operación sobre el conjunto de entrenamiento y no sobre el conjunto de validación.

El vocabulario se forma de la suma de todas las palabras presentes en el texto. El vocabulario del conjunto de entrenamiento y del conjunto de validación no tiene por qué ser el mismo, y de hecho si no lo son es mejor ya que así la evaluación es objetivamente mejor. Cuando creamos el diccionario que guarda cada palabra y su posición utilizando el método *fit_on_texts* estamos sólo teniendo en cuenta el vocabulario presente en el conjunto de entrenamiento, de tal forma que cuando intentemos aplicar este diccionario al conjunto de validación, si hay una palabra que no exista en el diccionario se la ignorará en la secuencia.

Es en este concepto de vocabulario donde cobra especial importancia lo que antes habíamos mencionado sobre las *stop words*. Para este caso se decidió mantenerlas, lo que habrá supuesto un incremento significativo del tamaño del vocabulario. Tengamos en cuenta que si las hubiésemos suprimido, el diccionario palabra-posición resultante habría sido menor y por tanto también la complejidad de la red neuronal.

Otros dos momentos importantes suceden dentro de esta función. Cuando calculamos la longitud máxima de las secuencias y cuando calculamos el número total de palabras. La primera operación es necesaria para calcular cuál es la secuencia más larga de entrada a nuestra red neuronal, ya que necesitamos que todas las entradas guarden coherencia en sus dimensiones y no haya algunas más largas que otras.

Para nosotros es fácil calcular la longitud máxima de nuestra secuencia de entrada, ya que anteriormente hemos comentado que nuestro algoritmo crea secuencias de un tamaño entre dos y ocho tokens. La longitud máxima será entonces de ocho, y la mínima de dos.

No podemos meter secuencias de diferentes tamaños, así que para solucionar esto se rellenan las secuencias de menor tamaño de tal forma que todas ocupen lo mismo que aquella que tiene la longitud máxima, es decir, ocho. Esto es lo que se conoce como *padding* y nosotros tendremos que realizarlo para las secuencias de tamaños entre dos y siete. Este proceso se realiza añadiendo ceros a la secuencia y se puede hacer de dos formas, antes de la secuencia en cuestión (*pre*) y después (*post*). Para nosotros no tiene ningún sentido añadir ceros a la derecha de la secuencia (*post*) ya que buscamos predecir el último elemento de dicha secuencia, así que el *padding* que realizamos es a la izquierda de la secuencia (*pre*).

Lo segundo que calcula esta función es el tamaño máximo del vocabulario. Como al principio hemos creado un índice palabra-posición, simplemente calculamos la longitud de este diccionario y obtendremos este valor, que será importante próximamente.

La función *train_test_sequences* también es común a las otras redes neuronales y simplemente se encarga de dividir las secuencias que hemos creado de tal forma que obtengamos el *input* de entrada y el elemento a predecir. En nuestro caso, la secuencia de entrada a la red neuronal estará formada por todos los elementos de la secuencia menos el último. Este último *token* será el elemento que tenga que predecir nuestra red neuronal.

Por último llegamos a la función *lstm_keras*. Tenemos al final otra función llamada *train* pero se encarga principalmente de poner en ejecución el resto de funciones, así que no entraremos en detalle sobre ella.

Para este punto y cómo ya hemos mencionado anteriormente podemos deducir que la estructura de nuestros otros archivos python, a excepción de *gpt2_hf* se compone de una clase y una lista de funciones exactamente iguales a las descritas hasta ahora. Así pues, lo que cambia dentro de cada clase es la existencia de una función que construye secuencialmente las capas de cada tipo de red neuronal de una forma u otra.

Podríamos haber agrupado todas las redes neuronales en un mismo archivo python, pero hemos decidido crear tres archivos diferentes a fin de que controlemos en todo momento qué estamos entrenando y de qué forma lo estamos haciendo.

Los dos archivos que tienen una estructura similar a *lstm_keras* son *bilstm_keras* y *gru_keras*. En cada uno de ellos encontraremos una clase (*LstmKerasTrainer, BilstmKerasTrainer* y *GruKerasTrainer*) y una función que se llama igual que el archivo (*lstm_keras*, *bilstm_keras* y *gru_keras*). En este punto entraremos a las particularidades de cada una de estas funciones ya que contienen los elementos mas importantes para crear nuestros modelos.

La primera de ellas, *lstm_keras* contiene una LSTM. Hemos creado el modelo de forma secuencial y queda compuesto por una capa de *embeddings*, una LSTM y la capa densa final. Es la red neuronal más simple que tenemos.

A continuación definimos el optimizador (*Adam*, *Adaptative moment estimation*) y la tasa de aprendizaje. Finalmente compilamos nuestro modelo usando la función de pérdida (*sparse_categorical_crossentropy*), el optimizador y las métricas seleccionadas (*accuracy*) y entrenamos.

Hemos definido muchos conceptos importantes así que nos detendremos un momento a examinarlos.

Como hemos mencionado, la primera capa de nuestra red neuronal es una capa de *embeddings*. Es la encargada de llevar nuestros *embeddings* a la red neuronal. En ella utilizamos el tamaño total de nuestro vocabulario, definimos las dimensiones y la longitud de entrada, que será el tamaño máximo de la secuencia menos uno (ya que este último *token* lo estamos utilizando como elemento a predecir), de esta forma nos aseguraremos que la entrada de datos a nuestra red neuronal tendrá siempre el mismo tamaño.

Luego encontramos una capa LSTM donde hemos definido una serie de unidades y, por último, la capa densa final que tiene como función de activación *softmax*. El número de activaciones en la capa densa final siempre es igual al número posible de predicciones, que en

nuestro caso se corresponde con el tamaño total del vocabulario, ya que cada palabra es un posible elemento a predecir.

Las funciones de activación son de vital importancia para hacer funcionar nuestra red neuronal. Se encuentran en cada neurona y les indican cuando se tienen que activar o apagar. Algunos ejemplos son *ReLU* o *sigmoid*, sin embargo nosotros utilizaremos *softmax* .La función de activación de esta capa es *softmax* porque nos estamos enfrentando a un problema de clasificación múltiple.

Esta función de activación toma vectores de números reales como entradas y los normaliza en una distribución de probabilidad proporcional a los exponenciales de los números de entrada. Antes de aplicar, algunos datos de entrada podrían ser negativos o mayores que 1. Además, es posible que no sumen 1. Después de aplicarla, cada elemento estará en el rango de 0 a 1, y los elementos sumarán 1. De esta manera, pueden interpretarse como una distribución de probabilidad. Para mayor claridad, cuanto mayor sea el número de entrada, mayores serán las probabilidades.

El optimizador puede pasar desapercibido, pero es una definición muy importante. Básicamente es el encargado de optimizar los valores de los parámetros, de tal forma que va reduciendo el error cometido por la red. Esto lo hace sirviéndose del *learning rate* que definamos. Existen muchas opciones como SGD o RMSprop. Nosotros hemos seleccionado Adam, que es como RMSprop pero con momentum. Adam calculará una combinación lineal entre el incremento actual y el incremento anterior, y considerará los gradientes recientemente aparecidos en las actualizaciones para mantener diferentes tasas de aprendizaje por variable. El *learning rate* es el hiperparámetro encargado de medir cuánto tardará en actualizarse el algoritmo de optimización.

La función de pérdida que definimos al compilar nuestro modelo es *sparse categorical crossentropy*. Es utilizada en tareas de clasificación multiclase (como nuestro caso). Tanto *categorical crossentropy* como *sparse categorical crossentropy* tienen la misma función de pérdida pero *categorical crossentropy* se usa cuando las etiquetas están codificadas usando *one hot encoding*, por ejemplo: [1,0,0], [0,1,0] y [0,0,1] y *sparse categorical crossentropy* es usada cuando las etiquetas están codificadas en números enteros, por ejemplo, [1], [2] y [3] para un problema de 3 clases. Como para nuestro caso hemos utilizado este último ejemplo de codificación, utilizaremos *sparse categorical crossentropy*.

Por último, simplemente entrenamos nuestro modelos sirviéndonos del método *fit*, y definiendo el número de *epochs*. Esto es el número de veces que se ejecutan los algoritmos de *forwardpropagation* y *backpropagation*. El primero se define como la manera en que las redes neuronales crean las predicciones. La red neuronal tiene una serie de valores aleatorios en

cada neurona. Los datos de entrenamiento pasarán por cada una de estas neuronas hasta llegar a la capa de salida (capa densa) y aquí será donde la red neuronal predecirá la categoría de los datos. Estas predicciones se sirven de la función de pérdida para medir la calidad de la red neuronal. El segundo es el encargado de optimizar la función de pérdida y mejorar las predicciones de la red neuronal. Calcula las derivadas de los parámetros para saber cómo están afectando al resultado de la función de pérdida.

Retomando el tema de los *epochs*, nuestro *dataset* tiene definidos 10. Si tuviéramos 5000 frases, en cada uno de los 10 ciclos definidos las 5000 frases pasarían por la red neuronal.

De esta forma hemos construido la arquitectura básica de nuestras redes neuronales recurrentes. No definiremos esto en profundidad para la BiLSTM o la GRU, ya que la configuración es prácticamente igual, aunque sí que resaltaremos algunas de las diferencias y explicaremos el por qué de estas.

Por ejemplo, hemos mencionado anteriormente que nuestra LSTM tan sólo contiene una capa entre la capa de entrada y la capa de salida. Si ahora prestásemos atención a la BiLSTM observaríamos que esta contiene un total de cuatro capas entre la de entrada y la de salida, mientras que la GRU vuelve a tener tan sólo una capa al igual que la LSTM.

La razón de definir estas tres estructuras es la experimentación. No entraremos en los detalles específicos de los números de cada configuración, ya que estos se encuentran en los archivos YAML comentados anteriormente, aunque sí detallaremos por qué los hemos ido variando.

LSTM

Primero elaboramos una LSTM simple, sin configuraciones complejas. Intentábamos observar el comportamiento de nuestro *dataset* en la red neuronal base sin caer en complicaciones. Cada uno de los entrenamientos ejecutados tardó aproximadamente 10 horas de media. Se hicieron varios entrenamientos, pero sólo reflejaremos aquí los más interesantes.

Para el siguiente entrenamiento utilizamos una dimensiones de entrada de 50 y un número de neuronas en la capa LSTM de 40.

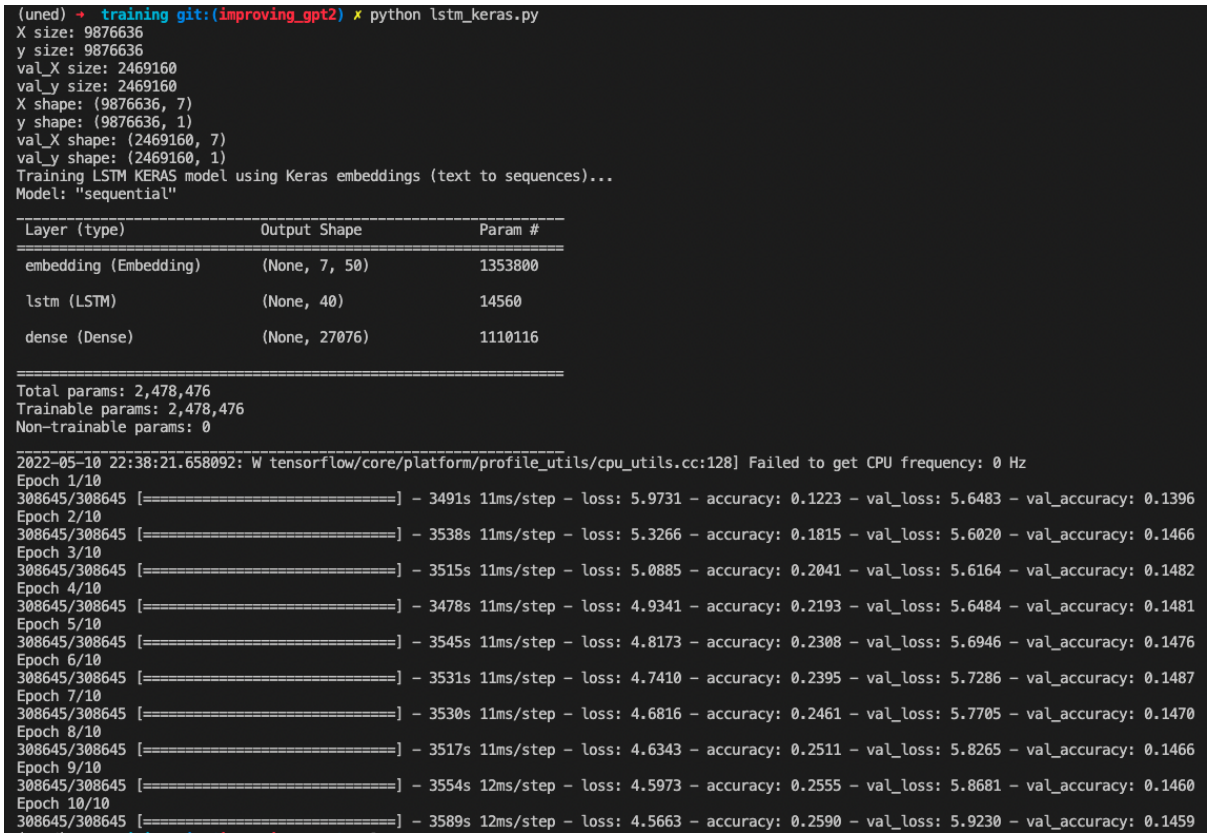

Figura 1: Crossvalidation de LSTM de 2.478.476 parámetros.

Podemos observar un número de parámetros de 2.478.476. El resultado obtenido es de 0.1459 de precisión en el set de validación. Está claro que es una precisión bastante reducida pero debemos tener en cuenta que estamos tratando de averiguar cuál es la siguiente palabra de entre más de 25.000 posibilidades (número que sacamos examinando el tamaño de nuestro vocabulario). Teniendo esto en cuenta podríamos asumir que la precisión es decente.

Otro de los entrenamientos lanzados fue el siguiente.

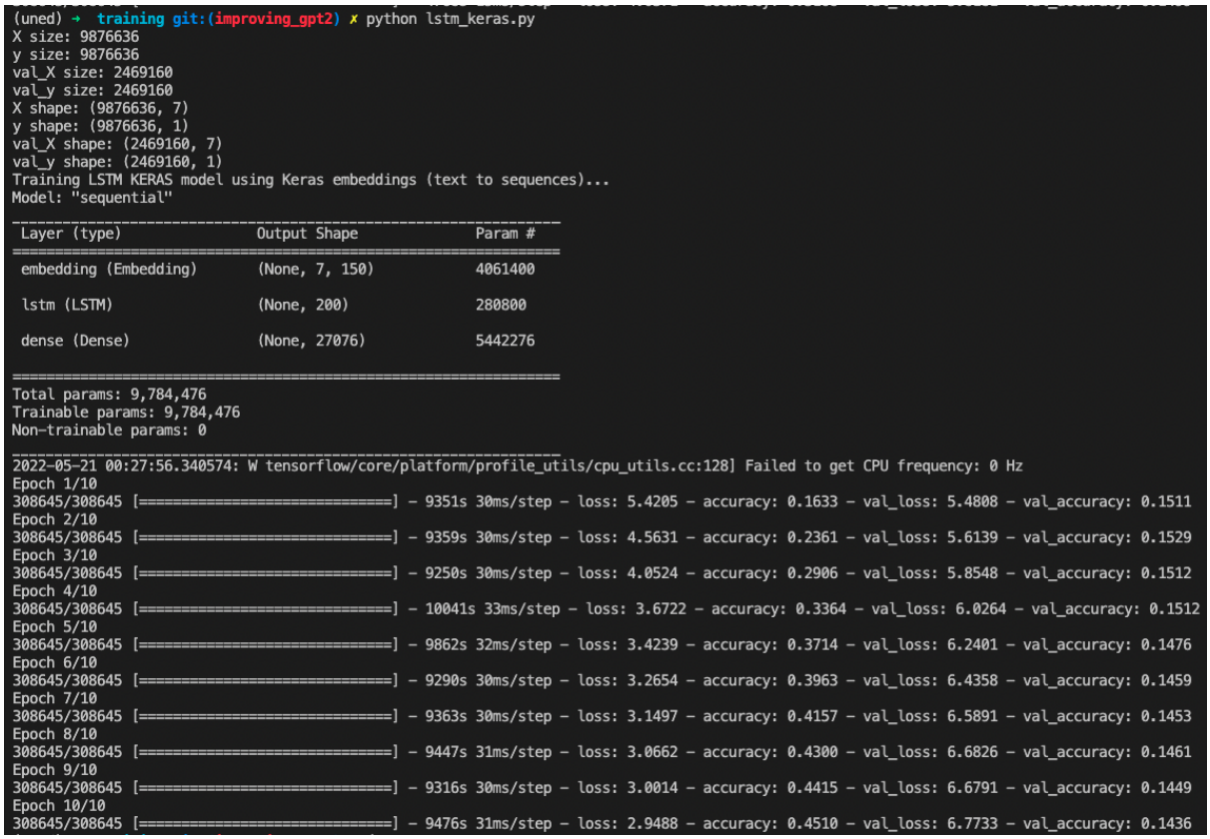

Figura 2: Crossvalidation de LSTM de 9.784.476 parámetros.

En este caso hemos aumentado bastante los parámetros de nuestro entrenamiento y de 2 millones hemos pasado a 9 millones. Hemos aumentado las dimensiones de entrada del embedding y las neuronas de la LSTM. El resultado en términos de val_accuracy es semejante. En términos de hiperparámetros, hemos aumentado las dimensiones de entrada del embedding a 150 y las neuronas de la LSTM a 200.

Finalmente lanzamos otro de los entrenamientos, reduciendo mucho la complejidad de nuestra red neuronal.

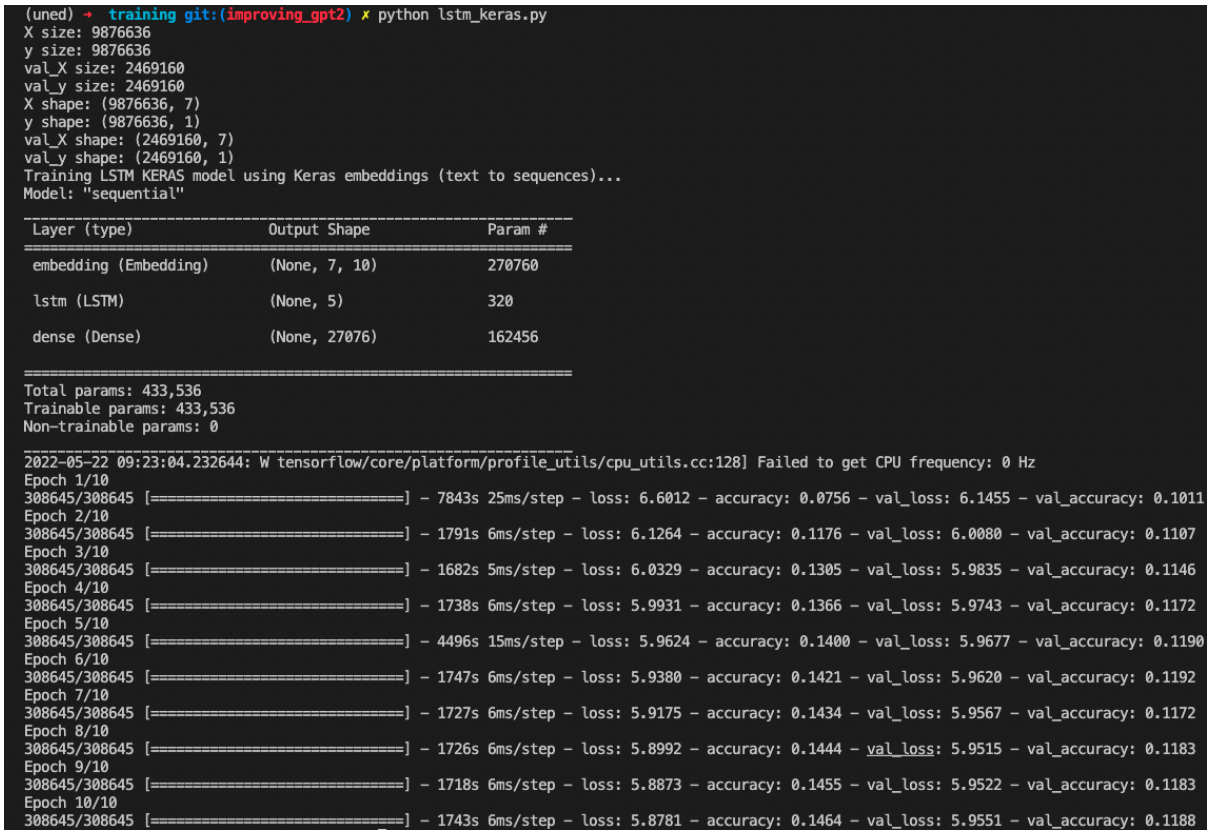

Figura 3: Crossvalidation de LSTM de 433.536 parámetros.

Para este caso hemos optado por justo lo contrario que en el caso anterior. Hemos reducido muchísimo los parámetros y de los 2 y 9 millones que hemos visto anteriormente ahora estamos en medio millón. Para esto hemos bajado las dimensiones de la primera capa de embeddings a 10 y las neuronas de la LSTM a 5. El resultado es inferior a otros entrenamientos mostrados anteriormente.

BiLSTM

Ahora encontramos una red neuronal BiLSTM. Para esta además hemos añadido un nuevo elemento: *dropout*. Teniendo en cuenta que la BiLSTM es mas potente que la LSTM hemos tomado la decisión de añadir *dropout* a fin de impedir el overfitting. También hemos apilado diferentes capas para comprobar el resultado.

El entrenamiento ha utilizado unas dimensiones de entrada en la capa de *embeddings* de 60 y un número de neuronas de 30 en la primera capa y 15 en la segunda. El *dropout* era de 0.2.

```
(uned) →  training git:(improving_gpt2) ✗ python bilstm_keras.py
X size: 9876636
y size: 9876636
val_X size: 2469160
val_y size: 2469160
X shape: (9876636, 7)
y shape: (9876636, 1)
val_X shape: (2469160, 7)
val_y shape: (2469160, 1)
Training LSTM KERAS model using Keras embeddings (text to sequences)...
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 7, 60)             1624560

 dropout (Dropout)           (None, 7, 60)             0

 bidirectional (Bidirectiona  (None, 7, 60)            21840
 l)

 dropout_1 (Dropout)         (None, 7, 60)             0

 bidirectional_1 (Bidirectio  (None, 30)               9120
 nal)

 dense (Dense)               (None, 27076)             839356

=================================================================
Total params: 2,494,876
Trainable params: 2,494,876
Non-trainable params: 0
_________________________________________________________________
2022-05-12 09:52:37.870384: W tensorflow/core/platform/profile_utils/cpu_utils.cc:128] Failed to get CPU frequency: 0 Hz
Epoch 1/10
308645/308645 [==============================] - 3917s 13ms/step - loss: 6.1339 - accuracy: 0.1063 - val_loss: 5.7823 - val_accuracy: 0.1204
Epoch 2/10
308645/308645 [==============================] - 4283s 14ms/step - loss: 5.6627 - accuracy: 0.1488 - val_loss: 5.6570 - val_accuracy: 0.1350
Epoch 3/10
308645/308645 [==============================] - 4105s 13ms/step - loss: 5.4578 - accuracy: 0.1702 - val_loss: 5.6115 - val_accuracy: 0.1408
Epoch 4/10
308645/308645 [==============================] - 3829s 12ms/step - loss: 5.3231 - accuracy: 0.1821 - val_loss: 5.6013 - val_accuracy: 0.1448
Epoch 5/10
308645/308645 [==============================] - 3897s 13ms/step - loss: 5.2349 - accuracy: 0.1906 - val_loss: 5.6076 - val_accuracy: 0.1460
Epoch 6/10
308645/308645 [==============================] - 3880s 13ms/step - loss: 5.1607 - accuracy: 0.1971 - val_loss: 5.6061 - val_accuracy: 0.1474
Epoch 7/10
308645/308645 [==============================] - 3858s 12ms/step - loss: 5.0957 - accuracy: 0.2025 - val_loss: 5.6174 - val_accuracy: 0.1484
Epoch 8/10
308645/308645 [==============================] - 3880s 13ms/step - loss: 5.0415 - accuracy: 0.2071 - val_loss: 5.6126 - val_accuracy: 0.1499
Epoch 9/10
308645/308645 [==============================] - 3900s 13ms/step - loss: 4.9929 - accuracy: 0.2110 - val_loss: 5.6216 - val_accuracy: 0.1507
Epoch 10/10
308645/308645 [==============================] - 3928s 13ms/step - loss: 4.9555 - accuracy: 0.2140 - val_loss: 5.6269 - val_accuracy: 0.1519
```

Figura 4: Crossvalidation de BiLSTM de 2.494.876 parámetros.

Hemos crecido en 20.000 parámetros con respecto a la LSTM que hemos presentado anteriormente. Esta nueva red neuronal presenta unos parámetros parecidos aunque no idénticos. Se ha modificado el tamaño de la dimensión de entrada y los de las neuronas de la LSTM. Así mismo, hemos añadido *dropout* a la ecuación.

El resultado es algo mejor que el de la LSTM. No sólo en lo que a la cifra final de accuracy se refiere sino también en lo que a la evolución del modelo observamos. El crecimiento de la *val_accuracy* indica un buen rendimiento y parece que iba avanzando positivamente pero sólo pudimos destinar 10 epochs a este entrenamiento debido a los altos periodos de entrenamiento totales.

Otro de los entrenamientos lanzados fue el siguiente:

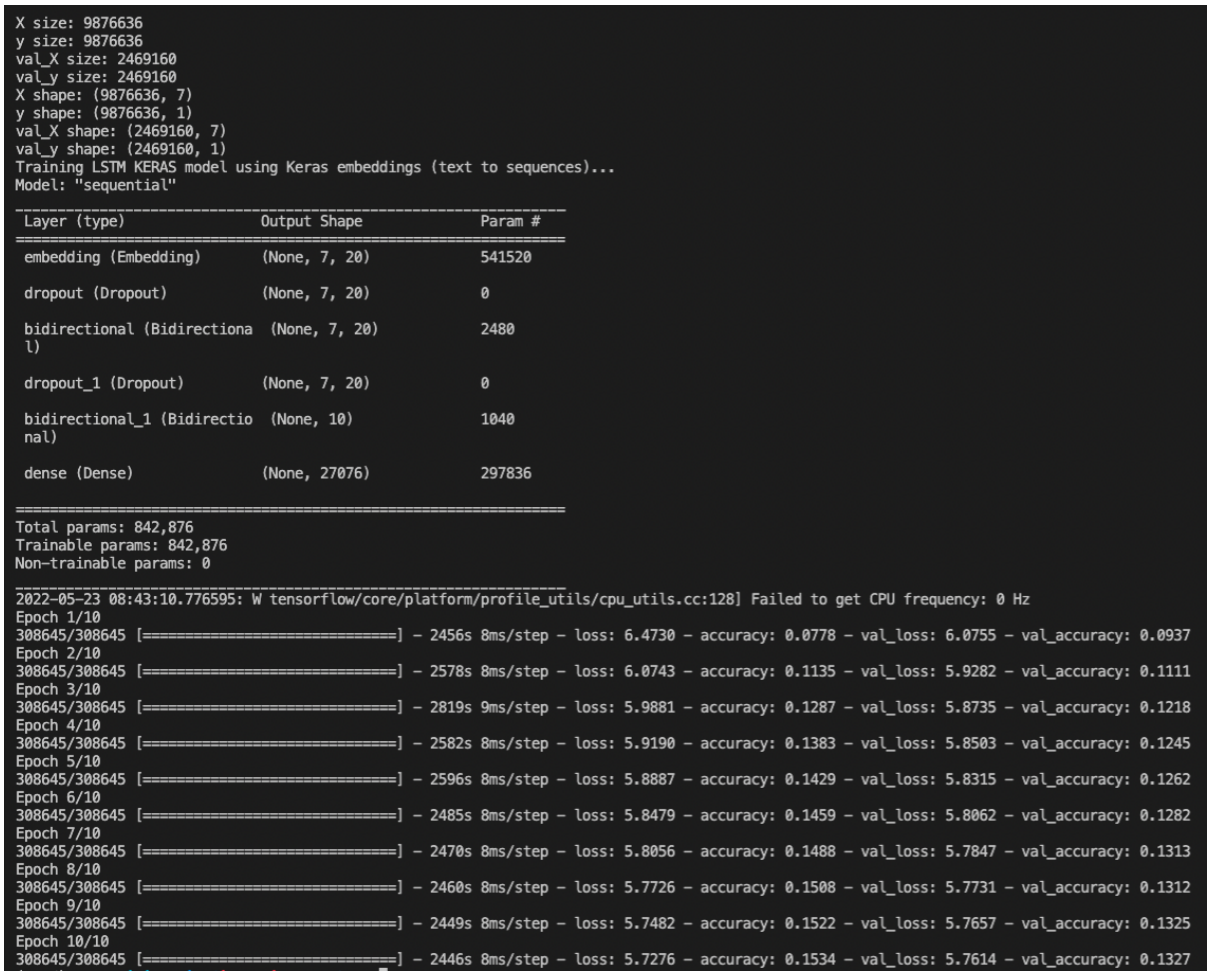

Figura 5: Crossvalidation de BiLSTM de 842.876 parámetros.

Este nuevo entrenamiento reduce los parámetros de 2 millones y medio a 800.000. Para ello hemos bajado el embedding de entrada hasta 20, la primera capa de neuronas de LSTM hasta 10 y la segunda hasta 5.

El resultado es inferior al que hemos visto anteriormente.

Para el caso de nuestra BiLSTM no hemos realizado más entrenamientos significativos, ya que el tiempo para completar 10 epochs era muy largo y no teníamos tanto tiempo disponible (18 horas por entrenamiento).

GRU

Por último llegamos a la GRU, que presenta una estructura similar a la LSTM.

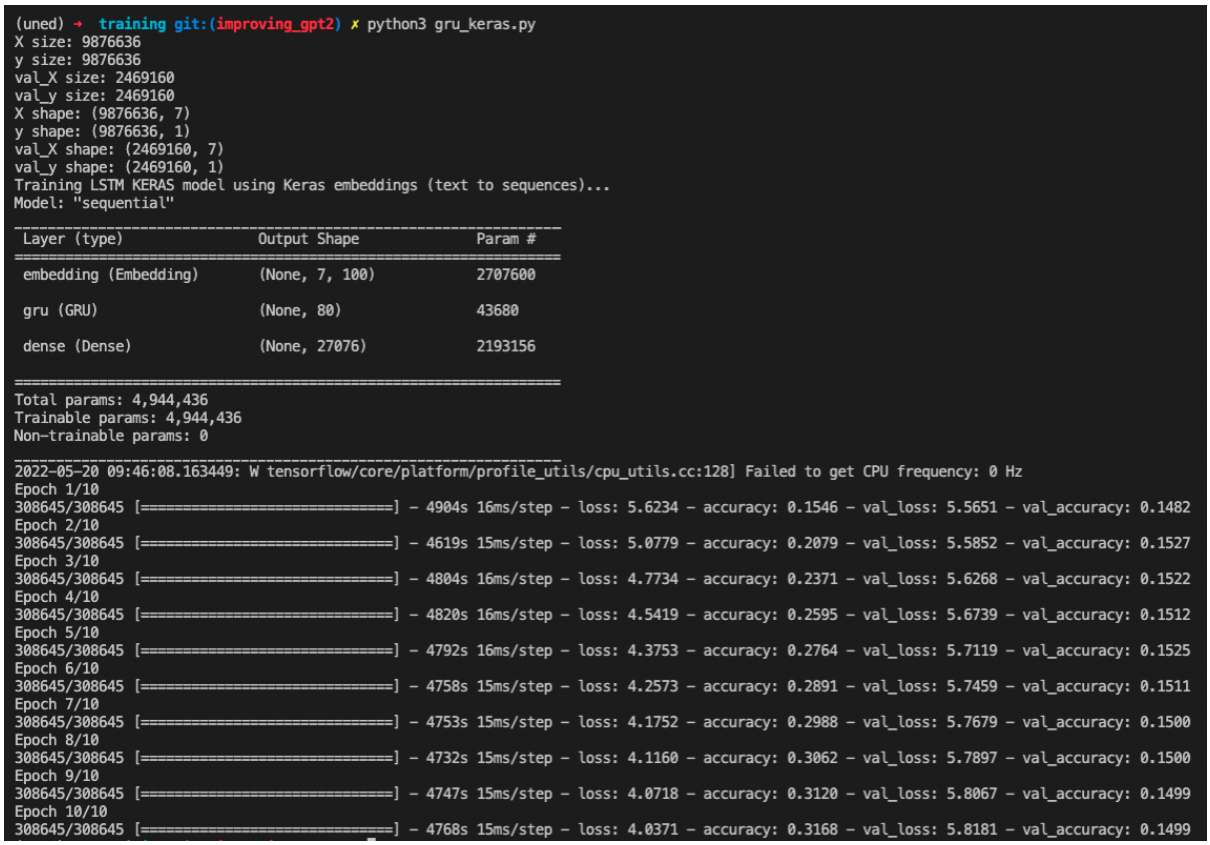

Figura 6: Crossvalidation de GRU de 4.944.436 parámetros.

Esta GRU cuenta con 5 millones de parámetros y en ella hemos usado un embedding de 120 y una LSTM de 100. El resultado es ligeramente superior a los obtenidos anteriormente.

Tenemos datos de un segundo entrenamiento:

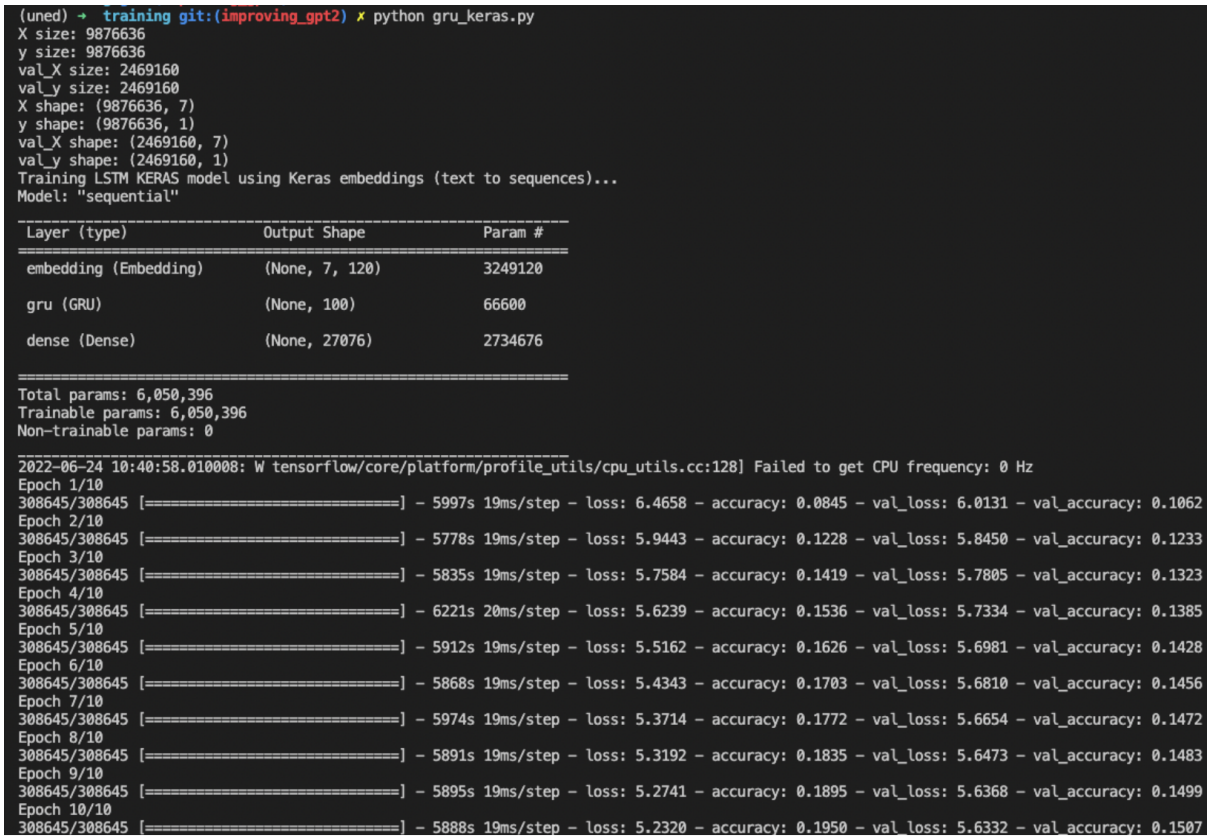

Figura 7: Crossvalidation de GRU de 6.050.396 parámetros.

En este experimento hemos subido el número de parámetros 1 millón hasta los 6 aumentado y las dimensiones del *embedding* y de la LSTM.

Con esto cerramos el apartado de redes neuronales recurrentes dentro de *ml_training*. Hemos comentado varias veces el tema del peso computacional de estos modelos. El ordenador era incapaz de realizar entrenamientos por periodos de tiempo muy largo y los primeros entrenamientos fueron descartados. El número de datos era muy grande por lo que tomamos la decisión de reducir el tamaño del *dataset* de forma azarosa mediante la extracción de algunas frases. A pesar e esto, el tiempo de los experimentos no permitió que pudiéramos realizar muchos más que los aquí presentados y algunos cuyos datos no fueron registrados por sus pobres resultados. también tengamos en cuenta que estos experimentos no podían realizarse en paralelo, ya que uno sólo de ellos significaba una gran carga para el ordenador.

GPT-2

Para terminar con la parte de *ml_training* tenemos dos archivo *gpt2_hf* que contienen todo lo necesario para entrenar el *dataset* utilizando un modelo de *transformers* previamente descargado. Podemos observar una versión para *datificate* y otra para *DeepESP*. Ambas

tienen una clase definida de manera idéntica por lo que no hablaremos de las dos, tan sólo de la clase encargada de entrenar estos modelos que es igual.

El código y las funciones dentro de la clase son más sencillos que en el apartado de redes neuronales recurrentes implementadas en *Keras*. *Hugging Face* y *Transformers* proveen de una lista de métodos y clases que automatizan gran parte del trabajo que hemos tenido que realizar en el apartado anterior.

La función *load_dataset* carga el conjunto de datos que previamente hemos dividido en *train* y *test*, y la función *train_model* utiliza estos datos sobre el modelo de GPT-2 anteriormente descrito.

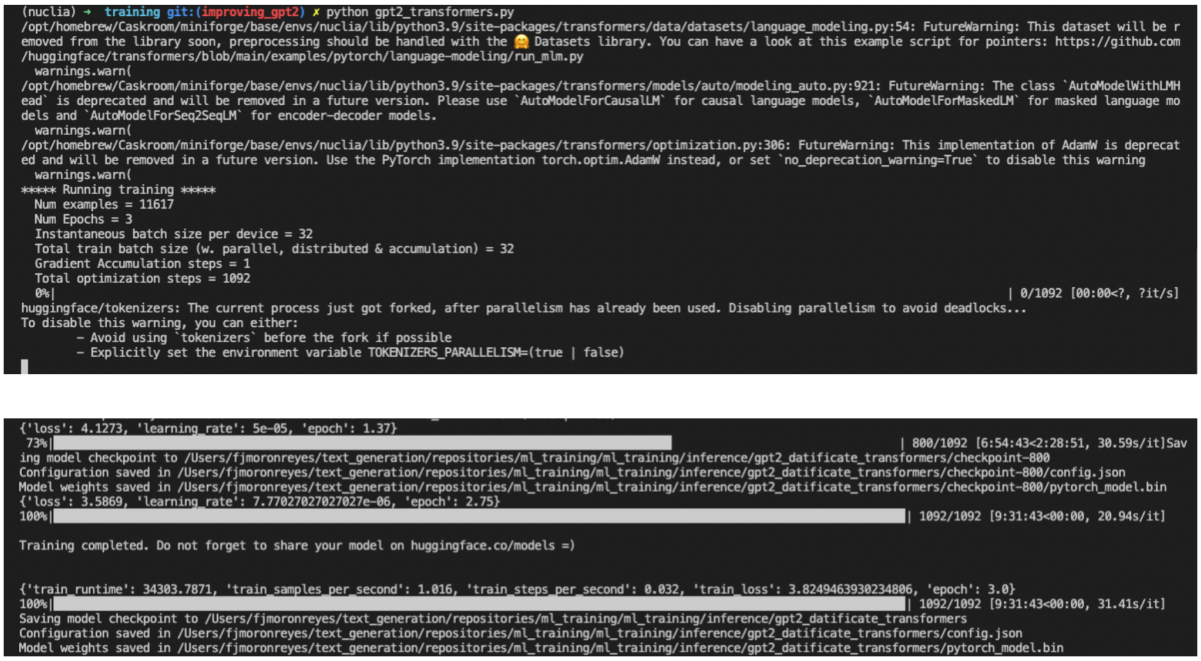

Figura 8: Ejemplo de logs de entrenamiento de gpt_2.

De esta forma obtenemos el modelo ajustado sobre nuestro *dataset*. En este punto no tuvimos que realizar experimentos, ya que utilizamos la configuración por defecto que nos deja *Hugging Face*, al considerarla la mejor posible para esta tarea.

Sí que se tuvo que investigar cómo construir el modelo de GPT-2 en fine tuning, pero al igual que en el caso de Keras y Tensorflow, el entrenamiento con *Hugging Face* es también muy largo, y este tardó unas dieciocho horas. Debido a los tiempos ajustados de este trabajo, no se pudo replicar este proceso cómo nos hubiera gustado.

Es así como entrenamos y obtenemos dos modelos de GPT-2 ajustados al *dataset* TolkienGen, que serán almacenados en la carpeta de *inference*.

En último lugar y tras analizar la carpeta de *training* pasamos a la carpeta de *inference*, donde se encuentra el código necesario para predecir los resultados y evaluar nuestros modelos.

En esta carpeta encontraremos una serie de ficheros nombrados con las redes neuronales recurrentes y otra nombrada como *gpt2_hf* seguido del nombre del modelo en cuestión. Estos ficheros contienen todos los archivos necesarios para cargar cada uno de los modelos. Decir que no hemos subido los modelos de *transformers* directamente porque su peso no los hacía viables para el repositorio, pero si se entrenase alguna de las clases anteriormente descritas podría observarse su creación y almacenamiento en la carpeta designada.

Empezaremos por el archivo *seed_paragraphs_generator*, que es el que menos relación guarda con el resto. En este archivo encontramos una serie de funciones encargadas de varias tareas. La función *get_paragraphs* es la encargada de calcular la media de *tokens* por frase y párrafo y almacenarlo. Luego encontramos *save_tokenized_paragraphs*, que guarda los párrafos *tokenizados* en un archivo. Y, por último, la función *save_seed_texts*, que se encarga de definir el inicio de cada párrafo y guardarlo en otro archivo a fin de que pueda servir como inicio para nuestra generación de texto.

Esta última función es importante ya que es la encargada de generar la lista que utilizaremos para evaluar cada uno de nuestros modelos y calcular la perplejidad de cada párrafo. Generará una lista que contendrá el inicio definido para cada párrafo que hemos almacenado en el paso anterior. Todo esto, como ya hemos comentado, se realiza sobre el conjunto de test, de tal forma que los datos usados para entrenar y los datos usados para validar los modelos son completamente diferentes.

Luego encontramos los archivos *predict_keras* y *predict_hf*, ambos encargados de cargar los modelos y realizar las predicciones.

El primero contiene la clase *PredictKeras.* Para llamarla debemos pasarle el nombre de una de nuestras tres redes neuronales (lstm, bilstm o gru), y según cual seleccionemos cargará una serie de archivos u otros. Luego encontramos la función *get_tokenizer*, que permite obtener el objeto *Tokenizer* de *Keras* necesario para realizar las predicciones. Por último, encontramos la función *predict* que se encarga de, dada una semilla, generar texto en función de lo que necesitemos.

Esta función recibirá una determinada entrada textual y generará la siguiente palabra. En ella definimos una variable llamada *predictions* (con valor 100), que nos permite definir cuántas palabras queremos generar. De esta forma todos los párrafos que generemos tendrán

el mismo tamaño y no habrá problemas con la longitud a la hora de aplicar nuestras métricas. El procedimiento a la hora de generar texto será el siguiente: definimos una entrada textual, que se introduce en el modelo para generar la siguiente palabra. Una vez que hemos generado dicha palabra, se añade a nuestra entrada y vuelta a empezar, así hasta que generemos el párrafo entero.

El segundo archivo contiene la clase *PredictGPT2*. Al igual que la anterior, para llamarla debemos pasarle el nombre de una de las cuatro versiones de GPT-2 que estamos usando (datificate, deepesp o las versiones ajustadas de ambos: *pretrained_datificate* y *pretrained_deepesp*).

Antes mencionábamos que habíamos hecho ajuste sobre dos modelos de GPT-2 y que usábamos los otros dos modelos directamente. Es aquí en esta clase donde podemos observar esto. Si nos fijamos, cuándo iniciamos la clase también usamos la *pipeline* de *Transformers* para descargar o cargar el modelo que necesitemos. Esto nos permitirá usarlo de manera directa sin problemas.

Al igual que en el caso anterior tenemos una función *predict* qué es la encargada de, dado un determinado modelo escogido, generar el párrafo. Al igual que en las redes neuronales recurrentes, hemos definido un tamaño máximo de 100, de tal forma que todos los párrafos tendrán el mismo tamaño independientemente del modelo que los esté generando.

Por último pasamos al archivo *perplexity*, que es el encargado de generar los párrafos y calcular la perplejidad para cada uno de ellos. Este cálculo ha sido realizado utilizando flairNLP [28], una librería que nos permite utilizar embeddings multilingües.

Las dos primeras funciones, *calculate_perplexity* y *average_perplexity* son las encargadas de realizar los cálculos semánticos necesarios. Por supuesto el modelo cargado es en español. Mientras que la primera función se encarga puramente del cálculo de perplejidad dado un determinado texto, la segunda calcula la perplejidad media en todo el conjunto de párrafos generado por un determinado modelo.

Las otras dos funciones presentes están destinadas simplemente a recuperar información (*get_tolkien_paragraphs* y *get_seed_tolkien_paragraphs*). La primera servirá para calcular la perplejidad original de la colección sobre el conjunto de test, y la segunda servirá como entrada textual para cada modelo, de tal forma que comprobaremos la perplejidad para cada uno de ellos.

El resto del código toma cada una de las clases de *Keras* y *HF* construidas anteriormente para llamar a cada tipo de red neuronal, generar el texto, almacenarlo en una lista y posteriormente calcular la perplejidad.

4.2. Métrica y metodología de evaluación

En el estado del arte dedicamos un apartado completo a explicar diferentes métricas utilizadas en procesamiento del lenguaje natural en general y en generación de texto en particular. De todas las métricas explicadas en esta memoria usaremos Perplejidad

BLEU, Rouge y METEOR se basaban principalmente en la comparación de textos generados con textos de referencia. Para nosotros estas tres métricas no eran especialmente de utilidad porque aunque podemos contar con los párrafos originales de Tolkien como referencia, los diferentes modelos de generación que hemos entrenado producirán diferentes textos que no tienen por qué estar relacionados con el párrafo original.

Es decir, nosotros estamos tomando como entrada textual la primera frase de cada uno de los párrafos seleccionados de nuestro *dataset*. Esta frase servirá como *input* a cada uno de los seis modelos presentados, que se encargarán de generar un párrafo de extensión similar al original. Como nuestro objetivo no es medir la calidad de la generación textual comparándola con la original, sino que buscamos medir simplemente la calidad de la generación textual por sí misma, no podemos utilizar BLEU, Rouge o METEOR.

Sin embargo, la Perplejidad no depende de un conjunto de frases de referencia, y es una métrica especialmente buena a la hora de medir la coherencia del texto generado. Necesitábamos evaluar los modelos que habíamos creado más allá de utilizar como referencia la validación cruzada que nos ofrecen las diferentes librerías que hemos utilizado. Así pues, uno de los objetivos de este trabajo siempre fue comprobar la calidad (la coherencia y cohesión) del texto generado. Es por ello que nos detuvimos más en su definición, y la razón por la que nos decantamos por ella en este apartado.

Para ello hemos optado por los siguientes pasos. Hemos extraído todos los párrafos presentes en nuestro *dataset* con un tamaño significativo (mayores de 30 *tokens*) presentes en el conjunto de evaluación. Estos párrafos se convertirán en nuestro conjunto de evaluación. Esto nos da como resultado unos mil párrafos aproximadamente. Hemos medido también la longitud media de estos párrafos, situándola aproximadamente en unos 300 *tokens.*

Aquí tomaremos dos caminos, uno será el de dejar los párrafos en su estado actual, y otro el de tomar el inicio de cada uno de estos párrafos y construir un *dataset* con esto. El inicio de estos párrafos será la primera frase.

Una vez tenemos estos dos conjuntos de datos, el de los párrafos sin alterar, y el de la primera frase de cada párrafo, y habiendo calculado la media de *tokens* por párrafo, se han realizado dos procesos, uno consistente en medir la perplejidad para los párrafos sin alterar y otro consistente en generar párrafos de 300 *tokens* de longitud con cada uno de nuestros modelos.

De esta forma obtendremos 8 medidas de perplejidad diferentes: la original de Tolkien y la de nuestras diferentes redes neuronales. Esta perplejidad estaría calculada sobre los párrafos originales del *dataset*, pero usando la primera frase como entrada y el resto generado por nosotros, hasta una longitud media que pretende alcanzar una extensión parecida a la de los párrafos originales, a fin de que podamos igualar las condiciones lo máximo posible y la evaluación sea lo más justa posible.

De esta forma podemos comparar la calidad del texto generada por Tolkien y la de las diferentes soluciones de aprendizaje automático que hemos realizado.

4.3. Resultados

Al final de la implementación hablábamos del proceso de cálculo de la perplejidad para cada uno de nuestros modelos. A continuación observaremos una tabla que hace referencia a dicho cálculo y que presenta siete valores diferentes, uno perteneciente a la perplejidad original de la colección y otros seis producto de los párrafos generados por cada uno de nuestros modelos.

Las aproximaciones que vamos a comparar son:

1. LSTM: La versión más manejable de las redes neuronales recurrentes que hemos presentado. Es una versión sin capas múltiples ni *dropout* que pretende ser una aproximación simple.
2. BiLSTM: La versión más compleja de nuestras redes neuronales. Es una LSTM bi-direccional apilada que pretende ser la aproximación más compleja a nuestro problema.
3. GRU: Versión simple como la LSTM pero aplicando una GRU para observar el comportamiento de diferentes arquitecturas pero con parámetros e hiperparámetros similares.
4. GPT2-DATIFICATE: Ajuste de GPT-2 en inglés mediante el entrenamiento del modelo usando la wikipedia en Español.

5. GPT2-DEEPESP: Ajusta de GPT-2 en inglés que también usa diferentes textos en Español de la wikipedia para entrenarse.
6. FINETUNE_DAT: Versión ajustada por nosotros del modelo GPT2-DATIFICATE. Se ha utilizado TolkienGen para su entrenamiento.
7. FINETUNE_DEP: Versión ajustada por nosotros del modelo GPT2-DEEPESP. Al igual que en la versión anterior se ha utilizado TolkienGen para su entrenamiento.

Perplejidad media de cada aproximación sobre el *dataset TolkienGen*

ORIGINAL	LSTM	BILSTM	GRU	GPT2 DATIFICATE	GPT2 DEEPESP	FINETUNE DAT	FINETUNE DEP
3.254	7.889	6.981	9.249	4.186	4.425	4.323	4.537

Tabla 2: Perplejidad media de cada aproximación sobre el dataset TolkienGen

Muchas veces ocurre que la métrica de evaluación utilizada en validación cruzada de la red neuronal a nivel de código no coincide con la que posteriormente se presenta para analizar los resultados del modelo.

Nosotros tenemos este caso, al usar Precisión para evaluar los resultados de nuestra red neuronal pero luego utilizar Perplejidad para medir la calidad del texto generado. En este trabajo no encontraremos consideraciones sobre la métrica utilizada para la validación cruzada que nos ofrece, por ejemplo, la librería de *Keras,* ya que nos hemos centrado principalmente en aquella que nos permite definir la calidad del texto generado; Perplejidad.

Recordemos una vez más que estas medidas se obtienen tras analizar la perplejidad de la colección sobre el conjunto de test y no sobre la totalidad del *dataset*.

Original hace referencia a la perplejidad de los textos de Tolkien sin ningún tipo de pre-procesamiento (tan sólo aquellos con más de un tamaño determinado, algo que se repite también en el resto de opciones).

Estamos trabajando con los dos mil párrafos de más de treinta *tokens* que hemos extraído de TolkienGen, para los cuales hemos calculado una media de trescientos *tokens*. Nuestra generación queda condicionada a esto a fin de comparar de forma realista los resultados.

Como es lógico, el texto de Tolkien presenta una perplejidad bastante menor que las otras. Esto tiene sentido si tenemos en cuenta que estamos hablando del texto original sin ningún tipo de generación, por lo que es natural que presente estas cifras. Aún así, las usaremos para analizarlas.

Luego encontramos un segundo grupo de resultados etiquetados como LSTM, BiLSTM y GRUs. Estos hacen referencia a las redes neuronales recurrentes y con un vistazo ya podemos ver que la perplejidad que muestran es bastante mayor a la de las otras opciones.

Dentro de este grupo, la LSTM es la que muestra el peor rendimiento, seguida de la GRUs y, por último, queda la BiLSTM como aquella que ha demostrado generar un texto de mayor calidad. Esto se corresponde con los resultados que podíamos observar en el punto anterior, dónde veíamos que las mejores cifras en la *crossvalidation* de *Keras* pertenecían a la LSTM.

Aún así, cabe destacar que, tras examinar los resultados de estas redes neuronales, podríamos decir que muchas veces son caóticos y sin sentido alguno. Hay frases que tienen muchos *tokens* repetidos y palabras sin sentido en su construcción.

La BiLSTM es la que mejor rendimiento ofrece en este apartado, también como cabría esperar. También fue la red más pesada de construir, la más compleja y la que presenta un mayor número de capas apiladas.

Por último tenemos el grupo de redes neuronales basadas *Transformers*, cuyos resultados son obviamente mejores a los presentes en el grupo de redes neuronales recurrentes.

Las cuatro opciones de *Transformers*, *Datificate*, *DeepESP* y nuestros modelos *fine tuning* de tienen cifras muy parecidas, aunque obviamente no llegan al estándar de calidad de los párrafos de Tolkien.

De estos cuatro modelos es el de *Datificate* el que ha mostrado mejores resultados pero cabe destacar que, tras hacer esta prueba muchas veces, la varianza entre los resultados de este grupo es mínima, y al ser la diferencia tan pequeña, en ocasiones algunos arrojan mejores cifras que otros.

Los modelos de *Datificate* parecen arrojar mejores resultados, tanto el modelo usado directamente como su versión ajustada, mientras que el modelo de *DeepESP* tiene un rendimiento peor aunque muy cercano en calidad. Como curiosidad mencionar que la versión ajustada de *datificate* es peor que la versión ajustada de *DeepESP*.

El análisis en profundidad muestra un texto de calidad que la mayoría de las veces tiene sentido (coherencia y cohesión), pero que todavía sigue teniendo algunos errores lógicos y fáciles de localizar.

Para el caso de los modelos que hemos ajustado a nuestra tarea, decir que no tienen por qué mejorar la perplejidad, ya que al ajustarlos lo que hemos hecho es básicamente mejorar la

calidad de sus predicciones para que sean más específicas de nuestro *dataset*. Esto puede incluso haberlo perjudicado, aunque como observamos no hay rastros de esto en esta tabla que presentamos.

Como conclusión decir que obviamente el grupo de redes neuronales basadas en *Transformers* es mucho mejor que el de redes neuronales recurrentes, tanto en cifras como en resultados. Ninguno llega a la medida estándar de los párrafos del autor, pero esto sería algo muy complejo.

Los resultados de GPT-2 pueden llegar a ser bastante buenos en algunas ocasiones, mientras que los resultados de las redes neuronales recurrentes caen usualmente en el error y tienden a generar frases repetidas y sin sentidos.

Esto podríamos observarlo con más detenimiento en una valoración cualitativa.

Dada la frase 'Los hobbits se dirigían a…' obtendremos los siguientes resultados usando las redes neuronales recurrentes:

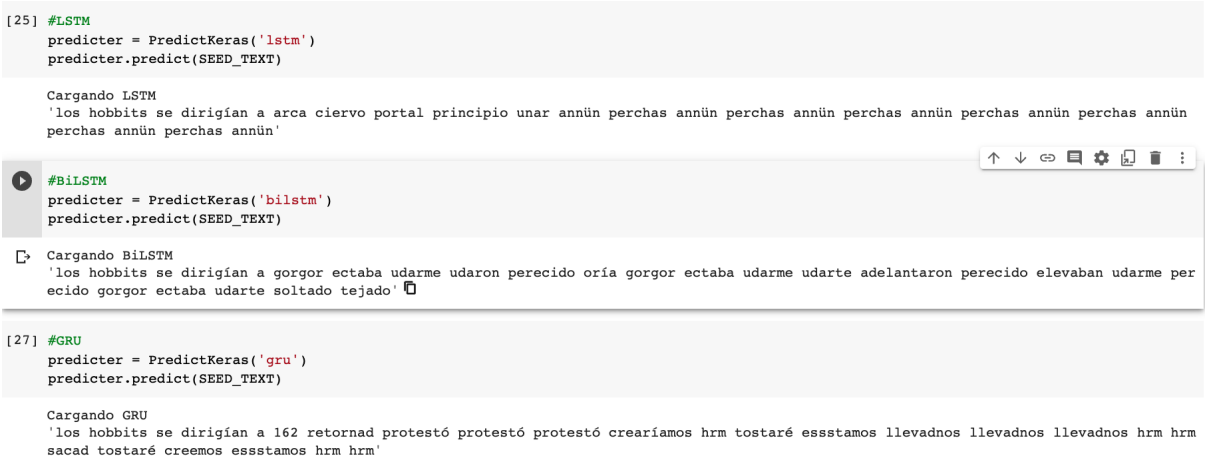

Figura 9: Ejemplo 1 de texto creado por los modelos de generación basados en RNN.

Y para las redes neuronales basadas en *Transformers*::

```
[19] predicter = PredictGPT2('datificate')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'los hobbits se dirigían a la ciudad de Nueva York, lo que provocó un gran impacto. Las noticias de la expedición de hobbits se difundieron ampliamente'

     predicter = PredictGPT2('deepesp')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'los hobbits se dirigían a la estación de autobuses a ver una sesión escolar. El coche estaba listo, y no había ningún inconveniente en que las'

[21] predicter = PredictGPT2('finetune_dat')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'los hobbits se dirigían a la frontera y no hubo ninguna ayuda formal para luchar en ese momento en la retaguardia.\n\nLas tropas del segundo grupo'

[22] predicter = PredictGPT2('finetune_dep')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'los hobbits se dirigían a los pueblos donde acudían las familias nobles del reino, donde se guardaban sus servicios y honras, pues allí los reyes'
```

Figura 10: Ejemplo 1 de texto creado por los modelos de generación basados en Transformers.

Dada la frase 'Vimos las colinas desde lo lejos...' obtendremos los siguientes resultados usando nuestras redes neuronales recurrentes:

```
[21] #LSTM
     predicter = PredictKeras('lstm')
     predicter.predict(SEED_TEXT)

     Cargando LSTM
     'Vimos las colinas desde lo lejos unar annün fauces principio principio fauces principio unar annün perchas annün perchas annün perchas a
     nnün perchas annün perchas annün perchas'

     #BiLSTM
     predicter = PredictKeras('bilstm')
     predicter.predict(SEED_TEXT)

     Cargando BiLSTM
     'Vimos las colinas desde lo lejos ectaba udarte soltado tejado perecido elevaban udarme transcurrido udarnos gorgor ectaba udarme udarte
     adelantaron perecido elevaban udarme perecido gorgor ectaba'

[23] #GRU
     predicter = PredictKeras('gru')
     predicter.predict(SEED_TEXT)

     Cargando GRU
     'Vimos las colinas desde lo lejos 1541 essstamos daur continuad hrm sacad sacad hrm hrm 28302903 essstamos hrm hrm essstamos essstamos hr
     m tostaré 2470 essstamos hrm'
```

Figura 11: Ejemplo 2 de texto creado por los modelos de generación basados en RNN.

Y para las redes neuronales basadas en *Transformers*:

```
[24] predicter = PredictGPT2('datificate')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'Vimos las colinas desde lo lejos hasta el punto de que no hay riesgo de quedar sumergidos.\n\nLa montaña está situada a orillas del río'

     predicter = PredictGPT2('deepesp')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'Vimos las colinas desde lo lejos. Los hombres estaban sentados en círculo en las altas torres, bebiendo con la gente que había en el patio, contemplando'

[29] predicter = PredictGPT2('finetune_dat')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'Vimos las colinas desde lo lejos hasta las puertas de la meseta. Se encuentra en la parte baja de la meseta en la parte alta de la meseta'

[27] predicter = PredictGPT2('finetune_dep')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'Vimos las colinas desde lo lejos y nos dirigimos más al sur. La carretera estaba abierta, con sus laderas, y a nosotros nos pararon cuando'
```

Figura 12: Ejemplo 2 de texto creado por los modelos de generación basados en Transformers.

Dada la frase 'Viajaron durante semanas...' obtendremos los siguientes resultados usando las redes neuronales recurrentes:

```
[31] #LSTM
     predicter = PredictKeras('lstm')
     predicter.predict(SEED_TEXT)

     Cargando LSTM
     'Viajaron durante semanas unar annün perchas annün perchas annün perchas annün perchas annün perchas annün perchas annün perchas annün pe
     rchas annün perchas annün'

     #BiLSTM
     predicter = PredictKeras('bilstm')
     predicter.predict(SEED_TEXT)

     Cargando BiLSTM
     'Viajaron durante semanas gorgor ectaba udarme udaron perecido oría gorgor ectaba udarme udarte adelantaron perecido elevaban udarme pere
     cido gorgor ectaba udarte soltado tejado'

[33] #GRU
     predicter = PredictKeras('gru')
     predicter.predict(SEED_TEXT)

     Cargando GRU
     'Viajaron durante semanas apostando 2950 hrm hrm 28302903 essstamos hrm hrm tostaré 2470 essstamos hrm hrm hrm sacad protestó daur protes
     tó hrm hrm'
```

Figura 13: Ejemplo 3 de texto creado por los modelos de generación basados en RNN.

Y para las redes neuronales basadas en *Transformers*::

```
     predicter = PredictGPT2('datificate')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'Viajaron durante semanas y medio en el edificio con un total de siete personas. Tres de las personas se encontraban con la familia de los trabajadores.\

[32] predicter = PredictGPT2('deepesp')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'Viajaron durante semanas con el pretexto de comerciar con los demás muchachos de la escuela. \n\nPor ejemplo, llegó también a conocer la noticia de'

[35] predicter = PredictGPT2('finetune_dat')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'Viajaron durante semanas por la salida de Bigo, que había aceptado «la oferta» de hacer su siguiente álbum, el cual ya había sido hecho'

[34] predicter = PredictGPT2('finetune_dep')
     predicter.predict(SEED_TEXT)

     Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
     'Viajaron durante semanas sin ser vistos. Pero no tenía sentido ponerse a pensar en nada que pudiera ser, pero no podía hacer nada al respecto si tenía'
```

Figura 14: Ejemplo 3 de texto creado por los modelos de generación basados en Transformers.

Estas imágenes nos permiten hacernos una idea de la calidad de la generación textual de cada uno de nuestros modelos. De esta forma podemos observar que lo que aparece en la tabla de perplejidad que mostramos al principio de este apartado también se cumple cuando lo llevamos a una evaluación cualitativa.

Los resultados de las redes neuronales recurrentes son algo decepcionantes en comparación con aquellos obtenidos por los modelos de *Transformers*. El proceso de implementación en *Keras y Tensorflow* fue mas complejo, llevó mas tiempo y se invirtieron mas horas, sin embargo sus resultados distan de ser satisfactorios. En el lado contrario de la balanza, las

soluciones implementadas en *Transformers* fueron rápidas y concisas de elaborar y sus resultados son bastante lógicos.

Capítulo 5

Conclusiones y trabajo futuro

Este trabajo se ha realizado sobre Generación Automática de Texto. Los objetivos de este trabajo consistían en realizar un estudio del estado del arte en cuestión para crear una base que nos permitiera contextualizar los posteriores experimentos sobre generación textual. En el apartado de conclusiones examinaremos detenidamente cada uno de nuestros objetivos a fin de comprobar qué conclusiones hemos podido extraer de su estudio.

Hemos introducido las ideas de Procesamiento del Lenguaje Natural unido a Inteligencia Artificial, siempre enfocado al campo de la generación de texto, aunque comentando brevemente otras diferentes aplicaciones de las soluciones propuestas. Dentro de esto nos hemos centrado principalmente en la inteligencia narrativa computacional y en la generación de texto fantástico de forma automática.

El estudio de la generación textual, sus posibilidades y limitaciones ha servido para construir una base teórica sobre la que aplicar los conocimientos prácticos demostrados en esta memoria.

Ha sido especialmente interesante estudiar las limitaciones de este campo. Personalmente creo que la escritura está más ligada al arte que a la ciencia. Esto, que podría pensarse en un principio como una limitación, se solventa con relativa facilidad en otras áreas. A día de hoy hemos visto a algunas máquinas construir canciones y elaborar partituras que nada tienen que envidiar a otras escritas por humanos.

En el área de la generación de imágenes, cada vez son mas los softwares que permiten crear impresionantes composiciones pictóricas con tan sólo darle unas indicaciones al programa en cuestión. Sin embargo, música e imágenes pueden traducirse esencialmente en números. Ambas aplicaciones tienen una fuerte base matemática que el ordenador puede entender y procesar con facilidad y que permite alcanzar esos resultados tan sorprendentes.

Sin embargo no es tan sencillo pasar el texto a números y no perder parte del significado en el proceso. Es más difícil en este sentido elaborar un texto coherente que música o imágenes coherentes.

Mediante una base matemática somos capaces de entender qué combinación de colores o notas musicales funcionan y son agradables para nuestros sentidos, sin embargo esta tarea es

realmente compleja en lo que a la construcción de una historia se refiere. Somos capaces de establecer los principios matemáticos que subyacen a la música y la pintura, y llevarlos a la computación con maestría y dominio, sin embargo esto no se traduce de igual forma para la construcción de historias.

Esto tiene una explicación si analizamos la literatura en sí como arte y no como ciencia. Música, pintura, arquitectura etc, existen en la naturaleza de una forma más natural que la literatura en sí. La literatura requiere de un contexto y de experiencia. Es una ciencia que ha evolucionado de la mano de las experiencias del ser humano y cuyo elemento principal, el lenguaje, está presente en nosotros de forma intrínseca. Sin embargo, hemos creado y moldeado el lenguaje en función de nuestras necesidades. Es un elemento único de nosotros. Es posible encontrar manifestaciones musicales y pictóricas en la naturaleza de una forma mucho más orgánica que representaciones literarias.

5.1. Recapitulación

Nuestra intención siempre estuvo relacionada con el estudio de la generación textual y su evolución. A lo largo de las páginas que componen esta memoria hemos centrado gran parte de nuestros esfuerzos en clarificar las bases de esta disciplina y su evolución con el paso de los años.

Muchos han sido los artículos consultados sobre generación narrativa de texto, pues en el centro de la cuestión que intentábamos responder siempre estuvo de fondo la complicación que tienen estos sistemas a la hora de desarrollar largas cantidades de texto manteniendo la coherencia y la cohesión. Por eso varios de los apartados en el Estado del arte se enfocan en la importancia de la narrativa y cómo se enlaza esto con las cascadas de eventos y los *embeddings contextuales*.

También en el estado del arte hemos prestado especial atención a algunos de los enfoques para la generación automática de texto con más éxito hasta la fecha. Los modelos de generación basados en redes neuronales recurrentes y los posteriores modelos de generación basados en *Transformers* han servido para ilustrar el marco teórico y práctico que pretendíamos esbozar al inicio de este trabajo.

Cada uno de ellos con sus ventajas e inconvenientes nos han permitido entender la evolución de un género quizás todavía inmaduro pero que cada vez está despertando más interés y esto se refleja en los continuos avances y en la llegada de nuevos modelos, como el comentado BLOOM [13].

La creación de nuestro propio *dataset*, TolkienGen, nos ha permitido llevar a la práctica lo que definíamos e investigábamos en el Estado del Arte. De esta forma dejábamos preparado un *dataset* que pretende servir como punto de partida para muchos proyectos que se centren en el estudio de la generación textual. Es por ello que hemos tratado siempre de explicar el proceso con claridad y simpleza, a fin de mantener la trazabilidad sobre cada uno de nuestros datos, permitiendo que se puedan seguir nuestros pasos a la hora de recrear el conjunto de datos que hemos creado.

La experimentación ha sido de vital importancia para finalmente extraer las conclusiones que aquí iremos desarrollando. Se ha implementado el marco teórico de tal forma que hemos creado un repositorio en *Github* de libre acceso donde se puede observar todo el código descrito en esta memoria así como el almacenamiento de los datos. Este paso ha sido especialmente largo, pues el desarrollo del repositorio ha sido un trabajo extra que nos ha llevado largas horas de investigación para poder plasmar lo que previamente habíamos estudiado.

Por último llegamos a este apartado, culmen de esta memoria. Tras esta recapitulación llegaremos a las conclusiones, contribuciones, dificultades e impresiones sobre el estado de la generación textual. Con este último capítulo ponemos punto y final a esta memoria, y damos nuestra impresión personal sobre el proceso de creación de la misma.

5.2. Conclusiones

En mi opinión personal, dentro de las diferentes ramas que componen el procesamiento del lenguaje natural, la tarea de la generación textual es una de las más complejas y difíciles de afrontar pues tiene numerosos inconvenientes todavía no cubiertos o cubiertos parcialmente. Precisamente debido a su complejidad creo que aún le falta mucho por evolucionar, pero el estado en el que se encuentra actualmente es bastante bueno comparado con lo que se obtenía años atrás.

El proceso de investigación ha sido bastante gratificante, pues observar las ideas de unos y otros autores y enlazarlas con los pensamientos de uno mismo han servido como un elemento de aprendizaje de valor incalculable. Establecer el marco teórico que esbozamos al inicio de esta memoria ha supuesto un gran aprendizaje que nos ha permitido darnos cuenta de la dimensión del problema al que estábamos intentando hacer frente.

Por otra parte, crear TolkienGen ha sido una tarea igualmente desafiante. Enfrentarse a la recopilación de datos y almacenamiento y manejo de los mismos nos permite hacernos una idea de lo complicado que es mantener los titánicos *datasets* que muchos de los grandes

modelos de la actualiza utilizan para entrenarse. Además, nos ha servido para darnos cuenta de lo avanzado que están estas áreas gracias a los grandes *datasets* en inglés, pero los defectos que encontramos a la hora de afrontarlas utilizando datos en Español.

Para el apartado de resultados mostrados por nuestros modelos de generación, diría que claramente las redes neuronales basadas en *Transformers* son superiores a aquellas mostradas en este trabajo basadas en redes neuronales recurrentes. Especifico lo de claramente porque *Transformers* no implica necesariamente mejor dentro del mundo del aprendizaje automático.

Mi propia experiencia tanto dentro de este máster como en mi día a día profesional me ha demostrado que muchas veces los algoritmos tradicionales de aprendizaje automático o redes neuronales simples pueden servir como solución ideal a diferentes problemas sin necesidad de usar *Transformers.*

El punto fuerte de esta tecnología, y que hemos podido comprobar de primera mano con nuestros resultados, es su capacidad para construir mucho con poco. Es decir, cómo los modelos vienen pre-entrenados, no es necesario realizar un entrenamiento muy grande para obtener buenos resultados.

Nuestro modelo de GPT-2 con *fine tuning* ha demostrado resultados tan buenos como sus hermanos y esto se debe a que, a pesar de que lo hemos entrenado de nuevo, este ya contaba con un fuerte entrenamiento. Este es uno de los puntos mas fuertes de esta arquitectura.

También me gustaría hacer mención a la simplicidad. Creo que la programación tiende a simplificarse con el paso del tiempo. Hago esta reflexión porque el aprendizaje automático, dentro de la programación, también está orientándose a esta tendencia.

Los antiguos modelos de aprendizaje automático requerían de mucha más cantidad de líneas de código de lo que luego podríamos conseguir utilizando *Keras, Tensorflow, Pytorch…* y ahora con *Transformers* el proceso vuelve a simplificarse todavía mas. Construir una *pipeline* utilizando *Hugging Face* es muy sencillo e intuitivo, y cuando la tecnología acabe por asentarse en la red y haya información suficiente (ya que considero que todavía hay desconocimiento sobre ella en comparación a otras soluciones), será muy sencillo construir modelos de aprendizaje automático muy buenos con pasos simples.

Un punto que no hemos mencionado y que cabría la pena destacar, relacionándolo con esto último que hemos comentado es aquel que hace referencia a los datos. Nosotros hemos utilizado un conjunto de datos que hemos perfilado para nuestra tarea; generación textual. Lo bueno de esto es que no hemos tenido que etiquetar ni buscar un *dataset* que estuviese disponible en la web.

Hemos contado con la ventaja de que, para este problema, la anotación es una solución relativamente sencilla. Sin embargo, la ingente cantidad de datos producida primero por las secuencias rodantes realizadas para las redes neuronales recurrentes y luego el entrenamiento de modelos de generación basados en *Transformers* nos hacen comprobar que esta es una tarea que requiere de un alto coste computacional. Es posible que los datos con los que contamos se hubiesen podido mejorar, ya que hemos utilizado libros que, sin limpiar, deben contener muchos espacios que partan los párrafos, puede haber imágenes que dificulten la extracción de texto, errores de formato… Pero la dificultad de empezar y terminar uno de estos entrenamientos ha sido clave a la hora de no poder pulir esta faceta. Menciono esto de nuevo aquí porque los modelos entrenados para generación textual requieren de ingentes cantidades de datos y es claro que en aprendizaje automático cuando mayor sea la calidad del dato, mejores serán los resultados.

Esto no es un problema fácilmente abordable, así que este campo tendrá que seguir lidiando con ello mientras poco a poco vamos consiguiendo más potencia computacional o procesos más optimizados.

Llegados a este punto conviene recapitular el pequeño apartado de objetivos enumerados en la introducción. Primero definimos el objetivo de crear un *dataset* completo en Español (TolkienGen) que pudiese servirnos tanto a nosotros como a futuros proyectos. Este paso se ha completado y hemos definido sus características a lo largo de estas páginas. TolkienGen queda disponible en el repositorio de Github que hemos habilitado para este proyecto. Así mismo, se ha especificado cómo ha sido su creación, adaptación, mantenimiento y elaboración, de tal forma que cualquier lector pueda replicar nuestros pasos sin complicaciones.

En segundo lugar planteamos la comparación de diferentes tipos de redes neuronales en la generación de texto en Español. Esta comparación también se ha llevado a cabo a lo largo de las páginas de este proyecto y podemos ver los resultados mas arriba, en el apartado de evaluación. De la forma mas visual y práctica que hemos podido se ha explicado toda la implementación del código en el repositorio de *text_generation*. Hemos seguido paso a paso los apartados de pre-procesamiento y procesamiento de los datos y hemos detallado la creación de cada uno de los modelos de generación textual propuestos a lo largo de la memoria. Se han desarrollado una LSTM, BiLSTM, GRU, dos modelos de GPT-2 con diferentes entrenamientos y una versión ajustada de uno de ellos. Hemos explicado los puntos generales y específicos de cada uno de estos modelos y se han comparado sus resultados.

Este paso ha sido completado con la creación de un repositorio de *Github* en el que se ha almacenado todo lo realizado para este trabajo. Dicho repositorio puede consultarse al final de la bibliografía [18].

Como tercer objetivo planteamos establecer un protocolo de evaluación y seleccionar las métricas que permitan cuantificar la calidad del texto generado. A este respecto hemos detallado la metodología seguida para crear nuestro conjunto de evaluación y el por qué de las decisiones tomadas a este respecto. También se han comentado diferentes métricas disponibles para generación textual y se ha seleccionado una (perplejidad) explicando detalladamente el por qué de su elección en comparación con el resto de métricas.

Como cuarto y último objetivo nos comprometíamos a analizar los resultados derivados de todo lo comentado anteriormente, permitiéndonos establecer algunas conclusiones que completen este proyecto. Para responder a este paso es que nos encontramos en este último apartado de la memoria. Es a través de estas últimas páginas que pretendemos analizar los resultados y componer las conclusiones que podemos extraer de todo lo estudiado hasta ahora.

De hecho, son los resultados que hemos obtenido los que nos ayudan a responder a las preguntas que nos planteábamos al inicio de este trabajo:

¿Se confirma que también en español las aproximaciones basadas en transformers obtienen mejores resultados? Sí, según nuestra experiencia y tras todo lo detallado anteriormente podemos concluir que las aproximaciones basadas en *transformers* obtienen mejores resultados que los modelos generativos basados en redes neuronales recurrentes. Además, el proceso de implementación es más sencillo y no requiere de tanto código lo que simplifica el proceso.

¿Se corresponde la evaluación cuantitativa propuesta en el marco de evaluación con una exploración cualitativa de los resultados? Sí, en función de los resultados observados podemos concluir que las evaluaciones cuantitativas y cualitativas coinciden. Las imágenes que hemos mostrado en el apartado de resultados nos permiten hacernos una idea de la calidad del texto generado sin cifras y tablas de por medio que nos distancien de la realidad. Los resultados que hemos generado utilizando las aproximaciones basadas en *transformers* tienen mejores cifras en lo que a Perplejidad se refiere y además tienen más sentido cuando las observamos detenidamente.

¿En qué grado un fine-tuning con TolkienGen permite mejorar los resultados en términos de perplejidad? Es interesante porque en nuestro caso, la tarea de ajuste ha empeorado la calidad de los modelos GPT-2 que estábamos usando. Son muchas las razones que podríamos

relacionar con esto. Quizás nuestro *dataset* no sea del todo adecuado para entrenar modelos en Transformers, o quizás debemos modificar los parámetros de la pipeline de entrenamiento nativa de *Hugging Face* para realizar mejores entrenamientos. También es destacable decir que, por problemas computacionales, no hemos podido realizar muchas pruebas sobre los modelos ajustados, lo que en cierta forma limita sus resultados.

5.3. Contribuciones

Para ir cerrando la memoria convendría repasar no sólo los objetivos y preguntas que nos planteábamos al inicio de este proyecto, sino también si las contribuciones que pretendíamos se cumplen y de qué forma.

Las contribuciones de este trabajo son:

1. Elaborar un *dataset* para evaluar la generación de lenguaje en español
2.Proponer un protocolo de evaluación basado en la comparación de los valores de perplejidad entre los textos originales y los textos generados automáticamente.
3. Validar esta propuesta comparando los resultados cuantitativos y cualitativos obtenidos sobre seis aproximaciones diferentes.
4. Comparar dos modelos de GPT-2 con y sin *fine tuning*.

Para este punto contamos con TolkienGen, nuestro conjunto de datos disponibilizado a través del repositorio que hemos creado. Se han especificado sus características y también su creación, de tal forma que hemos puesto a disposición de los lectores todos los pormenores involucrados en su creación.

Para el segundo punto contamos con la métrica y la metodología de evaluación propuestas anteriormente. Hemos establecido una métrica de evaluación; Perplejidad, y hemos creado un conjunto de datos de evaluación que nos ha permitido aplicar esta métrica y compararla con otros resultados obtenidos. De esta forma hemos podido extraer los resultados y concluir cuáles de nuestras aproximaciones tenían más éxito y por qué.

El tercer punto está directamente relacionado con esto. Tomando como referencia nuestras métricas y gracias al conjunto de datos de evaluación se han podido realizar dos estudios, uno cuantitativo y otro cualitativo, y prestar especial atención a ambos de tal forma que podíamos averiguar si lo que observábamos en uno se podía llevar al otro.

El cuarto punto también guarda relación con las métricas seleccionadas, ya que gracias a ellas podemos comparar los resultados de dos modelos diferentes de GPT-2 para los cuáles además tenemos sus versiones *fine tuning*.

Lo cierto es que los resultados del modelo de *datificate* son mejores que los de *DeepESP* tanto a nivel de perplejidad como en los resultados cualitativos. Ambos modelos se han probado por extenso tanto dentro como fuera de esta memoria y los resultados del primero son mejores y tienen más lógica.

Con respecto a sus versiones ajustadas, decir que la de *datificate* sigue siendo mejor que la *DeepESP* y observamos una diferencia similar entre ambos modelos a la hora de calcular su perplejidad. Por otra parte, mencionar que el modelo de *DeepESP* ha perdido menos calidad de texto en su versión ajustada que el modelo de *datificate*.

Es interesante a la par que extraño observar que al hacer *fine tuning* no hemos mejorado la perplejidad de nuestros modelos. Se ha reflexionado mucho a este respecto pero es difícil encontrar una respuesta que nos pueda dar una explicación ajustada a la realidad. Los modelos de aprendizaje automático basados en redes neuronales suelen tener el problema de la 'caja negra', que hace referencia a lo difícil que es explicar e interpretar sus resultados, ya que no tenemos acceso a la totalidad del proceso que se ha ejecutado para llevar a cabo el entrenamiento de estos modelos. Todavía podríamos encontrar explicación a los malos resultados que obtenemos con las redes neuronales recurrentes, ya que hemos participado en todo el pre-procesamiento e ingesta de datos. Sin embargo, cuando utilizamos la *pipeline* de *Hugging Face* cedemos gran parte de esa tarea a procesos nativos que esta incorpora, lo que hace aún más difícil entender qué está sucediendo. Podríamos acudir a la documentación de la librería *Transformers* a fin de obtener claridad en este asunto, pero es igualmente difícil comprobar todo lo que está sucediendo por detrás.

Es posible que la calidad de los datos de TolkienGen no fuese todo lo buena que podría. Pensemos que los modelos de GPT-2 utilizados directamente cuentan con lo que presuponemos que es un buen entrenamiento en máquinas muy potentes que han contado con más preparación y supervisión que los entrenamientos mostrados en esta memoria. Sin embargo, aunque esta podría ser a priori una de las causas de la pérdida de perplejidad, en teoría el *fine tuning* debería mantener la calidad del modelo sólo que ajustándolo a nuestro *dataset*. Como digo, la poca claridad en algunos de los procesos de estos algoritmos dificultan el estudio en sí de estos problemas.

También es posible que al usar la configuración base que viene por defecto en la librería no hayamos podido aprovechar todo el potencial de los modelos. Es importante mencionar también que se realizaron dos *epochs*, debido al alto coste computacional en memoria y

tiempo a la hora de entrenar. Quizás algún *epoch* mas habría servido para mejorar la calidad de los modelos con *fine tuning*.

5.4. Discusión

La realización de este trabajo no ha estado exenta de dificultades. Dentro de las diferentes ramas que componen el procesamiento del lenguaje natural, la generación de texto no ha sido de las que llaman más la atención ni de las más desarrolladas. Esto provoca que la búsqueda de información sea más lenta que en otras áreas.

También me gustaría mencionar a este respecto lo difícil que ha sido enfocar el trabajo única y exclusivamente al Español. No existen muchos artículos de referencia en nuestro idioma, y esto también se traduce en los modelos que hemos utilizado para generar texto. Si hubiésemos realizado la misma tarea pero en inglés habríamos tenido a nuestra disposición multitud de recursos con los que no hemos podido contar. Los modelos de GPT utilizados son todos versiones ajustadas del modelo original en inglés, y esto siempre ocasiona pérdidas en el rendimiento.

También debemos decir que para la realización de los experimentos se ha construido un repositorio en Github cuya elaboración ha sido costosa, principalmente por estar acostumbrados a realizar este tipo de tareas en formato *notebook*. Construir el código en una estructura de repositorios, módulos y submódulos con sus diferentes problemas ha sido difícil y es posible que dicho repositorio todavía tenga que ser mejorado en un futuro, pero ha servido como método de aprendizaje de gran valor.

Otra de las dificultades, ya mencionada en este trabajo, ha sido la de entrenar los diferentes modelos. Todas las soluciones propuestas requerían de una alta capacidad computacional y de prolongados periodos de tiempo. De hecho, en un principio el código se escribió en *Google Colab* a fin de poder probar de forma más cómoda el flujo de trabajo, pero era completamente imposible hacer las ejecuciones tan largas y pesadas que requeríamos, así que se tuvo que optar por trabajar en local con los recursos disponibles.

Es especialmente destacable la dificultad extra que hemos encontrado a la hora de pre-procesar los datos para utilizarlos en las redes neuronales recurrentes construidas en *Keras*. No sólo ha sido un proceso más largo que el de usar los modelos pre-entrenados de *Transformers*, también ha sido más complicado de elaborar ya que requeríamos de unos buenos datos de entrada que permitiesen a la red neuronal predecir correctamente la palabra en cuestión. Esta última dificultad ha sido especialmente significativa y a la vista están los

resultados de la evaluación cualitativa de las redes neuronales recurrentes, que reflejan como no hemos podido generar un texto con sentido en la mayoría de los casos.

Por último, ha estado la dificultad de encontrar información y recursos disponibles. Esta se hace especialmente notoria para los casos de *Hugging Face*, y es la razón por la que este trabajo no presenta ningún modelo de GPT-3 entre sus experimentos. Muchos de los modelos de *Transformers* presentan limitaciones al ser usados, y GPT-3 ha sido de ellos, teniendo que acceder previo pago o con un registro que pedía demasiada información y requería de demasiado tiempo.

Por otro lado, la información sobre *Transformers* es todavía muy reducida en la web y en la literatura, y no hemos podido encontrar muchos ejemplos sobre este tipo de tareas, lo que ha limitado significativamente el margen de acción.

5.4. Trabajo futuro

Debemos seguir investigando. Se ha mencionado en estas páginas, por ejemplo, el modelo BLOOM [13] , que no ha podido ser incluido al aparecer recientemente. Este modelo promete buenos resultados y además ya es multilingüe en su primera versión (incluyendo al español) lo que es muy interesante ya que podríamos haberlo utilizado directamente sin necesidad de tener que recurrir a otras versiones ajustadas de modelos ya existentes.

La investigación en Procesamiento del Lenguaje Natural avanza a pasos agigantados en prácticamente todas sus ramas.

Aunque ya hemos mencionado que la generación textual no acapara el mismo interés que las demás, podríamos mencionar que los agentes conversacionales, entre los cuáles existe una fuerte orientación a la generación de texto, son una herramienta en auge que promete novedades directas o indirectas en este campo.

Particularmente interesante me ha parecido el problema de la creación de contexto y tramas para elaborar un texto coherente y cohesionado a nivel global. A este respecto aún observo carencias en el área y creo que es algo a solventar próximamente.

La calidad de la representación textual ha mejorado exponencialmente con la llegada de los embeddings contextuales y una vez esta calidad alcance unos estándares aceptables y en cierta forma unificados, será posible enfocar los esfuerzos en representar el texto de dicha forma pero en un nivel más global. Contexto y coherencia serán los objetivos a alcanzar por unos modelos de generación de texto que cada vez son más capaces de generar una historia

semánticamente correcta pero que no terminan de alcanzar a elaborar historias interconectadas y, en esencia, vivas.

Como trabajo futuro personal, el repositorio creado para este trabajo tiene que ser mejorado. Se ha estudiado mucho al respecto, pero gran parte de las mejoras que tendrían que hacerse implican un gasto de tiempo con el que no contábamos, por lo que se realizarán a futuro. Por ejemplo, ahora mismo el repositorio no tiene una conexión real entre sus componentes, sino que deben utilizarse casi todos por separado. Esto es algo en lo que hay que seguir trabajando.

Por último y a modo de punto y final, queda como trabajo seguir estudiando la generación textual y las novedades que vayan llegando. Serán muchos los modelos que podremos aprovechar de la mano de *Transformers* y sus resultados acabarán impresionándonos con el tiempo.

Podemos deducir también que el texto que generemos no tendrá en cuenta ninguna secuencia de eventos en concreto. Hemos observado en diferentes artículos que sin un elemento de apoyo que permita introducir el contexto a nuestros algoritmos de predicción, estos no serán capaces de seguir el hilo de la historia. De igual forma, no podrá establecer relaciones entre los diferentes personajes y sus elementos característicos, ya que no hemos introducido ningún esquema de relaciones que permita simplificar esta tarea.

El trabajo futuro en este área debería enfocarse en mejorar la calidad de la representación textual no sólo a nivel de *token*, también a nivel textual completo de tal forma que seamos capaces de representar con calidad grandes cantidades de texto manteniendo la coherencia y la cohesión de la que tanto hemos hablado a lo largo de esta memoria.

Bibliografía

[1] Rield, Mark O. *Computational Narrative Intelligence: A Human-Centered Goal for Artifical Intelligence.* Georgia Institute of Technology. 2016.

[2] Alabdulkarim, Amal. Peng, Xiangyu. *Automatic Story Generation: Challenges and Attempts.* Georgia Institute of Technology. 2021.

[3] Lu, Sidi, et al. *Neural Text Generation: Past, Present and Beyond*. Shanghai Jiao Tong University and University College London. 2018.

[4] Vaswani, Ashish et al. *Attention Is All You Need*. 2017

[5] Medsker, LC Jain. *Recurrent neural networks design and applications.* 1999.

[6] Bird, Steven et al. *Natural Language Processing with Python*. 2009.

[7] Jurafsky, Daniel. *Speech and Language Processing. Third Edition draft.* 2019.

[8] Elkins, Katherine. Chun, Jon. *Can GPT-3 Pass a Writer's Turing Test?.* Journal of Cultural Analytics 5. 2020.

[9] Floridi, Luciano. Chiriati, Massimo. *GPT-3: Its Nature, Scope, Limits, and Consequences.* Minds and Machines 30. 2020.

[10] Wilner, Sean. Woolridge, Daniel. Glock, Madeleine. *Narrative Embedding: Re-Contextualization through Attention*. Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. 2021.

[11] Dale, R. *GPT-3: What's it good for?* Natural Language Engineering 27. 2021.

[12] Lucy, Li, Bamman, David. *Gender and Representation Bias in GPT-3 Generated Stories.* Proceedings of the Third Workshop on Narrative Understanding. 2021.

[13] BigScience. https://bigscience.huggingface.co/ 2022

[14] Karthiek Reddy Bokka, et al. *Deep Learning for Natural Language Processing*. 2019.

[15] datificate/gpt2-small-spanish. https://huggingface.co/datificate/gpt2-small-spanish. 2019.

[16] DeepESP/gpt2-spanish. https://huggingface.co/DeepESP/gpt2-spanish. 2019.

[17] Debeleer. https://www.debeleer.com/escritor/j-r-r-tolkien/ 2022

[18] fjmoronUNED/text_generation. https://github.com/fjmoronUNED/text_generation 2022.

[19] Day, Min-Yuh. *Text Generation Natural Language Generation (NLG).* National Taipe University. 2022.

[20] Goel, Mansi et al. Ratatouille: *A tool for Novel Recipe Generation.* IEEE International Conference on Data Engineering. 2022.

[21] Chernyavskiy, Alexander. *Improving Text Generation via Neural Discourse Planning.* Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining. 2022.

[22] Santhanam, Sivasurya. *Context Based Text-Generation using LSTM networks.* Institute for Software Technology. 2020.

[23] Wu, Jianfeng et al. *Contextual relation embedding and interpretable triplet capsule for inductive relation prediction.* Neurocomputing 505. 2022.

[24] van Stegeren, Judith et al. *Fine-tuning GPT-2 on annotated RPG quest for NPC dialogue generation.* The 16th International Conference on the Foundations of Digital Games (FDG) 2022.

[25] Ormazabal, Aitor, et al. *PoeLM: A Meter- and Rhyme-Controllable Language Model for Unsupervised Poetry Generation.* University of the Basque Country, Facebook AI Research, University of Copenhagen. 2022.

[26] Papineni, Kishore, et al. *BLEU: a Method for Automatic Evaluation of Machine Translation.* IBM T. J. Watson Research Center. 2002.

[27] Lin, Chin-Yew. *ROUGE: A Package for Automatic Evaluation of Summaries.* Text Summarization Branches Out. 2004.

[28] flairNLP/flair. https://github.com/flairNLP/flair. 2020.

[29] Barnejee, Satanjeev, Lavie, Alon. *METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments.* Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization. 2005.

[30] PDFtoText. https://pdftotext.com/es/. 2015

[31] Akbik, Alan, et al. Contextual String Embeddings for Sequence Labeling. 2018.