

**MÁSTER EN INTELIGENCIA ARTIFICIAL
AVANZADA Y APLICADA**



VNIVERSITAT
DE VALÈNCIA

TRABAJO DE FIN DE MÁSTER

**NLP APLICADO A LA PREDICCIÓN DE DATOS
ECOCARDIOGRÁFICOS**

AUTOR:
JOSEP LLUÍS CARRERAS GONZÁLEZ

TUTOR:
JOAN VILA FRANCÉS

SEPTIEMBRE, 2023

Declaración de autoría:

Yo, Josep Lluís Carreras González, declaro la autoría del Trabajo Fin de Máster titulado “NLP aplicado a la predicción de datos ecocardiográficos” y que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual. El material no original que figura en este trabajo ha sido atribuido a sus legítimos autores.

Alaior, 15 de septiembre de 2023



Fdo: Josep Lluís Carreras González

Resumen:

Este proyecto se encuadra en el campo del NLP (*Natural Language Processing*), que es una rama de la Inteligencia Artificial que actualmente está recibiendo mucha atención, sobre todo debido a la aparición de tecnologías como ChatGPT, que han supuesto una verdadera revolución en la sociedad.

El dataset que se utiliza en este proyecto consiste en unos archivos CSV que tienen información de medidas médicas realizadas con un ecocardiograma.

El objetivo de este proyecto es crear un modelo predictivo, más concretamente un clasificador multiclas, que permita predecir un código alfanumérico asociado a diferentes diagnósticos de los ecocardiogramas, predicción que se realiza a partir de un campo de texto libre. En el transcurso del proyecto se toman algunas decisiones para lograr este fin y se testean diferentes modelos, algunos enmarcados en el campo más tradicional del Machine Learning, y otros que se engloban en el campo más novedoso del Deep Learning (redes CNN y RNN), con el fin último de evaluar diferentes opciones y poder elegir la que ofrezca mejores resultados.

Abstract:

This project falls within the field of NLP (Natural Language Processing), which is a branch of Artificial Intelligence that is currently receiving a lot of attention, especially due to the appearance of technologies such as ChatGPT, which have represented a true revolution in society.

The dataset used in this project consists of CSV files that have information from medical measurements performed with an echocardiogram.

The objective of this project is to create a predictive model, more specifically a multiclass classifier, that allows predicting an alphanumeric code associated with different diagnoses from echocardiograms, a prediction that is made from a free text field. During the project, some decisions are made to achieve this goal and different models are tested, some framed in the more traditional field of Machine Learning, and others that are included in the more innovative field of Deep Learning (CNN and RNN networks), with the ultimate goal of evaluating different options and being able to choose the model that offers the best results.

Resum:

Aquest projecte s'enquadra al camp del NLP (*Natural Language Processing*), que és una branca de la Intel·ligència Artificial que actualment està rebent molta atenció, sobretot a causa de l'aparició de tecnologies com ChatGPT, que han suposat una veritable revolució a la societat.

El dataset que s'utilitza en aquest projecte consisteix en uns fitxers CSV que tenen informació de mesures mèdiques realitzades amb un ecocardiograma.

L'objectiu d'aquest projecte és crear un model predictiu, més concretament un classificador multi-classe, que permeti predir un codi alfanumèric associat a diferents diagnòstics dels ecocardiogrames, predicció que es realitza a partir d'un camp de text lliure. En el transcurs del projecte es prenen algunes decisions per aconseguir aquest objectiu i es testegen diferents models, alguns emmarcats al camp més tradicional del Machine Learning, i altres que s'engloben al camp més nou del Deep Learning (xarxes CNN i RNN), per tal d'avaluar diferents opcions i poder triar la que ofereixi millors resultats.

Índice general

1. Introducción	1
1.1. Introducción	1
1.2. Motivación	1
1.3. Objetivos	2
1.4. Organización de la memoria	3
2. Aspectos teóricos	5
2.1. Flujo de trabajo en NLP	5
2.1.1. Adquisición de texto	6
2.1.2. Limpieza y pre-procesado de texto	7
2.1.2.1. Vectores <i>sparse</i>	9
2.1.2.1.1. <i>One-Hot Encoding</i>	9
2.1.2.1.2. <i>Bag of Words (BoW)</i>	11
2.1.2.1.3. TF-IDF	13
2.1.2.2. Vectores densos	14
2.1.2.2.1. <i>Word embeddings</i>	14
2.1.2.2.2. <i>Document embeddings</i>	15
2.1.3. <i>Feature Engineering</i>	16
2.1.4. Modelado	16
2.1.5. Evaluación	17
2.1.5.1. Matriz de confusión	18
2.1.5.2. Curva ROC	21
2.1.6. Implementación	22
2.1.7. Monitorización y actualización del modelo	23
2.2. Modelos de clasificación	23
2.2.1. Modelos de Machine learning	24
2.2.1.1. Naïve Bayes	24
2.2.1.2. Clasificador SVM con SGD	24
2.2.1.3. Clasificador <i>Random Forest</i>	24
2.2.2. Modelos de Deep Learning	25
2.2.2.1. CNN	25
2.2.2.2. RNN	25
2.2.2.2.1. RNN-LSTM	26
2.2.2.2.2. RNN-GRU	27
2.2.2.2.3. Transformers	28

3. Análisis exploratorio y limpieza de los datos	31
3.1. Origen y características del <i>dataset</i>	31
3.2. Análisis Exploratorio de los Datos	31
3.2.1. Fichero codigos.csv	32
3.2.2. Fichero estudios.csv	35
3.2.3. Dataframe cardio_df	36
3.3. Limpieza de texto	39
4. Clasificación	43
4.1. Introducción	43
4.2. Modelos basados en BoW y TF-IDF	43
4.3. Modelos basados en CNN	46
4.3.1. <i>Word embeddings</i> propios	47
4.3.2. <i>Word embeddings</i> de spaCy	50
4.3.3. <i>Word embeddings</i> de fastText	51
4.3.4. Resultados finales con CNN	53
4.4. Modelos basados en RNN	53
4.4.1. RNN-LSTM	54
4.4.1.1. <i>Word embeddings</i> propios	54
4.4.1.2. <i>Transfer learning</i> con <i>word embeddings</i> de GloVe	55
4.4.1.3. Modelo con <i>fine-tuning</i>	57
4.4.2. RNN-GRU	59
4.4.2.1. <i>Word embeddings</i> propios	59
4.4.2.2. <i>Transfer learning</i> con <i>word embeddings</i> de GloVe	60
4.4.2.3. Modelo con <i>fine-tuning</i>	62
4.4.3. Resultados finales con RNN	63
4.5. sciBERT	64
5. Conclusiones	67
5.1. Conclusiones generales	67
5.2. Trabajo futuro	69
Bibliografía	71

Capítulo 1

Introducción

1.1. Introducción

Este Trabajo de Final de Máster, correspondiente al Máster en Inteligencia Artificial Aplicada y Avanzada de la Universidad de Valencia, se enmarca en el campo del NLP (*Natural Language Processing*, Procesamiento del Lenguaje Natural), que se corresponde con una de las asignaturas cursadas en dicho máster pero que, además, se basa en otras asignaturas previas, en las que se han tratado temas tales como la programación en el lenguaje Python, fundamentos matemáticos y las tecnologías de Deep Learning.

1.2. Motivación

El campo de la medicina es uno de los que se considera que se puede beneficiar más de las oportunidades que ofrece la Inteligencia Artificial. Constantemente aparecen en los medios de comunicación y en portales de internet diferentes noticias relacionadas con avances médicos que se desarrollan basados en Inteligencia Artificial. El campo de la medicina ofrece una gran oportunidad para la combinación simbiótica de la Inteligencia Artificial y el ser humano, combinación que fundamentalmente hará cambiar nuestra perspectiva sobre el futuro de la medicina y su impacto en nuestras vidas. Por ejemplo, las tecnologías que se engloban bajo el término de Natural Language Processing pueden

analizar los historiales médicos y la literatura científica médica para buscar y encontrar patrones en ellos.

Los algoritmos de Deep Learning, junto con la supervisión de profesionales de la medicina, pueden crear planes de tratamiento que mejoren los actuales. Se ha demostrado que con la unión de la potencia de cálculo y de detección de patrones de la Inteligencia Artificial y la intuición y conocimientos de estos profesionales se obtienen resultados más satisfactorios que los obtenidos por las dos aproximaciones por separado. Además, se considera que la Inteligencia Artificial puede provocar un gran aumento en la eficiencia de la profesión médica, lo que podrá liberar a sus profesionales de las tareas que interfieren con la conexión humana, tan importante en este campo pero que, sin embargo, cada vez se resiente más de la carga de trabajo rutinario de estos profesionales.

En definitiva, la promesa de la Inteligencia Artificial en la medicina es, a grandes rasgos:

- Proporcionar una mejor perspectiva y análisis de los datos médicos de cada persona.
- Mejorar la toma de decisiones.
- Evitar errores tales como errores de diagnóstico o tratamientos innecesarios.
- Ayudar en la interpretación de los análisis clínicos.
- Mejorar el tratamiento médico.

Este TFM se encuadra dentro de esta revolución en la medicina causada por la Inteligencia Artificial, y a nivel personal es el objetivo del autor que este proyecto suponga un primer paso en una carrera que pueda dirigirse en la aplicación de la Inteligencia Artificial para el beneficio de la sociedad.

1.3. Objetivos

El dataset que se utiliza en este proyecto consiste en unos archivos CSV que tienen información de medidas médicas realizadas con un ecocardiograma.

Este dataset tiene información de medidas realizadas en un hospital con un ecocardiograma, medidas que pasan a formar parte de unos informes médicos realizados sobre diferentes pacientes. El objetivo principal del proyecto es crear un clasificador multiclas que permita predecir un campo con un código alfanumérico que se asocia a diferentes diagnósticos de los ecocardiogramas. La predicción se realiza a partir de un campo de texto libre de conclusiones . El proyecto consiste principalmente en testear diferentes modelos de Machine Learning y Deep Learning (redes CNN y RNN), para así poder evaluarlos y poder elegir el mejor modelo.

1.4. Organización de la memoria

Esta memoria se organiza en una serie de apartados, que van de lo más general a lo más particular:

1. Una introducción que establece la motivación y objetivos del PFM.
2. Un apartado que establece los aspectos teóricos que afectan al proyecto, y explica brevemente las tecnologías que se utilizan en su desarrollo.
3. Un apartado práctico, en el que se explica la limpieza y análisis exploratorio de los datos.
4. El apartado principal, en el que se explican los diferentes modelos que se han estudiado en el desarrollo de este TFM.
5. Un apartado último de conclusiones.

Capítulo 2

Aspectos teóricos

2.1. Flujo de trabajo en NLP

Un proyecto de NLP puede pasar por diferentes etapas típicas y comunes, que se pueden visualizar esquematizadas en la figura 2.1.

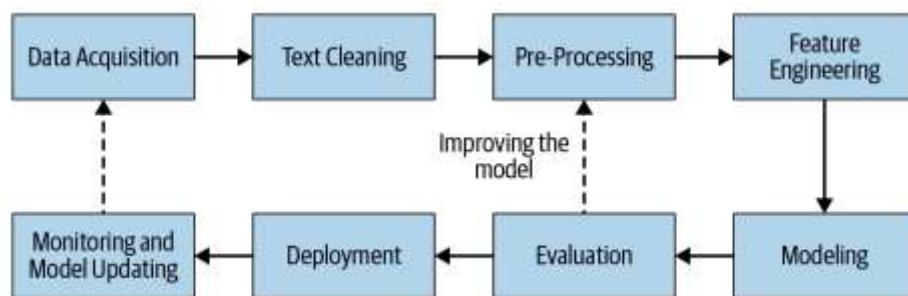


Figura 2.1: Flujo de trabajo en NLP.

El primer paso en el proceso de desarrollar un proyecto de NLP es adquirir datos relevantes a la tarea que queremos resolver. Estos datos raramente estarán limpios, por lo que se hace necesario contar con una etapa de limpieza de los datos. Una vez limpiados, normalmente los datos son muy dispares y necesitan ser transformados en un formato más canónico, lo cual se realiza en la etapa de pre-procesado. La siguiente etapa es de ingeniería de características, conocida como *feature engineering*, con la que obtenemos unos campos o *features* que son más indicados para la tarea que queremos acometer. Estas *features* se convierten a un formato que es más comprensible para los

modelos que vamos a entrenar posteriormente. A continuación, vienen las etapas de modelado y de evaluación, en las que se prueban uno o varios modelos y se comparan utilizando una métrica relevante. A partir de aquí se elige el modelo que funciona mejor y, en entornos reales, se pone en producción. Finalmente, este modelo se monitoriza y, si es necesario, se actualiza para mantener o mejorar su rendimiento.

En la realidad, este proceso puede no ser linear, tal como se muestra en la figura 2.1, a menudo se mueve atrás y adelante entre las etapas consecutivas. Además, puede haber ciclos, normalmente entre las etapas de evaluación y pre-procesado, *feature engineering*, modelado y volviendo a la etapa de evaluación. También existe un ciclo global, que va de la etapa de monitoraje a la de adquisición de datos, normalmente cuando se quiere actualizar el modelo con nuevos datos.

Pasemos a explicar cada una de estas etapas para tener una idea de cómo se desarrolla un proyecto de NLP. Cuando sea conveniente o haya diferentes alternativas, la explicación se centrará en el caso concreto que se estudia en este TFM.

2.1.1. Adquisición de texto

Los datos son la base de cualquier proyecto de ML (*Machine Learning*). Para trabajar en un proyecto de NLP necesitamos tener un *dataset*, y en un caso ideal tendremos acceso directo a un *dataset* que nos servirá para tirar adelante el proyecto. Sin embargo, en la realidad, esta etapa puede ser una de las más difíciles en todo el proyecto, ya que normalmente es complicado disponer de un *dataset* adecuado para el objetivo marcado en el proyecto. Hay muchas maneras de obtener un *dataset*, entre las que destacan:

- A través de consultas SQL a una base de datos.
- A partir de uno o varios ficheros que nos vengan dados (como es el caso de este TFM).
- Realizando *web scrapping* en la web.
- Utilizando un *dataset* público.

En nuestro caso, trabajaremos a partir de ficheros CSV que cargaremos en un *dataframe* de pandas en el contexto de un notebook de Jupyter.

2.1.2. Limpieza y pre-procesado de texto

Hay que tener en cuenta que los algoritmos de Machine Learning suelen trabajar sobre un conjunto de *features* (características de entrada) que normalmente debe ser numérico. Por lo tanto, para realizar el procesado de texto será necesario convertirlo a una matriz numérica, pero antes de esto hay un paso previo, que consiste en limpiar y normalizar el texto que está en lenguaje natural. También es conveniente esta etapa para reducir la ambigüedad y la dimensionalidad del texto, para facilitar el trabajo del algoritmo.

Esta etapa puede consistir en varios pasos:

- División del texto: El texto se debe dividir en palabras (*word tokenization*) y/o en oraciones. Los tokens son las unidades mínimas sobre las que extraeremos las características. Normalmente se realiza con la librería spaCy.
- Limpieza y normalización del texto: Esto puede consistir en diferentes tareas. No se trata de aplicarlas todas, sino que según el proyecto en que estemos trabajando se aplicarán unas u otras. Normalmente las diferentes tareas que deseamos aplicar se agrupan en una función de Python. Las principales tareas que se consideran son:
 - ✓ Convertir el texto a mayúsculas o minúsculas.
 - ✓ Eliminar los acentos.
 - ✓ Eliminar o substituir caracteres especiales.
 - ✓ Anonimizar el texto.
 - ✓ Expandir contracciones: Por ejemplo, en inglés, pasar de *aren't*, que es la contracción de dos palabras, a *are not*.
 - ✓ Quitar los signos de puntuación (puntos, comas...).

- ✓ Quitar palabras de parada (*stopwords*): Pude interesar eliminar palabras muy comunes y que no suelen aportar información. En nuestro caso, utilizaremos un conjunto de *stopwords* en castellano contenido en la librería NLTK.
 - ✓ Corrección ortográfica.
 - ✓ Obtener el lexema o raíz (*stem*) de la palabra (*stemming*): Se trata del morfema que lleva el significado de la palabra. Por ejemplo, en las palabras perro, perra, perras y perros tenemos que la raíz es “perr”.
 - ✓ Obtener el lema de la palabra (*lemmatization*): Se trata de una serie de caracteres en una palabra que forman una unidad semántica. Siguiendo el ejemplo anterior, para el conjunto de palabras perro, perra, perras y perros el lema sería “perro”. Podemos considerar que es lo que encontrariamos al buscar en un diccionario; por ejemplo, en un diccionario encontraremos la palabra “perro”, pero no encontraremos ni “perra”, ni “perras” ni “perros”.
 - ✓ Buscar colocaciones (n-gramas): Un n-grama es una secuencia de n palabras seguidas. Un caso particular son las colocaciones, que son n-gramas con un significado particular en su conjunto (ejemplo: Palma de Mallorca, punto y seguido, *Part Of Speech*, New York, Iron Maiden...).
- Análisis morfológico y sintáctico:
 - ✓ Morfología: Se considera el *POS tagging* (POS = *Part Of Speech*), que asigna a cada token su función morfológica (es decir, si se trata de un nombre, un verbo, un adjetivo...). Se puede utilizar para eliminar ciertos tipos de palabras, como por ejemplo los determinantes, que no suelen aportar información, sino que solamente sirven para ofrecer corrección gramatical; o se puede utilizar para quedarnos solo con ciertos tipos de palabras, como los nombres y adjetivos. Otra aplicación puede ser el análisis de estilo de escritura o de autor, mediante el análisis frecuencial de los tipos de palabras en un texto.
 - ✓ Análisis gramatical: Por ejemplo, el *dependency-parsing* determina las relaciones gramaticales que existen entre las palabras de una oración.
 - ✓ *Named Entity Recognition*: Se utiliza para identificar entidades propias, tales como empresas, instituciones, nombres de personas, países, ciudades...

2.1.3. Preprocesado

Se convierte el texto a una serie de valores numéricos que nos permitirán aplicar algoritmos de Machine Learning o Deep Learning. Principalmente tenemos dos tipos: las características simples y el modelo de espacio de vectores:

- Características simples:

Son tareas muy simples y que realmente se suelen utilizar poco, como por ejemplo la extracción de la longitud del texto (en caracteres o en palabras), el número o ratio de palabras fuera del diccionario, número de emojis, número de hashtags, etc.

- Modelo de espacio de vectores:

Consiste en convertir el texto en un espacio vectorial, es decir, convertir cada frase o cada palabra en un vector numérico de unas dimensiones fijas. Hay dos grandes grupos de técnicas para generar estos vectores numéricos: los vectores *sparse* y los vectores densos.

2.1.3.1. Vectores *sparse*

Se trata de técnicas basadas en la distribución de las palabras y que generan vectores de tipo *sparse* (*sparse* significa que casi todos sus valores son cero y solo unos pocos valores son distintos a cero). Suelen ser más fáciles y rápidas de entrenar que las basadas en vectores densos. Hay tres técnicas muy importantes: *One-Hot Encoding*, *Bag of Words* y *TF-IDF*.

2.1.3.1.1. One-Hot Encoding

Con este método, a cada palabra w en el vocabulario global del corpus se le asigna un identificador ID entero único, comprendido entre 1 y V , donde V es el tamaño del corpus. Cada palabra se representa por un vector binario, de ceros o unos, de dimensión V . Al final, para cada palabra, este vector está formado por ceros excepto en el índice w_{ID} , que es 1.

Veamos un sencillo ejemplo para entender el concepto. Consideremos un corpus muy sencillo, de solo 4 textos:

- Mi gato es peludo
- Tu perro es pequeño
- Me gusta mi gato
- No me gusta tu perro

El corpus de estos 4 textos es [mi, gato, es, peludo, tu, perro, pequeño, me, gusta, no] y, como vemos, está formado por 10 palabras distintas. Lo primero que debemos hacer es mapear cada una de las 10 palabras a unos IDs únicos:

Palabra	ID
mi	1
gato	2
es	3
peludo	4
tu	5
perro	6
pequeño	7
me	8
gusta	9
no	10

Tabla 2.1: Mapeado entre palabras y IDs.

Consideremos ahora el primer texto: “mi gato es peludo”. Cada vector tendrá 10 dimensiones. Por ejemplo, la primera palabra que aparece en el corpus, “mi”, tendrá asociado el vector [1 0 0 0 0 0 0 0 0], y la cuarta palabra, “peludo”, tendrá el vector [0 0 0 1 0 0 0 0 0]. De esta manera, este primer texto se representará como:

```
[[[1 0 0 0 0 0 0 0 0],  
[0 1 0 0 0 0 0 0 0],  
[0 0 1 0 0 0 0 0 0],  
[0 0 0 1 0 0 0 0 0]]]
```

Este método tiene una serie de ventajas:

- Es muy intuitivo.
- Fácil de implementar.

En cambio, sus inconvenientes son:

- El tamaño de un vector *one-hot* es directamente proporcional al tamaño del vocabulario del corpus y, generalmente, estos corpus pueden ser muy grandes. Esto resulta en vectores muy *sparse*, con muchos ceros y muy pocos unos. Afortunadamente, Python posee maneras de manejar y guardar estos vectores de una manera muy eficiente.
- Trata las palabras como unidades atómicas y no tiene noción de la similitud o disimilitud entre las palabras. Por lo tanto, es un método muy pobre para capturar el significado de las palabras en relación con el resto de las palabras.
- Esta representación no tiene una longitud fija. Por ejemplo, un texto de 8 palabras tendrá una representación con 8 vectores, mientras que uno con 5 palabras lo tendrá de 5 vectores.
- No puede tratar las palabras que no aparecen en el corpus de entrenamiento. Esto se conoce como el problema de *Out Of Vocabulary* (OOV: Fuera de vocabulario).

2.1.3.1.2. Bag of Words (BoW)

La idea principal de esta técnica es representar el texto como una bolsa o colección de palabras, ignorando el orden y el contexto. La intuición básica es que asume que el texto que pertenece a una determinada clase en el *dataset* se caracteriza por un conjunto único de palabras. Por lo tanto, si dos textos contienen aproximadamente las mismas palabras entonces es muy probable que pertenezcan a la misma clase. Así, se concluye que analizando las palabras presentes en un texto se podrá identificar la clase a la que pertenece.

En esta técnica se genera un vocabulario que será común para todos los documentos, que contendrá todos los términos únicos. Luego se crea una matriz con tantas filas como número de documentos o muestras y tantas columnas como términos únicos. De esta manera, cada texto o muestra quedará representado por una fila de la matriz, en la que

cada columna representará la frecuencia con la que aparece un término del vocabulario en el texto.

Si tomamos un ejemplo sencillo de un conjunto de 6 textos cortos, que en total contienen 15 palabras distintas, la representación de la matriz *sparse* podría ser la siguiente:

```
array([[1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
       [1, 0, 0, 2, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0],  
       [0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1],  
       [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1],  
       [0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1]])
```

Figura 2.2: Ejemplo de matriz *sparse*.

Aquí vemos como tenemos 6 filas, correspondientes a los 6 textos, y 15 columnas, una para cada palabra. Por ejemplo, en la primera fila tenemos solo 3 de las 15 posiciones con valores distintos a cero, cada una con un uno, lo cual nos dice que este texto contiene una vez la primera palabra, una vez la segunda palabra y una vez la cuarta palabra. Realmente Python no guarda la información de esta manera, lo cual sería muy poco eficiente, sino que guarda una lista con las posiciones que contienen un valor diferente de cero y para cada elemento de esta lista guarda la posición y el valor de esa posición.

Las ventajas de este método son:

- Igual que el *One-Hot Encoding*, BoW es muy simple y fácil de interpretar.
- Los vectores permiten calcular la similitud entre dos textos usando una medida de similitud como la similitud coseno. Así, los textos que tengan las mismas palabras tendrán sus representaciones vectoriales muy cercanas.
- Tenemos una codificación de longitud fija para cualquier texto de longitud arbitraria.

En cambio, sus inconvenientes son:

- El tamaño de los vectores aumenta con el tamaño del vocabulario. Una manera típica de mejorar este aspecto es limitando el vocabulario a un número n de palabras más frecuentes.
- No captura la similitud entre diferentes palabras que tienen el mismo significado.
- No maneja palabras OOV (*Out Of Vocabulary*).
- Tal como su nombre indica, es una bolsa de palabras, por lo que el orden de las palabras en el texto se pierde.

En la práctica, usaremos la clase CountVectorizer de scikit-learn para utilizar BoW, que convierte un conjunto de documentos de texto en una matriz de características BoW.

2.1.3.1.3. *Term Frequency – Inverse Document Frequency (TF-IDF)*

Es una técnica basada en BoW, en la cual se aplica una ponderación con un doble objetivo: restar importancia a los términos más comunes y promover los menos comunes. Se calcula a partir del producto de dos términos:

- **tf(w, D)**: Frecuencia del término w en el documento D . Realmente es lo mismo que la matriz BoW.

$$tf(w, D) = \frac{\text{Número de ocurrencias del término } w \text{ en el documento } D}{\text{Total de términos en el documento } D}$$

- **idf(w, D)**: Frecuencia inversa del término w , calculada a partir del logaritmo del número total de documentos, D , dividido por $df(w)$, el número de documentos en los que el término w aparece. El logaritmo se utiliza para suavizar este término.

$$idf(w, D) = \log_e \left(\frac{\text{Total de documentos en el corpus}}{\text{Número de documentos con el término } w \text{ en ellos}} \right)$$

Al final, la fórmula total es:

$$tfidf(w, D) = tf(w, D) \times idf(w, D) = tf(w, D) \times \log \left(\frac{C}{df(w)} \right)$$

Sus ventajas son:

- Al igual que con BoW, los vectores permiten calcular la similitud entre dos textos usando una medida de similitud como la similitud coseno.
- Tiene el resto de las ventajas que tiene BoW.
- En principio, TF-IDF suele dar mejores resultados que BoW, ya que resta valor a las palabras menos importantes.

En cambio, sus inconvenientes son:

- Trata las palabras como unidades atómicas, lo cual dificulta establecer relaciones entre las palabras.
- Son muy *sparse* (poco densos) y con dimensiones muy grandes.
- No manejan palabras OOV.

En la práctica, así como en scikit-learn tenemos la clase CountVectorizer para BoW, para TF-IDF utilizaremos la clase TfidfVectorizer, que convierte una colección de documentos de texto en una matriz de características TF-IDF.

2.1.3.2. Vectores densos

Son vectores normalmente con muchas menos dimensiones que los vectores *sparse* y en los que todos o casi todos sus valores son distintos de cero. Estos vectores están relacionados con el significado semántico del texto.

A diferencia de los modelos *sparse*, en los que el vector numérico no guarda ninguna relación con el significado de las palabras, los modelos densos codifican cada palabra o oración como un vector numérico continuo que sí tiene información semántica.

Tenemos de dos tipos: *word embeddings* y *document embeddings*.

2.1.3.2.1. *Word Embeddings*

Los vectores se aplican a las palabras o tokens y, por lo tanto, se genera un vector para cada palabra. Este método consiste en realizar una representación vectorial de los

términos contenidos en el vocabulario, en un espacio vectorial continuo que se basa en la similitud semántica y del contexto. Se basa en dos conceptos semánticos: la semántica distribuida, según la cual las palabras que aparecen en un mismo contexto tienen muchas posibilidades de tener significados similares, y la connotación, según la cual el significado de una palabra se puede extraer de su contexto.

Los *word embeddings* se obtienen a partir de unos modelos que se entrenaen según un modelo no supervisado, a partir de muchos textos. Tienen la ventaja de que palabras similares están representadas por vectores similares. Además, las analogías que pueda haber entre palabras se codifican como diferencias en los vectores de estas palabras. En un *word embedding*, cada palabra se mapea a su correspondiente vector.

Para medir la similitud entre vectores se tiene que utilizar una función de similitud, y se suele utilizar la *similitud coseno*, que mide el coseno del ángulo que forman dos vectores entre sí; cuanto menor sea su valor, más similares son dos palabras.

Tenemos 3 modelos de *word embeddings* muy conocidos:

- word2vec: Es el modelo más conocido. Está basado en una red neuronal muy simple, de solo 3 capas, que a través de un método de enventanado intenta predecir cada palabra en función de su contexto. Este modelo es, pues, predictivo. Las representaciones son de baja dimensionalidad, típicamente entre 50 y 500 dimensiones.
- GloVe (GLObal VEctors): Se basa en la matriz de co-ocurrencia global entre palabras. Como esta matriz suele ser enorme, se le aplica una reducción de dimensionalidad. No es un modelo predictivo sino estadístico. spaCy utiliza los *word vectors* de GloVe.
- fastText: Está basado en el word2vec. Este modelo aplica el modelo word2vec a un vocabulario, pero este vocabulario es especial ya que no es de las palabras en cuestión, sino que cada palabra está representada por una secuencia de n-gramas de caracteres.

2.1.3.2.2. Document embeddings

En este caso los vectores se aplican a todo un documento. Se trata de una representación vectorial con longitud fija para documentos, independientemente de su longitud. Hay dos aproximaciones, una basada en la distribución de palabras en el documento y otra basada en la semántica distribuida.

2.1.4. Feature Engineering

Se trata de una serie de métodos que persiguen el objetivo de disponer de datos adecuados que puedan alimentar un posterior modelo. Su objetivo principal es capturar las características del texto en un vector numérico que pueda ser entendido por los algoritmos de ML.

2.1.5. Modelado

El modelado es el proceso mediante el cual se genera un modelo que nos permitirá realizar la función que se espera de él, y que en nuestro caso se trata de una clasificación. Así pues, de los diferentes tipos de modelos que se pueden generar con Machine Learning, nos centraremos en el modelado de modelos clasificadores.

Podemos encontrarnos con tres tipos distintos de problemas de clasificación:

1. Binaria: Se trata de clasificar entre dos clases mutuamente excluyentes. Un ejemplo sería un clasificador de spam.
2. Multi-clase: Se clasifica entre varias clases que son mutuamente excluyentes. Nuestro problema en este TFM se enmarca en este tipo de clasificador.
3. Multi-etiqueta: Se clasifica entre varias clases que no son excluyentes. Por lo tanto, un documento puede pertenecer a varias clases. Un ejemplo podría ser la clasificación de tweets según los estados de ánimo que se infiere de los mismos (enfado, alegría...).

Un proceso de clasificación pasa por 3 etapas diferenciadas:

1. Entrenamiento:

El entrenamiento de un clasificador pasa, asimismo, por diversas etapas:

- Partimos de un conjunto de entrenamiento, que consiste en un corpus de texto y sus etiquetas.
- A partir del corpus, se generan los vectores de características mediante un vectorizador.
- Los vectores de características y las etiquetas alimentan un algoritmo de Machine Learning.

2. Validación:

Una vez entrenado un clasificador, pasamos a la validación, que también tiene varias etapas:

- Introducimos al modelo textos del conjunto de test, debidamente vectorizados con el mismo vectorizador con el que se ha entrenado el modelo.
- El modelo nos da un resultado.
- El resultado se compara con la etiqueta real.
- Se calcula una métrica para evaluar el modelo.

3. Inferencia:

Por último, tenemos la inferencia, etapa en la cual:

- Se introducen nuevos textos, previamente convertidos con el mismo vectorizador.
- El modelo realiza la predicción, lo cual nos da una etiqueta. El objetivo del modelo es, precisamente, proporcionarnos esta etiqueta predicha a partir de un dato nuevo.

2.1.6. Evaluación

En la etapa de validación necesitamos tener algún tipo de métrica para poder evaluar el rendimiento del modelo. Hay muchas métricas que pueden ser utilizadas, pero dependiendo del tipo de problema que tengamos unas métricas pueden ser más adecuadas que otras.

2.1.6.1. Matriz de confusión

Una herramienta muy útil para el caso de clasificadores binarios es la matriz de confusión, a partir de la cual se definen diferentes métricas. Si consideramos el caso de un modelo clasificador binario, tendremos que una de las dos clases se considerará la clase positiva y la otra será la negativa (esto es meramente un tema de nomenclatura). Por ejemplo, si consideramos un modelo detector de spam en emails, podemos considerar que la clase positiva es cuando un mail es spam y la clase negativa es cuando no lo es. El modelo se evalúa sobre un conjunto de test, y a partir de los resultados predichos por el modelo se puede definir la matriz de confusión. Se trata de una matriz 2x2 en la que en las columnas se representan los valores reales, con una columna para la clase positiva y otra para la negativa, y en las filas se representan los valores predichos por el modelo, con una fila para los casos en que se predice la clase positiva y la otra para cuando se predice la clase negativa. Esto define 4 celdas, tal como podemos ver en la figura.

		true class		predicted class	total
		EFR	LFR		
predicted class	EFR	True Positives (TP)	False Positives (FP)		predicted EFR
	LFR	False Negatives (FN)	True Negatives (TN)		predicted LFR
		true EFR	true LFR		

$$PR = \frac{TP}{TP+FP}$$

$$RE = \frac{TP}{TP+FN}$$

$$CA = \frac{TP+TN}{TP+TN+FP+FN}$$

$$F_1 = \frac{2TP}{2TP+FP+FN}$$

Figura 2.3: Matriz de confusión.

Estas 4 celdas definen 4 conceptos:

- Verdaderos Positivos (*True Positives – TP*): Casos en los que las muestras que pertenecen a la clase positiva son correctamente predichas como positivas.
- Falsos Positivos (*False Positives – FP*): Casos en los que las muestras que pertenecen a la clase negativa son incorrectamente predichas como positivas.
- Falsos Negativos (*False Negatives – FN*): Casos en los que las muestras que pertenecen a la clase positiva son incorrectamente predichas como negativas.

- Verdaderos Negativos (*True Negatives – TN*): Casos en los que las muestras que pertenecen a la clase negativa son correctamente predichas como negativas.

A partir de estos 4 conceptos se definen diferentes métricas:

- Accuracy: Es la fracción de muestras que han sido correctamente predichas en relación con el total de predicciones realizadas. Se utiliza cuando la variable de salida es categórica o discreta. Se usa principalmente en tareas de clasificación.

$$ACC = (TP + TN) / (TP + TN + FP + FN)$$

- Precision: Fracción de muestras predichas como positivas (TP y FP) que han sido correctamente clasificadas (TP). Por lo tanto, responde a la pregunta: entre las muestras que se han clasificado como positivas, ¿cuántas han sido correctamente clasificadas?

$$PR = TP / (TP + FP)$$

- Recall: Fracción de muestras que realmente son positivas (TP + FN) que han sido correctamente clasificadas (TP). Es decir, responde a la pregunta: entre las muestras positivas, ¿cuántas han sido correctamente clasificadas?

$$RE = TP / (TP + FN)$$

- F1: Media harmónica entre *precision* y *recall*. Por lo tanto, combina la *precision* y el *recall* en una sola métrica. Si tanto la *precision* como el *recall* son altos, F1 será alto. Sin embargo, si uno cualquiera de ellos es bajo, F1 será bajo. El objetivo de esta métrica es medir si tanto la *precision* como el *recall* son altos, y dar un toque de alerta cuando uno de ellos es bajo.

$$F1 = 2TP / (2TP + FP + FN)$$

Al evaluar un modelo podemos tener dos tipos de errores, Falsos Positivos y Falsos Negativos, y según el tipo de problema estos diferentes tipos de errores pueden tener diferentes niveles de importancia. Por ejemplo, en un clasificador de spam interesa que se detecten todos los correos que son o no son spam, pero en caso de error seremos más permisivos con el hecho de que un spam sea clasificado como no-spam (FN) que con el caso de que un correo no-spam sea clasificado como spam (FP). En cambio, en el filtro de seguridad de un aeropuerto interesa detectar absolutamente todos los casos en que un pasajero lleve un arma y no se admite dejar pasar a un pasajero con un arma (FN), aunque esto signifique que se tienen que detectar muchos casos en que pasajeros sin armas se detectan como si llevasen armas (FP). Para el primer ejemplo (spam), pues, interesa tener pocos FP, lo cual implica tener una *precision* alta. En cambio, para el segundo ejemplo (filtro de seguridad), interesa tener pocos o ningún FN, lo cual implica que deseamos tener un *recall* alto. Esto se puede evaluar a partir de una modificación de la métrica F1, que se llama F_β , que introduce en la fórmula de F1 un parámetro β que modula la importancia relativa entre precisión y recall.

$$F_\beta = (1 + \beta^2)PR / (\beta^2P + R)$$

donde $P = precision$ y $R = recall$.

Todo esto que hemos visto hasta ahora se refiere al caso de un clasificador binario, pero ¿qué pasa en el caso de un clasificador multiclas? En este caso, se puede considerar una matriz de confusión con tantas filas y columnas como clases distintas tengamos. En la figura se puede ver una comparación entre el caso binario y el caso multiclas (con 3 clases distintas).

		True Class		
		A	B	C
Predicted Class	True/Actual Class	A	TP_A	E_{BA}
		B	E_{AB}	TP_B
C	E_{AC}	E_{BC}	TP_C	

		True/Actual Class	
		Positive (P)	Negative (N)
Predicted Class	True (T)	True Positive (TP)	False Positive (FP)
		False Negative (FN)	True Negative (TN)
		$P=TP+FN$	$N=FP+TN$

Figura 2.4: Comparación entre una matriz de confusión binaria y una matriz multiclas.

En este caso tenemos que, al igual que en el caso binario, la diagonal identifica las muestras clasificadas correctamente, y los elementos de la matriz fuera de la diagonal identifican las clasificaciones erróneas. Por ejemplo, la celda marcada con E_{BA} representa las muestras del tipo B que han sido erróneamente clasificadas como del tipo A.

2.1.6.2. Curva ROC

La curva ROC es una representación gráfica de cómo va variando una métrica llamada *specificity* (especificidad) al ir variando otra métrica llamada *sensitivity* (sensitividad). La *sensitivity* es lo mismo que el recall:

$$sensitivity = RE = TP / (TP + FN)$$

Y la *specificity* es la capacidad del modelo para identificar las muestras negativas:

$$specificity = TN / (TN + FP)$$

Para utilizar la curva ROC debemos hacer la suposición de que el modelo retorna la predicción como un valor continuo o, mejor dicho, como una probabilidad. Esto es verdadero para modelos tales como los clasificadores basados en *logistic regression*, en los que la predicción no es una clase sino un valor entre 0 y 1. Lo que normalmente hacemos con este valor entre 0 y 1 es tomar un valor umbral, como por ejemplo 0,5, y clasificar como positivo cada punto que recibe una predicción mayor o igual que el umbral y los que reciben una predicción menor como negativos. Sin embargo, este valor puede ser cualquier valor, no tiene porque ser 0,5. Así pues, el procedimiento de la curva ROC consistirá en variar este umbral entre 0 y 1 y representar la *sensitivity* y la *specificity* del modelo para cada valor de este umbral.

De esta manera, obtenemos una gráfica como la que se ve a la izquierda en la figura 2.5. Para obtener una métrica a partir de esta gráfica, se calcula el área que queda por debajo de la curva (AUC: *Area Under Curve*), lo cual nos dará un valor entre 0 y 1. La interpretación se puede ver en la figura de la derecha: un valor de AUC de 0,5 es

equivalente a lo que no ofrecería un modelo que prediga de manera aleatoria, y sirve como referencia a batir. El caso extremo en que $AUC = 1$ es el caso ideal. Interesa tener el mayor valor posible de AUC .

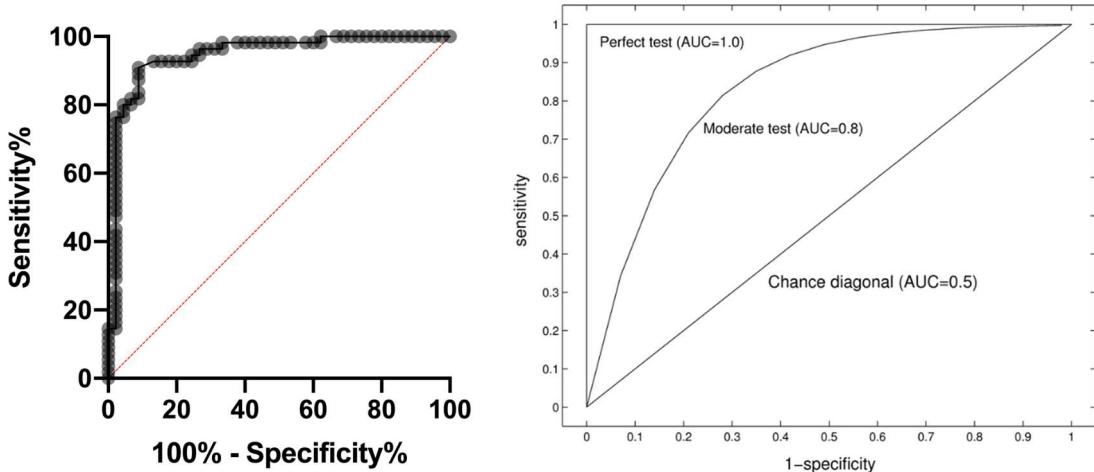


Figura 2.5: Curvas ROC.

Hay más métricas, pero las derivadas de la matriz de confusión y la curva ROC son las más utilizadas.

2.1.7. Implementación

Una vez tenemos un modelo que nos convence, en una situación real o no académica interesaría poner en producción el modelo. En la mayoría de las aplicaciones prácticas, el módulo NLP que se implementa forma parte de un sistema más grande (por ejemplo, un sistema clasificador de *spam* forma parte de una aplicación de email). Una vez que tenemos un modelo que consideramos satisfactorio, este debe ser puesto en producción como parte de este sistema más grande.

Este TFM, al tratarse de un trabajo académico, no contempla esta situación, pero no está de más recordar que en entornos no académicos el paso más importante y que da valor a todo el trabajo realizado en las etapas anteriores es la puesta en producción del modelo.

2.1.8. Monitorización y actualización del modelo

Teniendo el modelo en producción, interesa ir monitorizando su funcionamiento. Puede darse el caso que el modelo deba ser actualizado, principalmente porque tenemos nuevos datos y interesa tener una versión más nueva del modelo que haya sido entrenado teniendo en cuenta estos nuevos datos. Así pues, el modelo sería actualizado, empezando el ciclo de nuevo. Al igual que en la etapa anterior, este TFM, al tratarse de un trabajo académico, no contempla esta etapa.

2.2. Modelos de clasificación

En Machine Learning, la clasificación es el problema de categorizar un dato en una o más clases conocidas. Los datos pueden ser originalmente de diferentes formatos, tales como texto, audio, imagen o numérico. Así pues, la clasificación de texto es un ejemplo particular del tema general de clasificación, donde la entrada es texto y el objetivo consiste en categorizarlo dentro de una o varias clases pertenecientes a un conjunto predefinido de posibles clases.

Previamente hemos visto los diferentes pasos de un proyecto de NLP. Para el caso más concreto de un sistema clasificador de texto, los pasos que típicamente se contemplan son:

1. Crear o obtener un *dataset* etiquetado y adecuado para la tarea.
2. Dividir el *dataset* en dos partes: entrenamiento y test, y decidir una métrica.
3. Transformar los textos en vectores.
4. Entrenar un clasificador usando los vectores y las correspondientes etiquetas del conjunto de entrenamiento.
5. Utilizar la métrica para evaluar el rendimiento con el conjunto de test.
6. Poner el modelo en producción.

Normalmente se itera entre los pasos 3 y 5 para poder explorar diferentes modelos o variantes de estos.

2.2.1. Modelos de Machine Learning

Dentro del conjunto de modelos de clasificación de Machine Learning disponibles, probaremos algunos de ellos, basándonos en la librería scikit-learn.

2.2.1.1. Naïve Bayes

Se trata de un clasificador probabilístico que utiliza el teorema de Bayes para clasificar textos basándose en la evidencia vista en el conjunto de entrenamiento. Estima la probabilidad condicional de cada *feature* de un texto para cada clase basándose en la ocurrencia de dicha *feature* en esa clase y multiplica las probabilidades de todas las *features* de un texto dado para calcular la probabilidad final de clasificación en cada clase. Por último, elige la clase con más probabilidad.

2.2.1.2. Clasificador SVM con SGD

Modelo clasificador de tipo SVM, regularizado y con entrenamiento con aprendizaje con SGD (*Stochastic Gradient Descent*). El gradiente del *loss* se estima una muestra cada vez y el modelo se actualiza según el valor marcado por el *learning rate*.

Los modelos de SVM (*Support Vector Machine*) son clasificadores discriminativos, igual que, por ejemplo, los modelos de regresión logística. Su objetivo es buscar un hiperplano óptimo en un espacio con más dimensiones que el original, que pueda separar las clases con el máximo posible margen. También pueden aprender separaciones entre clases de tipo no lineal, a diferencia de la regresión logística.

2.2.1.3. Clasificador Random Forest

Estos modelos son un ejemplo de los modelos de tipo *bagging ensemble*, y funcionan entrenando muchos árboles de decisión sobre subconjuntos aleatorios de las muestras del conjunto de entrenamiento, y luego promedian sus predicciones para ofrecer la predicción final.

El concepto de *bagging ensemble* consiste en que se consideran subconjuntos aleatorios del conjunto de entrenamiento, con reemplazo, y se entrena un modelo diferente sobre cada uno de estos subconjuntos. Estos modelos se denominan *weak learners*. A partir de estos *weak learners* se genera lo que se denomina el *strong learner*, que es una combinación de los *weak learners*, y la predicción se calcula a partir de una votación entre ellos.

2.2.2. Modelos de Deep Learning

Deep Learning es una familia de algoritmos de Machine Learning en los que el aprendizaje se desarrolla a partir de arquitecturas de red neuronal multicapa. Dos de las estructuras más utilizadas para realizar clasificación de texto son las redes neuronales convolucionales (CNN: *Convolutional Neural Networks*) y las redes neuronales recurrentes (RNN: *Recurrent Neural Networks*).

2.2.2.1. CNN

Las CNN típicamente consisten en una estructura secuencial de capas convolucionales y de *pooling* como capas internas. En el contexto de la clasificación de texto, las CNN aprenden las *features* más útiles en vez de tomar la colección entera. La salida de la red tiene tantas salidas como clases se consideren en el problema. Se define una serie de capas, con sus correspondientes funciones de activación, y se compila la red considerando algunos parámetros que se deben elegir, como el optimizador a utilizar, la función de pérdidas (*loss function*) y la o las métricas a considerar. Una vez definida la red, el siguiente paso es entrenarla y evaluarla, sobre los conjuntos de entrenamiento y test, respectivamente.

2.2.2.2. RNN

El lenguaje natural es inherentemente secuencial. Una frase en cualquier lengua fluye en una dirección. Por lo tanto, un modelo que pueda leer progresivamente un texto de entrada de principio a final y de manera secuencial puede ser muy útil para la

comprensión del texto. Precisamente las redes RNN están especialmente diseñadas para tratar adecuadamente este aprendizaje secuencial. Estas redes tienen unidades neuronales que son capaces de recordar lo que han procesado hasta entonces. Esta memoria es temporal, y la información es guardada y actualizada en cada paso temporal, al leer la red la siguiente palabra de entrada. La figura 2.6 muestra cómo se suele representar una célula recurrente, su equivalente desenrollado y cómo sigue la pista de la entrada en diferentes pasos temporales. Este tipo de redes suelen ser muy potentes y suelen trabajar muy bien en las tareas de NLP, tal como la clasificación de texto.

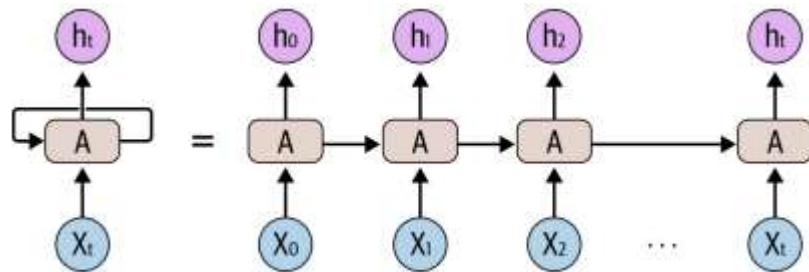


Figura 2.6: Célula recurrente.

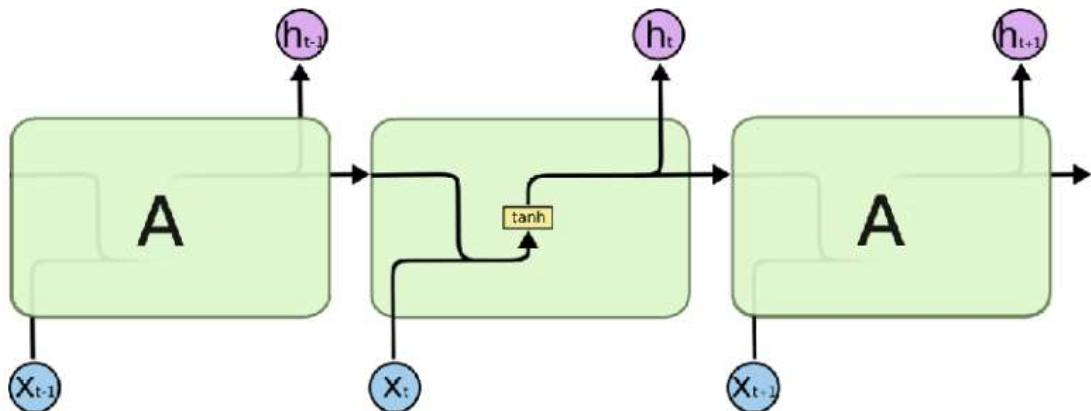


Figura 2.7: Arquitectura de una célula recurrente estándar.

2.2.2.2.1. RNN-LSTM

A pesar de su capacidad y versatilidad, las RNN sufren del problema de la memoria olvidadiza, es decir, no pueden recordar contextos amplios y por lo tanto no ofrecen muy buenos resultados cuando el texto de entrada es largo, lo cual suele ser bastante

normal. Las LSTM (*Long Short-Term Memory*), un tipo de RNN, fueron inventadas precisamente para mitigar esta desventaja de las RNN. Las LSTM evitan este problema dejando pasar al contexto irrelevante y recordando solo la parte del contexto que se necesita para resolver la tarea. Esto aligera la carga de recordar un contexto muy amplio en una representación vectorial.

Normalmente el entrenamiento de una red LSTM tarda bastante más que una CNN. Además, aunque las LSTMs son más potentes al utilizar la naturaleza secuencial del texto, son mucho más exigentes que las CNNs en cuanto a los datos necesarios. Por lo tanto, puede ocurrir que una red LSTM funcione peor que una red LSTM, debido a que la cantidad de datos de entrenamiento no sean suficientes para aprovechar el potencial de la LSTM.

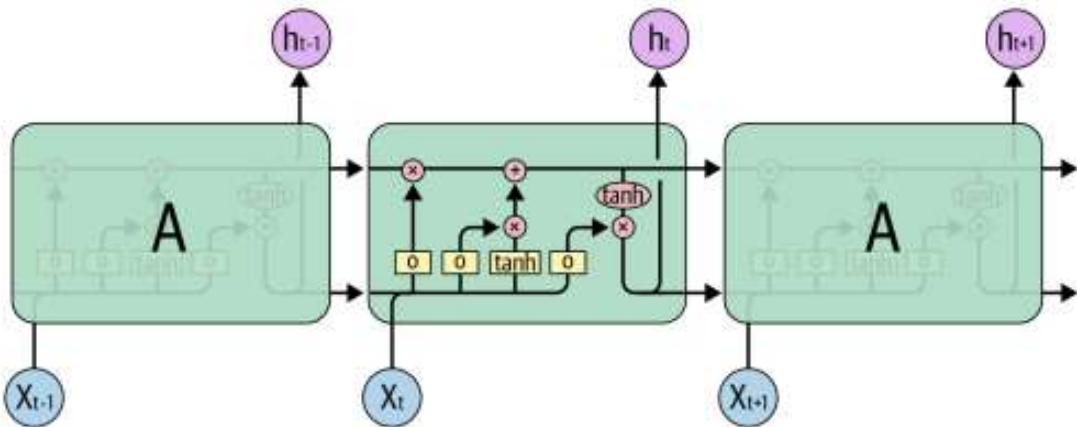


Figura 2.8: Arquitectura de una célula LSTM.

2.2.2.2.2. RNN-GRU

Las GRU (*Gated Recurrent Unit*) son otra variante de RNN que también se utilizan en temas de texto. Su arquitectura es parecida a las LSTMs, con una puerta de olvido (*forget gate*), pero tiene menos parámetros que la LSTM. Aún así, su rendimiento en algunas tareas de NLP es similar a la de las LSTMs.

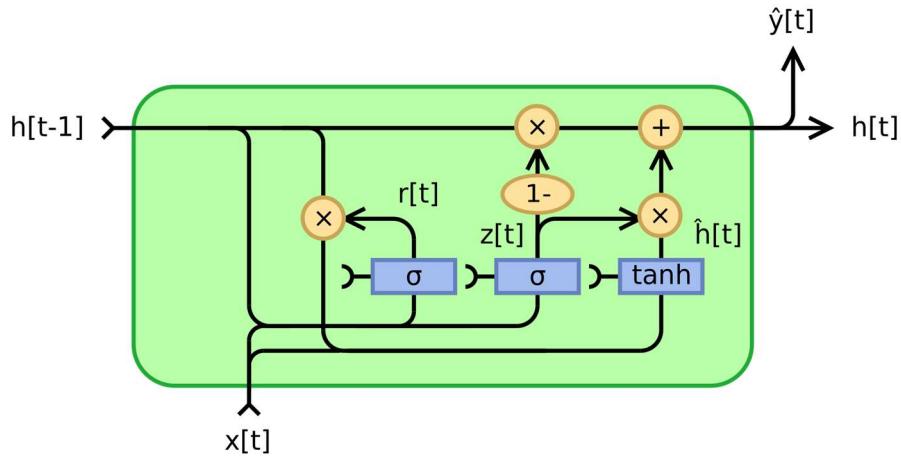


Figura 2.9: Arquitectura de una célula GRU.

2.2.2.2.3. Transformers

Los Transformers son la novedad en el campo de los modelos de Deep Learning para NLP. Últimamente están recibiendo mucha atención, mediática y del público en general, debido a los últimos avances en esta tecnología, como por ejemplo el ChatGPT, que recientemente ha causado un auténtico revuelo en la sociedad.

Los Transformers representan el *state-of-the-art* más novedoso en las principales tareas de NLP. Modelizan el contexto textual pero no de una manera secuencial, sino que, dada una palabra en la entrada, prefiere mirar a las palabras de su alrededor, proceso que se llama auto-atención (*self-attention*), y representan cada palabra con respecto a su contexto. Esto les confiere una mayor capacidad de representación que otras redes basadas en Deep Learning.

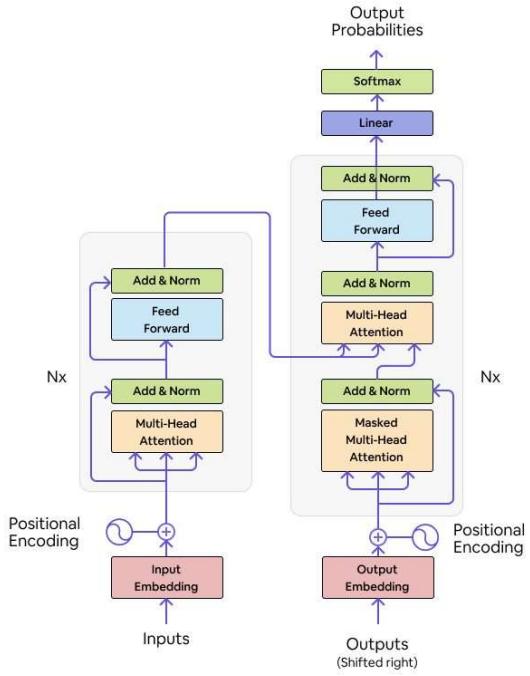


Figura 2.10: Arquitectura de un Transformer.

Recientemente, se han usado grandes Transformers para realizar *transfer learning*, con resultados muy positivos. El *transfer learning* es una técnica en la que el conocimiento obtenido al solucionar un problema se aplica a un problema distinto pero relacionado. La idea consiste en entrenar un Transformer muy grande de manera no-supervisada (concepto conocido como *pre-training*), para predecir una parte de una frase dado el resto de la frase, de manera que pueda codificar los matices del lenguaje en él. Estos modelos se entrena sobre grandes cantidades de datos textuales, como por ejemplo 40 GB, obtenidos de internet mediante *scrapping*. Un ejemplo paradigmático de un Transformer de este tipo es BERT (*Bidirectional Encoder Representations from Transformers*).

Capítulo 3

Análisis exploratorio y limpieza de los datos

3.1. Origen y características del dataset

El dataset que se utiliza en este TFM consiste en 5 ficheros en formato CSV (*Comma Separated Values*, Valores Separados por Comas), correspondientes a medidas de ecocardiografía recogidas en un hospital, mediante una solución tecnológica de Philips encuadrada en su sistema de gestión de ecocardiografía ISCV (IntelliSpace CardioVascular). Estas medidas corresponden a varios pacientes, a los cuales se les asigna un estudio o informe, de tal manera que un mismo paciente puede tener varios estudios y un estudio puede tener varios códigos de diagnóstico diferentes. Cada código de diagnóstico tiene un calificador de severidad. Un mismo código puede tener diferentes calificadores. También hay un campo *TextoCodigo* que, junto al calificador, permite obtener el campo *codigo*, que es un código alfanúmerico.

3.2. Análisis Exploratorio de los Datos

En este apartado realizaremos un análisis exploratorio de los datos de que disponemos. Lo primero que realizamos es cargar los 5 ficheros en un notebook de Jupyter y utilizar

diferentes comandos y funciones de Python para analizarlos. Los 5 ficheros de que disponemos son los siguientes:

- codigos.csv
- estudios.csv
- Pacientes.csv
- Medidas.csv
- Intervenciones.csv

De estos 5 ficheros, en un principio, solo utilizaremos los dos primeros, por lo que pasemos a analizar estos dos ficheros, para conocer su estructura, los diferentes campos de que constan y más información que nos puede ser útil en el desarrollo de TFM.

3.2.1. Fichero codigos.csv

Se trata de un fichero con un tamaño de 50.331 KB. Una vez cargado en un Jupyter Notebook, en un *dataframe* de pandas, podemos ver que se trata de un fichero con:

- 759.312 filas.
- 5 columnas o campos.

Mediante el método *sample()* podemos visualizar el aspecto de algunas filas del fichero:

```
codigos_df = pd.read_csv('ecocardio/procesados/codigos.csv', index_col=None, encoding='utf8', engine='python')
codigos_df.sample(5)
```

	ID	studyid	codigo	textoCodigo	calificador
429833	429834	136409	AV-0148	Estenosis aórtica con bajo flujo/bajo gradiente	NaN
346470	346471	128693	LA-0001	Tamaño normal de aurícula izquierda	NaN
186614	186615	114119	TV-0038	Válvula tricúspide normal en estructura y función.	NaN
389504	389505	132246	PE-0069	No signos de compresión de cavidades derechas.	NaN
625552	625553	151727	LV-0175	_ inferior (segmento basal)	Hipoquinesia

Figura 3.1: Muestra del fichero codigos.csv.

Mediante el método *info()* podemos obtener información sobre el fichero:

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 759312 entries, 0 to 759311
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   ID          759312 non-null   int64  
 1   studyid     759312 non-null   int64  
 2   codigo       759312 non-null   object  
 3   textoCodigo 758922 non-null   object  
 4   calificador 158915 non-null   object  
dtypes: int64(2), object(3)
memory usage: 29.0+ MB

```

Figura 3.2: Información del fichero codigos.csv.

Además, para cada columna calculamos el número de valores distintos que hay, mediante una función que utiliza el método *unique()*:

Unique values	
ID	759312
studyid	39103
codigo	1820
textoCodigo	1808
calificador	2650

Figura 3.3: Valores únicos de los campos en el fichero codigos.csv.

Por lo tanto, tenemos la siguiente información sobre las 5 columnas:

Nombre	Tipo	No-nulos	Valores únicos
ID	Integer	759.312	759.312
studyid	Integer	759.312	39.103
codigo	Object	759.312	1.820
textoCodigo	Object	758.922	1.808
calificador	Object	158.915	2.650

Tabla 3.1: Resumen del fichero codigos.csv.

Vemos que tenemos un campo *ID* de tipo entero, sin valores nulos y con un valor distinto para cada fila, tal como corresponde para un campo que es un identificador. El campo *studyid* también es de tipo entero, sin valores nulos y con 39.103 valores únicos, lo cual

ya nos indica que a un mismo *studyid* le pueden corresponder diferentes filas. Veamos un ejemplo, tomando al azar un *studyid* concreto y visualizando las diferentes filas que tiene este *studyid*:

```
codigos_df[codigos_df.studyid == 128693]
```

ID	studyid	codigo	textoCodigo	calificador
346465	346466	128693	AO-0047	La raíz aórtica es de tamaño normal.
346466	346467	128693	AV-0062	Válvula aórtica normal en estructura y función.
346467	346468	128693	CT-0001	Fase del estudio cardiotoxicidad _
346468	346469	128693	CT-0017	Estudio cardiotox: antraciclinas
346469	346470	128693	DX-0050	Linfoma
346470	346471	128693	LA-0001	Tamaño normal de aurícula izquierda
346471	346472	128693	LV-0065	La pared del ventrículo izquierdo muestra un grosor normal.
346472	346473	128693	LV-0152	El patrón de llenado del VI sugiere función diastólica normal
346473	346474	128693	LV-0187	Ventrículo izquierdo de tamaño y contractilidad normales.
346474	346475	128693	MV-0051	La válvula mitral es normal en estructura y función.
346475	346476	128693	PE-0015	No existe derrame pericárdico.
346476	346477	128693	PV-0028	Válvula pulmonar normal en estructura y función.
346477	346478	128693	RA-0017	El tamaño de la aurícula derecha es normal.
346478	346479	128693	RV-0029	El ventrículo derecho es normal respecto al tamaño y función.
346479	346480	128693	SP-0010	Oncología
346480	346481	128693	SU-0119ch	Ecocardiograma normal.
346481	346482	128693	TV-0038	Válvula tricúspide normal en estructura y función.
346482	346483	128693	TV-0219	No se ha podido estimar PSVD por ausencia de insuficiencia tricúspide.
346483	346484	128693	UB-0001	Ambulatorio
346484	346485	128693	VE-0068	No se observan signos de aumento de la PVC

Figura 3.4: Ejemplo de datos asociados a un estudio particular.

Podemos ver que las diferentes filas asociadas a este *studyid* de ejemplo tienen valores del campo *ID* consecutivos, y valores distintos para el resto de las columnas. Si seguimos analizando los diferentes campos, vemos que el campo *codigo* es de tipo *object*, que es la manera como pandas suele identificar el tipo *string*, no tiene valores nulos y comprende 1.820 valores únicos. El campo *textoCodigo* también es de tipo *string*, tiene unos pocos valores nulos (390, más concretamente) y comprende 1.808 valores únicos. Por último, tenemos el campo *calificador*, que es de tipo *string* y tiene muchos valores nulos (solo un 20,93% de los valores son no nulos).

3.2.2. Fichero estudios.csv

Se trata de un fichero con un tamaño de 19.926 KB. Una vez cargado en el Jupyter Notebook, en un *dataframe* de pandas, podemos ver que se trata de un fichero con:

- 137.874 filas.
- 5 columnas o campos.

Para este fichero realizaremos el mismo tipo de análisis que hemos realizado para el fichero *códigos.csv*:

```
estudios_df = pd.read_csv('ecocardio/procesados/estudios.csv', index_col=None, encoding='utf8', engine='python')
estudios_df.sample(5)
```

	studyid	fechaEstudio	perfilInforme	pacienteMRN	conclusiones
70808	90772	2018-09-10 11:05:25	Adult	12121227	NaN
101686	123655	2020-10-14 17:35:49	Pediatric	7193176	NaN
121349	144699	2021-12-21 11:47:37	Adult	4402894	Estudio de calidad subóptima por escasa colaboración de la paciente. Canal AV intervenido. No signos de cortocircuito residual (difícil de evaluar). Válvula mitral con hendidura reparada, con regurgitación mitral residual probablemente de grado moderado. Asocia aumento de velocidad en el llenado mitral compatible con estenosis de grado moderado. Ambos ventrículos de dimensiones y función normales. No se ha podido estimar PSVD por ausencia de insuficiencia tricúspide.
88442	109777	2019-10-15 08:44:41	Adult	1183902	Ventrícuo izquierdo de tamaño y contractilidad normales. Válvula aórtica trivalva con cambios degenerativos. Insuficiencia aórtica leve a moderada. Dilatación de la raíz aórtica. Dilatación de la aorta ascendente
53162	72313	2017-04-07 09:26:18	Adult	2978118	NaN

Figura 3.5: Muestra del fichero estudios.csv.

```
estudios_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 137874 entries, 0 to 137873
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   studyid     137874 non-null  int64  
 1   fechaEstudio 137874 non-null  object  
 2   perfilInforme 137874 non-null  object  
 3   pacienteMRN  137874 non-null  object  
 4   conclusiones  42368 non-null  object  
dtypes: int64(1), object(4)
memory usage: 5.3+ MB
```

Figura 3.6: Información del fichero estudio.csv.

Unique values	
studyid	137874
fechaEstudio	137719
perfilInforme	4
pacienteMRN	53244
conclusiones	37906

Figura 3.7: Valores únicos de los campos en el fichero estudios.csv.

Tenemos 5 columnas o campos, cuya información la resumimos en la tabla 3.2:

Nombre	Tipo	No-nulos	Valores únicos
studyid	Integer	137.874	137.874
fechaEstudio	Object	137.874	137.719
perfilInforme	Object	137.874	4
pacienteMRN	Object	137.874	53.244
conclusiones	Object	42.368	37.906

Tabla 3.2: Resumen del fichero estudios.csv.

Vemos que tenemos un campo *studyid* de tipo entero, sin valores nulos y con 137.874 valores únicos, y que sabemos que se corresponde con el mismo campo *studyid* del fichero *códigos.csv* por lo que, por lo tanto, nos puede servir para realizar un *join* entre ambos ficheros.

3.2.3. Dataframe cardio_df

Si tenemos en cuenta que el objetivo de este TFM es intentar predecir los códigos de hallazgo de cada estudio (columna *código* del fichero *códigos.csv*) a partir del campo de texto de conclusiones de dicho estudio (columna *conclusiones* del fichero *estudios.csv*), está claro que tenemos que relacionar ambos ficheros. Hemos visto que ambos ficheros tienen un campo *studyid*, que nos permite relacionarlos. Así pues, lo que podemos hacer es crear un *dataframe* que llamaremos *cardio_df* que se puede crear a partir de un *merge* de tipo *left join* entre los *dataframes* que tenemos a partir de los ficheros

codigos.csv y *estudios.csv*. Esto es lo que hacemos, y si visualizamos una muestra del *dataframe* resultante tenemos:

```
cardio_df = codigos_df.merge(estudios_df, on='studyid', how='left')
cardio_df.sample(5)
```

	ID	studyid	codigo	textoCodigo	calificador	fechaEstudio	perfilInforme	pacienteMRN	conclusiones
721578	721579	158916	CV-0052	Mieotomia septal (MCHO): fecha (mm/aa): _	10/22	2022-10-17 13:34:50	Adult	271056	Miocardiopatía hipertrófica con mieotomía setpal. Buena función sistólica con gradiente dinámico residual de 35 mmHg en relación a hipertrofia de músculo papilar. Insuficiencia mitral ligera-moderada con mínimo SAM mitral. Ventrículo derecho de pequeño tamaño con función conservada. Ausencia de derrame pericárdico.
251436	251437	120055	VE-0068	No se observan signos de aumento de la PVC	NaN	2020-07-08 13:24:03	Adult	3151684	Regular ventana acústica. Estenosis aórtica valvular probablemente moderada, calcificada. Insuficiencia aórtica leve. Válvula mitral calcificada a nivel anular y velos, con apertura limitada y estenosis que parece leve. Ventrículo izquierdo de tamaño y contractilidad normales. No se observan alteraciones segmentarias de la contractilidad del VI. El patrón de llenado sugiere disfunción diastólica grado II.
174612	174613	113001	MV-0277	Plastia mitral no obstructiva y sin signos de regurgitación residual	NaN	2019-12-17 13:33:27	Adult	1188007	Estudio sin cambios relevantes respecto al previo VI de tamaño normal, con función sistólica severamente reducida. Prótesis aórtica biológica normofuncionante. Plastia mitral no obstructiva y sin signos de regurgitación residual. El ventrículo derecho es normal respecto al tamaño y función. No se ha podido estimar PSVD por ausencia de insuficiencia tricúspide.
263305	263306	121442	AV-0124	Válvula aórtica trivalva con cambios degenerativos	NaN	2020-08-19 09:37:08	Adult	3008840	Ventrículo izquierdo de tamaño y contractilidad normales. Insuficiencia aórtica leve a moderada. Ventrículo derecho ligeramente dilatado con función sistólica en límite inferior de la normalidad con contractilidad longitudinal alterada. Conducto protésico en aorta ascendente (supracoronario) Derrame pericárdico ligero. No signos de compromiso hemodinámico
239372	239373	118886	RV-0032c	El ventrículo derecho está dilatado. levemente	levemente	2020-06-12 10:32:08	Adult	5441171	CIA tipo ostium secundum intervenida (cierra quirúrgico). No signos de cortocircuito residual. Ventrículo izquierdo de tamaño y contractilidad normales. Ligera hipertrabeculación del VI a nivel de ápex. Ventrículo derecho levemente dilatado, con función sistólica normal. Ausencia de valvulopatías. Presión sistólica ventricular derecha estimada normal.

Figura 3.8: Muestra del *dataframe* cardio_df.

Mediante el método *info()* podemos visualizar información sobre este *dataframe*:

```
cardio_df.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 759312 entries, 0 to 759311
Data columns (total 9 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   ID          759312 non-null   int64  
 1   studyid     759312 non-null   int64  
 2   codigo       759312 non-null   object  
 3   textoCodigo 758922 non-null   object  
 4   calificador 158915 non-null   object  
 5   fechaEstudio 759312 non-null   object  
 6   perfilInforme 759312 non-null   object  
 7   pacienteMRN 759312 non-null   object  
 8   conclusiones 754850 non-null   object  
dtypes: int64(2), object(7)
memory usage: 57.9+ MB
```

Figura 3.9: Información del *dataframe* cardio_df.

Vemos que tenemos algunos casos en que hay valores nulos en el campo conclusiones y, como estos casos no nos sirven, ya que queremos predecir a partir de este campo, el siguiente paso es eliminar estos casos:

```

cardio_df = cardio_df.loc[cardio_df['conclusiones'].notnull()]
cardio_df.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 754850 entries, 11 to 759311
Data columns (total 9 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   ID          754850 non-null  int64  
 1   studyid     754850 non-null  int64  
 2   codigo       754850 non-null  object  
 3   textoCodigo 754506 non-null  object  
 4   calificador 157180 non-null  object  
 5   fechaEstudio 754850 non-null  object  
 6   perfilInforme 754850 non-null  object  
 7   pacienteMRN  754850 non-null  object  
 8   conclusiones 754850 non-null  object  
dtypes: int64(2), object(7)
memory usage: 57.6+ MB

```

Figura 3.10: Información del *dataframe* *cardio_df* una vez se han eliminado las filas con el campo *conclusiones* nulo.

Veamos ahora cómo se distribuyen los valores del campo *codigo*:

```

counts = cardio_df.value_counts('codigo', ascending=False)
counts

codigo
PE-0015      28721
AO-0047      26715
RV-0029      25063
UB-0001      22092
LV-0187      21568
...
ST-0004a3      1
ST-0004a91     1
ST-0007-5      1
ST-0009      1
CT-0040      1
Length: 1781, dtype: int64

```

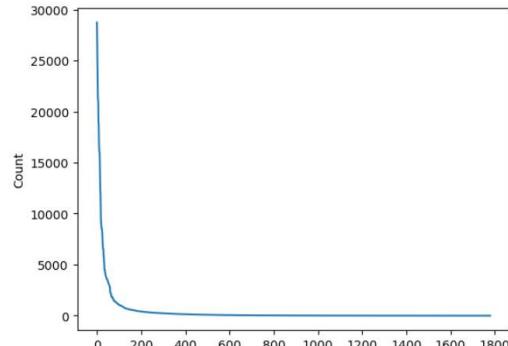


Figura 3.11: Distribución de los valores del campo *codigo*.

Vemos que la distribución es muy desbalanceada, con casos en los que tenemos cerca de 30.000 muestras y otros en los que solo tenemos una. También podemos representar la suma acumulada de los valores:

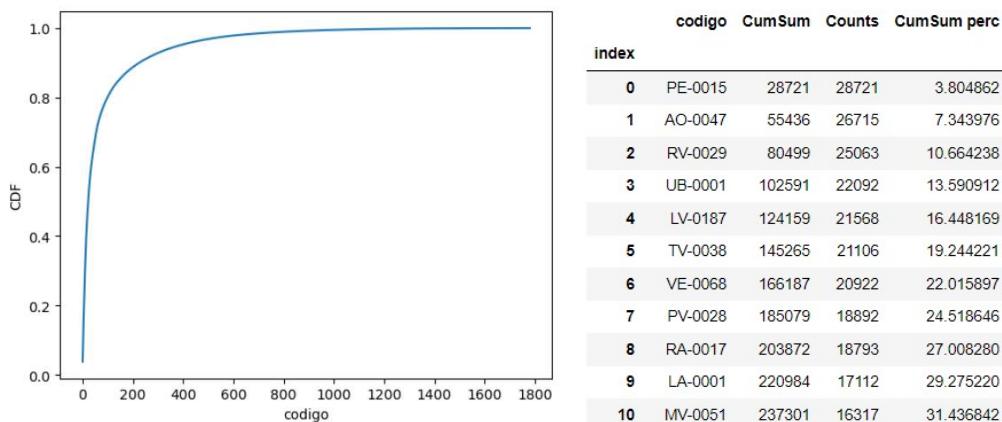


Figura 3.12: Distribución de la suma acumulada de los valores del campo *codigo*.

Podemos ver que un pequeño porcentaje del total de códigos representa un porcentaje bastante alto de todos los casos que tenemos. Por ejemplo:

cumsum_df.iloc[50]	cumsum_df.iloc[100]	cumsum_df.iloc[200]	cumsum_df.iloc[300]
codigo A0-0079	codigo MS-0003	codigo SP-0017	codigo AT-0029-2
CumSum 515719	CumSum 605364	CumSum 669888	CumSum 700796
Counts 3350	Counts 1063	Counts 403	Counts 240
CumSum perc 68.320726	CumSum perc 80.196595	CumSum perc 88.744519	CumSum perc 92.839107
Name: 50, dtype: object	Name: 100, dtype: object	Name: 200, dtype: object	Name: 300, dtype: object

Figura 3.13: Ejemplos de la suma acumulada de valores del campo *codigo*.

Vemos que los 50 códigos más representativos conforman el 68,32% de los casos, y que los 100 códigos más representativos conforman el 80,20% de los casos.

Pasemos a analizar el campo *conclusiones*, que recordemos que es el campo de texto a partir del cual queremos predecir el campo *codigo*. Si calculamos su longitud en caracteres, tenemos la siguiente estadística:

cardio_df['conclusion_len'] = cardio_df.conclusiones.str.len()
cardio_df.conclusion_len.describe()
count 754850.000000
mean 303.015998
std 168.789485
min 14.000000
25% 187.000000
50% 284.000000
75% 400.000000
max 1797.000000
Name: conclusion_len, dtype: float64

Figura 3.14: Estadísticas del campo *conclusiones*.

Por lo tanto, vemos que la longitud en caracteres del campo *conclusiones* tiene un valor mínimo de 14, un valor máximo de 1.797, una media de 303,0 y una desviación estándar de 168,8. Vemos, pues, que la longitud es muy dispar.

3.3. Limpieza de texto

Veamos algún texto de ejemplo del campo *conclusiones*. Por ejemplo, tomemos la muestra 500:

```
print(cardio_df.conclusiones[500])
```

Valvulopatía mitral reumática con doble lesión mitral ambas de grado moderado, área valvular estimada por tiempo de hemipresión en 1'4 cm² y por planimetría en 1'6 cm², insuficiencia mitral moderada con un ORE por método de PISA de 0'29 cm². Válvula aórtica con calcificación de uno de sus velos, aceptable apertura sistólica, estenosis aórtica ligera con gradiente máximo instantáneo de 34 y gradiente medio de 20 mmHg. Ligera dilatación de aorta ascendente. Ventrículo izquierdo de tamaño en límites altos sin hipertrofia de sus paredes, con fracción de eyección del 45%. Ventrículo derecho de tamaño y contractilidad normales, con insuficiencia tricúspide ligera funcional que permite estimar una presión sistólica de arteria pulmonar de 42 mmHg. Ausencia de derrame pericárdico. Ausencia de signos de aumento de la presión venosa central. Fdo.: Dr. Miró

Figura 3.15: Ejemplo del campo *conclusiones*.

Podemos ver que se trata de texto libre, en castellano, y que contiene valores numéricos correspondientes a valores de medidas de diferentes factores.

Vamos a realizar un pre-procesado del texto del campo *conclusiones* antes de extraer las características. Este pre-procesado consistirá en:

- Eliminar los signos de puntuación.
- Eliminar las palabras menores de 3 caracteres.
- Eliminar las palabras que estén en una lista de *stopwords*. Utilizo el conjunto de *stopwords* en castellano según NLTK.
- Eliminar los valores numéricos.
- Pasar todas las palabras a minúsculas.
- Lematizar el texto. El resultado de lematizar lo guardaremos en otra columna, para así poder comparar resultados del clasificador lematizando y sin lematizar.

Todo esto lo realizamos a partir de una función en Python, con un parámetro para elegir si queremos lematizar o no. A partir de esta función, se crean dos columnas: la columna *limpido*, con el texto pre-procesado pero sin lematizar, y la columna *lemas*, con el texto pre-procesado y lematizado. Si tomamos el mismo ejemplo que antes, tenemos que el texto limpiado y el texto lematizado son así:

Limpido:

valvulopatía mitral reumática doble lesión mitral ambos grado moderado área valvular estimada tiempo hemipresión planimetría insuficiencia mitral moderada método pisa válvula aórtica calcificación velos aceptable apertura sistólica estenosis aórtica ligera gradiente máximo instantáneo gradiente medio mmhg ligera dilatación aorta ascendente ventrículo izquierdo tamaño límite es altos hipertrofia paredes fracción eyección ventrículo derecho tamaño contractilidad normales insuficiencia tricúspide ligera funcional permite estimar presión sistólica arteria pulmonar mmhg ausencia derrame pericárdico ausencia signos aumento presión venosa central miró

Lematizado:

valvulopatía mitral reumático doble lesión mitral ambos grado moderado área valvular estimado tiempo hemipresión planimetría insuficiencia mitral moderada método pisa válvula aórtico calcificación velo aceptable apertura sistólica estenosis aórtico ligeramente gradiente máximo instantáneo gradiente medio mmhg ligero dilatación aorta ascendente ventrículo izquierdo tamaño límite alto hipertrofia pared fracción eyección ventrículo derecho tamaño contractilidad normal insuficiencia tricúspide ligero funcional permitir estimar presión sistólica arteria pulmonar mmhg ausencia derrame pericárdico ausencia signo aumento presión ve nos central miró

Figura 3.16: Ejemplo texto una vez limpiado y lematizado.

Por último, ahora calculamos la longitud en palabras según el texto limpiado, y obtenemos esta estadística y distribución:

```
count    754850.000000
mean     27.180038
std      14.842810
min      1.000000
25%     17.000000
50%     26.000000
75%     36.000000
max     156.000000
Name: words, dtype: float64
```

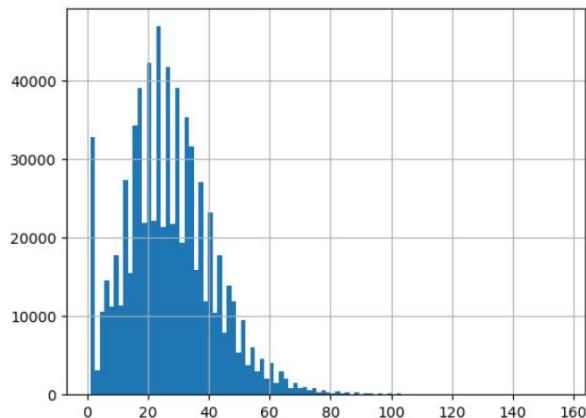


Figura 3.17: Estadísticas y distribución del campo *limpio*.

Para poder trabajar más cómodamente a lo largo del TFM, procedemos a guardar en disco el *dataframe* final, en un fichero CSV. Así, cada modelo que consideremos lo podremos entrenar en un notebook diferente, y así ganar en eficiencia y claridad.

Capítulo 4

Clasificación

4.1. Introducción

La estrategia que se seguirá en el desarrollo de este TFM será considerar diferentes modelos, del más sencillo al más complicado, comprobar si realmente podemos predecir el campo *codigo* a partir del campo *conclusiones* y, en caso afirmativo, ver si mejoramos la predicción al aumentar la complejidad del modelo.

4.2. Modelos basados en BoW y TF-IDF

Lo primero que haremos será aplicar modelos simples basados en BoW y TF-IDF. La idea preliminar es que el texto clínico suele ser muy esquemático, por lo que suele funcionar bastante bien con estos modelos. Además, el hecho de que sea esquemático provoca que sea más difícil de interpretar por modelos semánticos.

Los pasos que vamos a seguir en el desarrollo de este apartado son los siguientes:

- Dividir el dataset en los conjuntos de entrenamiento y test:
 - Como *features* tomamos el campo *limpio* y como *target* tomamos el campo *codigo*.
 - Elegimos un tamaño de test típico, del 30%.
- Extracción de características BoW y TF-IDF:
 - Para BoW, utilizamos la clase CountVectorizer de scikit-learn.

- Para TF-IDF, utilizamos la clase TfidfVectorizer.
- Probamos dos clasificadores típicos de la librería scikit-learn, como son Naïve Bayes y un SVM lineal.
- Como métrica, elegimos el *accuracy*.

Con todo esto, entrenamos las diferentes combinaciones de modelos que podemos tener con las anteriores opciones. Los resultados de *accuracy* que se obtienen se resumen en la siguiente tabla:

Enfoque	MultinomialNB		SVM lineal	
	Sin lematizar	Lematizado	Sin lematizar	Lematizado
BOW	3,51%	3,57%	2,19%	2,78%
TF-IDF	3,33%	3,73%	2,17%	2,24%

Tabla 4.1: *Accuracies* con los datos originales.

Los resultados son muy bajos. Si consideramos como *benchmark* un modelo muy naif, que asigne siempre como resultado del clasificador la clase más representada, tendríamos que el *accuracy* esperado sería de 3,80%, y vemos que en ningún caso llegamos siquiera a este valor. Por lo tanto, tras realizar algunas pruebas y consultarlos con el tutor, llegamos a la siguiente conclusión. En los datos que tenemos, un informe, que corresponde a un mismo paciente, puede tener varias líneas, que se corresponden a diferentes códigos. Cada una de estas filas tiene un campo *codigo* diferente, pero el campo *conclusiones* es el mismo en todos los casos. Así pues, tenemos que a partir de un mismo campo de conclusiones esperamos poder predecir diferentes códigos. Esto me parece muy anormal y considero que es complicado que, a partir de un campo que no cambia en todo el informe, podamos predecir un campo, *codigo*, que sí que cambia. Sabiendo esto, ya no encuentro tan extraño que se hayan obtenido resultados tan pobres, ya que intuyó que el campo *conclusiones*, por sí solo, no tiene suficiente poder predictivo.

Así pues, es necesario redefinir la estrategia a seguir, para poder utilizar este *dataset* para el objetivo marcado. Tras realizar más pruebas con el *dataset*, hay una estrategia

que resulta muy prometedora, y que consiste en que, para cada informe, considerar solamente el primer código, el que tiene el ID más bajo, tomando como hipótesis de trabajo que los códigos tienen un ID que siguen un orden cronológico, siendo el código con el ID más bajo el primer código consignado, y también que el primer código consignado es el más importante.

Tomando estas hipótesis en consideración, definimos un nuevo *dataset* a partir del *dataframe* que habíamos creado, de tal manera que ahora para cada informe tenemos solo la fila referida al primer código consignado. La información de los campos importantes es:

<code>cardio_df[['codigo', 'conclusiones', 'limpio', 'lemas']].info()</code>	<code>len(cardio_df['codigo'].unique())</code>
<class 'pandas.core.frame.DataFrame'> Int64Index: 37774 entries, 0 to 754818 Data columns (total 4 columns): # Column Non-Null Count Dtype --- -- -- -- -- 0 codigo 37774 non-null object 1 conclusiones 37774 non-null object 2 limpio 37774 non-null object 3 lemas 37774 non-null object dtypes: object(4) memory usage: 1.4+ MB	428
	<code>len(cardio_df['limpio'].unique())</code>
	32713
	<code>len(cardio_df['lemas'].unique())</code>
	32688

Figura 4.1: Información de los datos modificados.

Vemos que ahora tenemos 37.774 muestras, en las cuales tenemos que el campo *codigo* tiene 428 valores únicos, el campo *limpio* tiene 32.713 y el campo *lemas* tiene 32.688.

Miremos ahora cómo se distribuyen estos 428 códigos. Si calculamos la suma acumulada y su porcentaje sobre el total, podemos ver que los 50 códigos más importantes representan el 96,1% del total de los casos. Así pues, otra aproximación que podemos tomar es considerar solo los 50 códigos más representados y eliminar el resto. Con esta aproximación, podemos volver a nuestra estrategia inicial, pero considerando este *dataset* más reducido y con muchas menos clases a predecir. Además, en vista de que antes se obtenían peores resultados con el modelo SVM lineal, ahora prescindimos de este modelo y probamos otros dos modelos de la misma librería `scikit_learn`: *SGDClassifier* (que, igualmente, es un SVM lineal con SGD) y *RandomForestClassifier*. Los resultados de *accuracy* que se obtienen ahora son:

Enfoque	MultinomialNB		SGDClassifier		RandomForestClassifier	
	Sin lematizar	Lematizado	Sin lematizar	Lematizado	Sin lematizar	Lematizado
BOW	76,68%	76,81%	82,98%	82,85%	72,77%	72,77%
TF-IDF	73,72%	73,63%	81,10%	81,21%	72,77%	72,77%
2-grams	77,19%	77,01%	84,88%	84,64%	72,77%	72,77%
3-grams	78,71%	78,77%	84,64%	84,61%	72,77%	72,77%
4-grams	79,31%	79,37%	84,70%	84,47%	72,89%	73,02%
5-grams	79,52%	79,52%	84,67%	84,58%	73,04%	73,02%

Tabla 4.2: *Accuracies* con los datos modificados.

Podemos ver que ahora los resultados obtenidos son mucho mejores, sobre todo con el modelo *SGDClassifier*.

4.3. Modelos basados en CNN

Pasaremos ahora a implementar diversos modelos de clasificador basados en *Convolutional Neural Networks* usando la librería Keras. Aplicaremos una primera capa de *embeddings* para convertir las palabras en vectores y luego entrenaremos con una red CNN.

Para calcular los *embeddings* usamos tres metodologías diferentes, para así poder comparar los resultados obtenidos:

- Una capa de *embeddings* propia.
- *Transfer Learning* con los *word embeddings* GloVe de spaCy.
- *Transfer Learning* con los *word embeddings* de FastText.

El proceso seguido en este apartado contempla los siguientes puntos a considerar:

- En este caso utilizaremos solamente el campo *lemas*, y desechamos el campo *limpio*.
- Para poder utilizar CNN, se deben añadir columnas adicionales, una para cada posible código, mediante *One-Hot Encoding*.
- Convertimos el texto en *tokens* y asignamos una ID numérica a cada token.
- Convertimos a secuencias de longitud fija.

- La longitud de la secuencia viene dada por la longitud en tokens del texto más largo. Sólo se conservan los tokens de las palabras en el vocabulario.
- Dividimos el *dataset* en los conjuntos de entrenamiento y test:
 - Como *features* tomamos el campo *lemas* y como *target* tomamos las columnas generadas con el *One-Hot Encoding*.
 - Elegimos un tamaño de test típico, del 30%.

Teniendo en cuenta todo esto, obtenemos:

- 6.186 tokens distintos en X_train.
- Una vez tokenizado, la longitud de cada *embedding* es de 135 tokens. Los tamaños de los conjuntos de entrenamiento y test son:

```
print(X_train.shape,y_train.shape)
print(X_test.shape,y_test.shape)
```

(25409, 135) (25409, 50)
(10890, 135) (10890, 50)

Figura 4.2: Tamaño de los conjuntos de entrenamiento y test.

4.3.1. *Word embeddings* propios

Definimos un modelo en Keras, en el cual entrenamos una capa de *embedding* para aprender los *word embeddings* con los textos de nuestro problema. Empezamos con el modelo de la figura 4.3.

Este modelo presenta *overfitting* muy pronto, ya en las primeras épocas. Lo que hago, entonces, es ir probando diferentes cambios, como eliminar capas, aumentar el *dropout*, cambiar el tamaño de *batch* o reducir el número de filtros, hasta que llego al siguiente modelo, que presenta muy poco *overfitting*. Los cambios finales respecto al primer intento son:

- Pasar de 64 a 16 filtros.
- Eliminar las dos capas densas centrales.
- Aumentar el *dropout*, de 0,2 a 0,5.

```

# Parámetros de la red
embed_dim = 50
filters = 64
kernel_size = 3

# Modelo
model = Sequential()
model.add(Embedding(max_features, embed_dim, input_length = MAX_SEQUENCE_LENGTH))
model.add(Dropout(0.2))

# Añadimos una capa de convolución 1D que aprende filtros de grupos de palabras de tamaño kernel_size
model.add(Conv1D(filters,
                 kernel_size,
                 padding='valid',
                 activation='relu',
                 strides=1))

# Calculamos el max pooling:
model.add(GlobalMaxPooling1D())
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(50, activation='softmax'))

# Compilamos el modelo
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

```

Figura 4.3: CNN: Modelo inicial para el caso con *word embeddings* propios.

```

# Parámetros de la red
embed_dim = 50
filters = 16
kernel_size = 3

# Modelo
model = Sequential()
model.add(Embedding(max_features, embed_dim, input_length = MAX_SEQUENCE_LENGTH))
model.add(Dropout(0.5))

# Añadimos una capa de convolución 1D que aprende filtros de grupos de palabras de tamaño kernel_size
model.add(Conv1D(filters,
                 kernel_size,
                 padding='valid',
                 activation='relu',
                 strides=1))

# Calculamos el max pooling:
model.add(GlobalMaxPooling1D())
model.add(Dense(50, activation='softmax'))

# Compilamos el modelo
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

```

Figura 4.4: CNN: Modelo final para el caso con *word embeddings* propios.

El sumario de este modelo es:

Model: "sequential"		
Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 135, 50)	309350
dropout (Dropout)	(None, 135, 50)	0
conv1d (Conv1D)	(None, 133, 16)	2416
global_max_pooling1d (GlobalMaxPooling1D)	(None, 16)	0
dense (Dense)	(None, 50)	850

=====
Total params: 312616 (1.19 MB)
Trainable params: 312616 (1.19 MB)
Non-trainable params: 0 (0.00 Byte)
..

Figura 4.5: CNN: Sumario del modelo para el caso con *word embeddings* propios.

Para entrenar, elijo un tamaño de *batch* de 16 y 20 épocas. Utilizo el método de *checkpoint* para ir guardando la época que ofrece mejor resultado sobre el conjunto de validación. De esta manera, obtenemos:

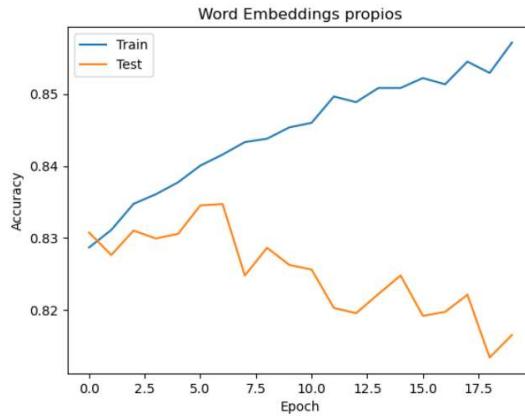


Figura 4.6: CNN: Accuracy para el caso con word embeddings propios.

Con los siguientes valores:

- Mejor época: 6
- Accuracy: 83,47%

Los *embeddings* tienen un tamaño de 6.187 x 50:

```
embeddings.shape
(6187, 50)
```

Figura 4.7: CNN: Tamaño de los *embeddings* para el caso con word embeddings propios.

Un ejemplo de *embedding* obtenido sería:

```
embeddings[54]
array([-0.01213438,  0.55471176, -0.2060268 ,  0.09459436, -0.07327766,
       0.0275359 ,  0.19869956, -0.01751716, -0.04459254,  0.15741397,
      -0.20487776,  0.04612475,  0.07956894,  0.06832976, -0.07384454,
       0.07410106,  0.07179404,  0.00471124,  0.08995294, -0.2649023 ,
      0.20086232, -0.02882028,  0.10799477,  0.04940287, -0.2060443 ,
     -0.02738357,  0.08572834,  0.2431352 , -0.14234096,  0.02561305,
       0.15392266, -0.2659551 , -0.1470608 ,  0.3098948 ,  0.08229178,
       0.09001922,  0.06969604, -0.10253214,  0.01248016,  0.04619435,
      -0.2585984 , -0.34124154, -0.1518887 ,  0.06301978,  0.40985715,
     -0.20383164, -0.01202726, -0.14768313,  0.0601731 ,  0.06762961],
      dtype=float32)
```

Figura 4.8: CNN: Ejemplo de emmbdding para el caso con word embeddings propios.

4.3.2. Word embeddings de spaCy

Definimos un modelo en Keras, en el cual entrenamos una capa de *embedding* para aprender los *word embeddings* aplicando *transfer learning* usando los *embeddings* de GloVe incluidos en el modelo de spaCy. La estrategia es similar a lo que hemos hecho en el apartado anterior, de ir realizando cambios sobre un modelo inicial hasta que vemos que reducimos el overfitting. Esto será así también para los posteriores modelos. Seguimos también con la utilización del *checkpoint* para elegir la época que ofrece un mejor *accuracy* sobre el conjunto de validación. Para entrenar, elijo un tamaño de *batch* de 16 y 20 épocas. El modelo final que obtenemos es:

```

filters = 64
kernel_size = 3

embedding_layer = Embedding(max_features,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=True)

model = Sequential()
model.add(embedding_layer)
model.add(Dropout(0.2))
model.add(Conv1D(filters,
                 kernel_size,
                 padding='valid',
                 activation='relu',
                 strides=1))
model.add(GlobalMaxPooling1D())
model.add(Dense(50, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
print(model.summary())

```

Figura 4.9: CNN: Modelo final para el caso con *word embeddings* de spaCy.

Model: "sequential_2"		
Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, 135, 300)	1856100
dropout_2 (Dropout)	(None, 135, 300)	0
conv1d_2 (Conv1D)	(None, 133, 64)	57664
global_max_pooling1d_2 (GlobalMaxPooling1D)	(None, 64)	0
dense_2 (Dense)	(None, 50)	3250

Total params:	1917014	(7.31 MB)
Trainable params:	1917014	(7.31 MB)
Non-trainable params:	0	(0.00 Byte)

Figura 4.10: CNN: Sumario del modelo para el caso con *word embeddings* de spaCy.

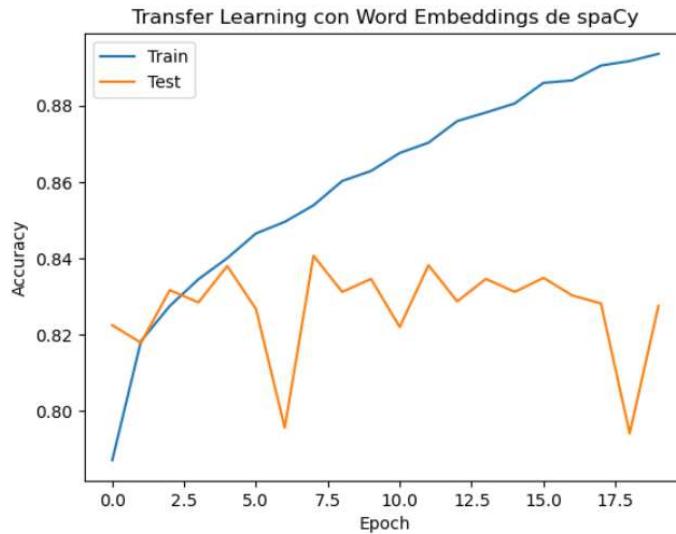


Figura 4.11: CNN: Accuracy para el caso con word embeddings de spaCy.

Con los siguientes valores:

- Mejor época: 7
- Accuracy: 84,07%

4.3.3. Word embeddings de FastText

En este caso utilizamos un conjunto de *word embeddings* de fastText, concretamente el conjunto contenido en el fichero *cc.es.300.vec*.

Los *embeddings* tienen un tamaño de 6.187 x 300:

```
embedding_matrix.shape
(6187, 300)
```

```

filters = 16
kernel_size = 3

embedding_layer = Embedding(max_features,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=True)

model = Sequential()
model.add(embedding_layer)
model.add(Dropout(0.5))
model.add(Conv1D(filters,
                 kernel_size,
                 padding='valid',
                 activation='relu',
                 strides=1))
model.add(GlobalMaxPooling1D())
model.add(Dense(50, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
print(model.summary())

```

Figura 4.12: CNN: Modelo final para el caso con *word embeddings* de FastText.

```

Model: "sequential_6"
=====
Layer (type)          Output Shape       Param #
=====
embedding_6 (Embedding)    (None, 135, 300)     1856100
dropout_6 (Dropout)        (None, 135, 300)      0
conv1d_6 (Conv1D)         (None, 133, 16)      14416
global_max_pooling1d_6 (GlobalMaxPooling1D) (None, 16)      0
dense_6 (Dense)           (None, 50)          850
=====
Total params: 1871366 (7.14 MB)
Trainable params: 1871366 (7.14 MB)
Non-trainable params: 0 (0.00 Byte)
=====
```

Figura 4.13: CNN: Sumario del modelo para el caso con *word embeddings* de FastText.

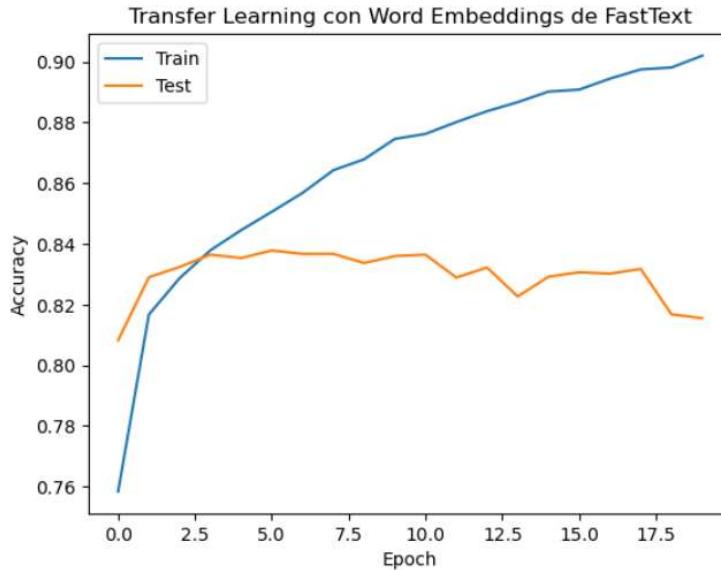


Figura 4.14: CNN: Accuracy para el caso con *word embeddings* de FastText.

Con los siguientes valores:

- Mejor época: 5
- *Accuracy*: 83,78%

4.3.4. Resultados finales con CNN

Si resumimos los resultados obtenidos con CNN y con los 3 métodos considerados, obtenemos la siguiente tabla:

	Propio	spaCy	fastText
CNN	83,47%	84,07%	83,78%

Tabla 4.3: *Accuracies* obtenidos con los 3 modelos basados en CNN.

Los resultados obtenidos por los 3 modelos son bastante parecidos, con valores de *accuracy* alrededor del 84%, siendo el mejor el modelo con Word embeddings de spaCy.

4.4. Modelos basados en RNN

Pasaremos ahora a implementar diversos modelos de clasificador basados en Recurrent Neural Networks (RNN) usando la librería Keras. Igual que en el caso de CNN, añadiremos una primera capa de *embeddings* para convertir las palabras en vectores y luego entrenaremos con una red RNN.

Para calcular los *embeddings* usamos tres metodologías diferentes, para así poder comparar los resultados obtenidos:

- Una capa de *embeddings* propia.
- *Transfer Learning* con los word embeddings de spaCy (modelo GloVe).
- *Transfer Learning* con los word embeddings de spaCy, actualizando los pesos en el entrenamiento.

Esto lo haremos utilizando dos tipos de RNN diferentes:

- LSTM (*Long Short-Term Memory*).
- GRU (*Gated Recurrent Unit*).

El proceso seguido en este apartado contempla los mismos puntos considerados en el caso CNN.

4.4.1. RNN-LSTM

4.4.1.1. *Word embeddings* propios

Definimos un modelo en Keras, en el cual entrenamos una capa de *embedding* para aprender los *word embeddings* con los textos de nuestro problema, sin entrenamiento previo. El modelo final es:

```
model = Sequential()
model.add(Embedding(MAX_NB_WORDS, EMBEDDING_DIM, input_length = MAX_SEQUENCE_LENGTH, mask_zero=True))
model.add(SpatialDropout1D(0.5))
model.add(RNN_layer(RNN_DIM, dropout=0.5, recurrent_dropout=0.5))
model.add(Dense(50, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print(model.summary())
```

Figura 4.15: RNN-LSTM: Modelo final para el caso con *word embeddings* propios.

Donde:

- MAX_NB_WORDS: Número máximo de *features* + 2. El motivo de sumar 2 es porque hay que considerar un término para OOV (*Out Of Vocabulary*) y otro para el 0. En nuestro caso, el número máximo de *features* es 3.440, por lo que MAX_NB_WORDS es 3.442.
- EMBEDDING_DIM: Es la dimensión del *embedding*. En nuestro caso es 300.
- MAX_SEQUENCE_LENGTH: La longitud de la secuencia viene dada por la longitud en tokens del texto más largo. En nuestro caso es 135.
- RNN_DIM: Número de neuronas en la capa RNN. En nuestro caso, elegimos que sea 50.

```

Model: "sequential_2"
-----  

Layer (type)          Output Shape       Param #
embedding_2 (Embedding)    (None, 135, 300)     1032600
spatial_dropout1d_2 (SpatialDropout1D) (None, 135, 300)      0
lstm_2 (LSTM)           (None, 50)          70200
dense_2 (Dense)          (None, 50)          2550
-----  

Total params: 1105350 (4.22 MB)
Trainable params: 1105350 (4.22 MB)
Non-trainable params: 0 (0.00 Byte)

```

Figura 4.16: RNN-LSTM: Sumario del modelo para el caso con *word embeddings* propios.

Para entrenar, elegimos un tamaño de batch de 16 50 épocas.

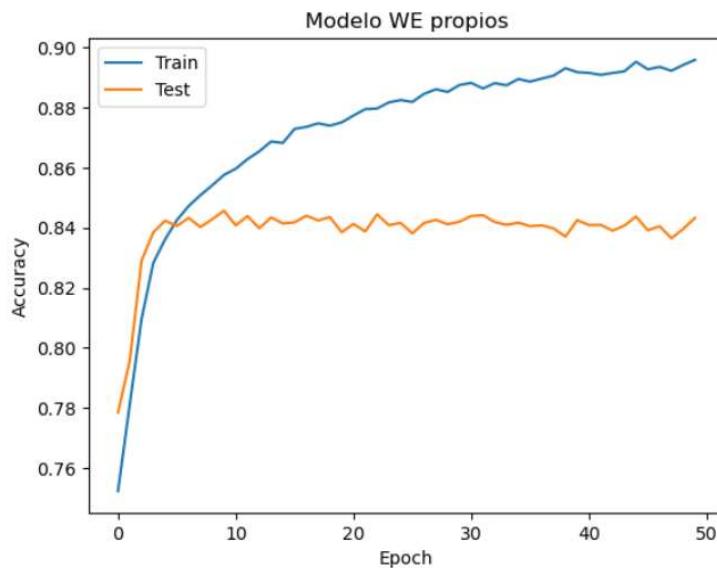


Figura 4.17: RNN-LSTM: *Accuracy* para el caso con *word embeddings* propios.

Tenemos:

- Mejor época: 9
- Accuracy: 84,56%

4.4.1.2. *Transfer learning* con *word embeddings* de GloVe

Definimos un modelo en Keras, en el cual usaremos los *word embeddings* GloVe ya calculados en la librería spaCy. Vamos a entrenar el modelo secuencial con los textos, donde cada texto será una secuencia. En este modelo, cada palabra de cada texto se

convierte en un vector (lo que entendemos como *word_embedding*) de 300 dimensiones. Es necesario que todas las secuencias tengan una misma longitud, así que utilizaremos la longitud del texto más largo y rellenaremos con ceros los vectores de las palabras que no existen. Este relleno lo realizaremos por la izquierda, para provocar que las últimas palabras de entrada a la red nunca estén vacías. Fijamos una longitud del texto de *max_length* palabras. Cada vector tiene 300 elementos, por lo que cada muestra de entrada a la red LSTM es una matriz de tamaño *max_length* x 300. Hay 25.409 muestras (número de muestras en *X_train*), por lo que la matriz total de datos tendrá un tamaño 25.409 x *max_length* x 300.

Como hemos limitado el vocabulario a *max_features* términos, los tokens de las secuencias codificadas tienen índices del 0 al *max_features*+2 (donde el 0 es 'Secuencia vacía') y el 1 es 'OOV') y por tanto necesitamos una matriz de *embeddings* de (*max_features*+2) x 300 dimensiones. En nuestro caso, serán 3.442 x 300 dimensiones.

El modelo final es:

```
embedding_layer = Embedding(MAX_NB_WORDS,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False,
                            mask_zero=True)

model = Sequential()
model.add(embedding_layer)
model.add(SpatialDropout1D(0.4))
model.add(LSTM(RNN_DIM, dropout=0.2, recurrent_dropout=0.2, return_sequences=True))
model.add(LSTM(RNN_DIM, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(50, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print(model.summary())
```

Figura 4.18: RNN-LSTM: Modelo final para el caso con *word embeddings* de spaCy.

```

Model: "sequential"
=====
Layer (type)          Output Shape         Param #
=====
embedding (Embedding)    (None, 135, 300)      1032600
spatial_dropout1d (Spatial Dropout1D)   (None, 135, 300)      0
lstm (LSTM)            (None, 135, 50)       70200
lstm_1 (LSTM)          (None, 50)           20200
dense (Dense)          (None, 50)           2550
=====
Total params: 1125550 (4.29 MB)
Trainable params: 92950 (363.09 KB)
Non-trainable params: 1032600 (3.94 MB)

```

Figura 4.19: RNN-LSTM: Sumario del modelo para el caso con *word embeddings* de spaCy.

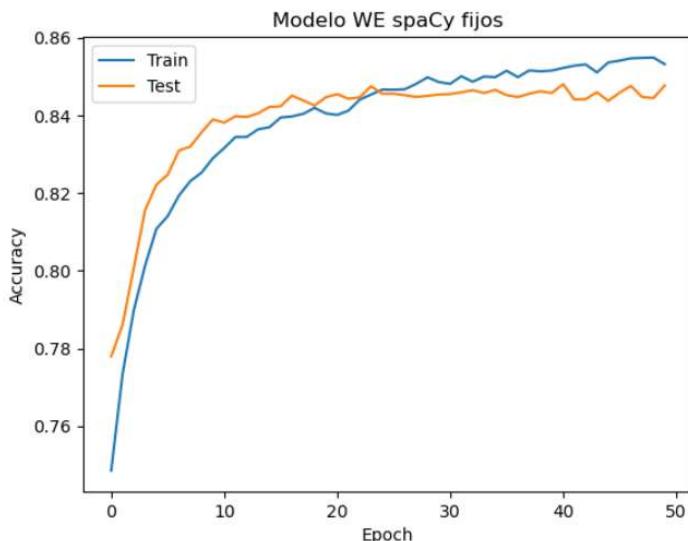


Figura 4.20: RNN-LSTM: Accuracy para el caso con *word embeddings* de spaCy.

Resultados:

- Mejor época: 40
- Accuracy: 84,80%

4.4.1.3. Modelo con fine-tuning

Utilizamos los *word embeddings* de spaCy para hacer *transfer learning*, actualizando los pesos en el entrenamiento, re-entrenando los vectores de spaCy con nuestros textos (poniendo el parámetro *trainable* de la capa de *embedding* a *True*).

```

embedding_layer = Embedding(MAX_NB_WORDS,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=True,
                            mask_zero=True)

model = Sequential()
model.add(embedding_layer)
model.add(SpatialDropout1D(0.5))
model.add(RNN_layer(RNN_DIM, dropout=0.5, recurrent_dropout=0.5))
model.add(Dense(50, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print(model.summary())

```

Figura 4.21: RNN-LSTM: Modelo final para el caso con *word embeddings* entrenables.

```

Model: "sequential_1"
-----  

Layer (type)          Output Shape         Param #
-----  

embedding_1 (Embedding)    (None, 135, 300)       1032600  

spatial_dropout1d_1 (Spati (None, 135, 300)           0  

alDropout1D)  

lstm_2 (LSTM)           (None, 50)            70200  

dense_1 (Dense)          (None, 50)            2550  

-----  

Total params: 1105350 (4.22 MB)  

Trainable params: 1105350 (4.22 MB)  

Non-trainable params: 0 (0.00 Byte)

```

Figura 4.22: RNN-LSTM: Sumario del modelo para el caso con *word embeddings* entrenables.

Resultados:

- Mejor época: 38
- Accuracy: 83,47%

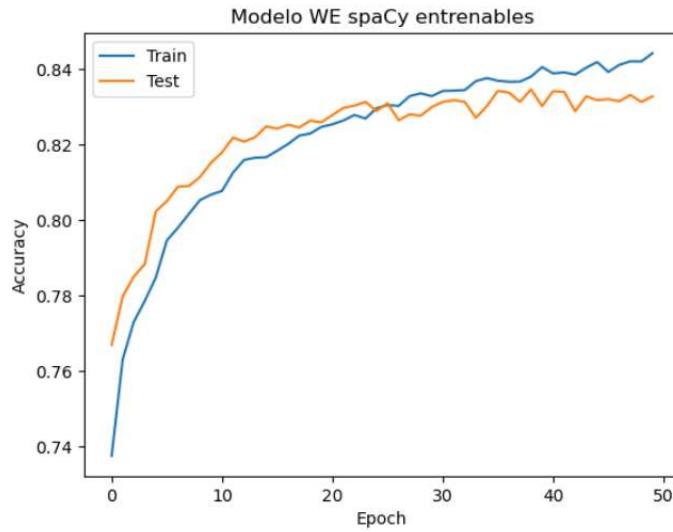


Figura 4.23: RNN-LSTM: *Accuracy* para el caso con *word embeddings* entrenables.

4.4.2. RNN-GRU

La estrategia es idéntica a l caso LSTM, pero ahora cambiamos las capas LSTM por capas GRU.

4.4.2.1. *Word embeddings* propios

```
model = Sequential()
model.add(Embedding(MAX_NB_WORDS,
                    EMBEDDING_DIM,
                    input_length = MAX_SEQUENCE_LENGTH,
                    mask_zero=True))
model.add(SpatialDropout1D(0.5))
model.add(RNN_layer(RNN_DIM, dropout=0.5, recurrent_dropout=0.5))
model.add(Dense(50, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print(model.summary())
```

Figura 4242: RNN-GRU: Modelo final para el caso con *word embeddings* propios.

```

Model: "sequential"
=====
Layer (type)          Output Shape       Param #
=====
embedding (Embedding) (None, 135, 300)    1032600
spatial_dropout1d (Spatial Dropout1D)   (None, 135, 300)    0
gru (GRU)              (None, 50)        52800
dense (Dense)         (None, 50)        2550
=====
Total params: 1087950 (4.15 MB)
Trainable params: 1087950 (4.15 MB)
Non-trainable params: 0 (0.00 Byte)

```

Figura 4.25: RNN-GRU: Sumario del modelo para el caso con *word embeddings* propios.

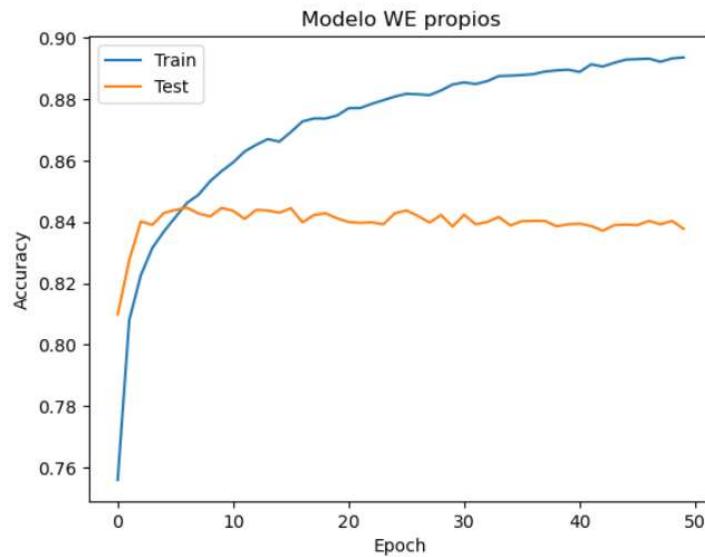


Figura 4.26: RNN-GRU: Accuracy para el caso con *word embeddings* propios.

Resultados:

- Mejor época: 10
- Accuracy: 84,45%

4.4.2.2. Tranfer learning con Word embeddings de GloVe

```

embedding_layer = Embedding(MAX_NB_WORDS,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False,
                            mask_zero=True)

model = Sequential()
model.add(embedding_layer)
model.add(SpatialDropout1D(0.4))
model.add(GRU(RNN_DIM, dropout=0.2, recurrent_dropout=0.2, return_sequences=True))
model.add(GRU(RNN_DIM, dropout=0.2, recurrent_dropout=0.2))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print(model.summary())

```

Figura 4.27: RNN-GRU: Modelo final para el caso con *word embeddings* de spaCy.

```

Model: "sequential"
=====
Layer (type)          Output Shape         Param #
=====
embedding (Embedding) (None, 135, 300)       1032600
spatial_dropout1d (Spatial Dropout1D)        (None, 135, 300)       0
gru (GRU)                (None, 135, 50)        52800
gru_1 (GRU)               (None, 50)           15300
dense (Dense)            (None, 50)           2550
=====
Total params: 1103250 (4.21 MB)
Trainable params: 70650 (275.98 KB)
Non-trainable params: 1032600 (3.94 MB)

```

Figura 4.28: RNN-GRU: Sumario del modelo para el caso con *word embeddings* de spaCy.

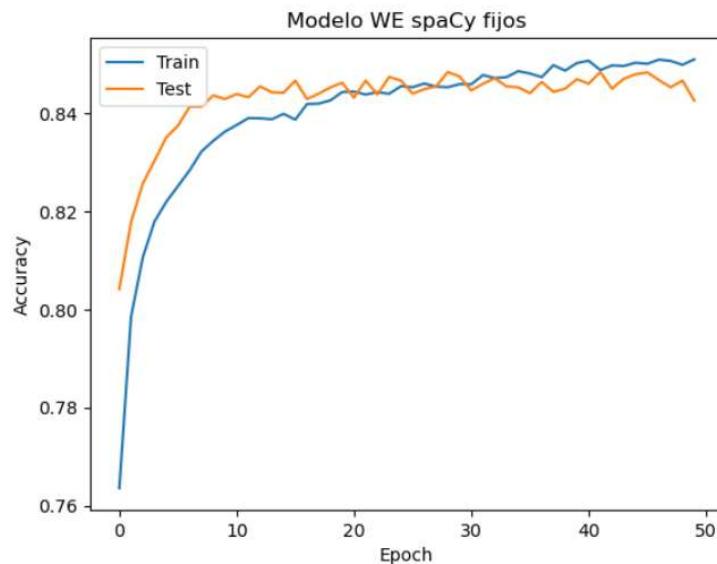


Figura 4.29: RNN-GRU: *Accuracy* para el caso con *word embeddings* de spaCy.

Resultados:

- Mejor época: 28
- Accuracy: 84,84%

4.4.2.3. Modelo con fine-tuning

```
embedding_layer = Embedding(MAX_NB_WORDS,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=True,
                            mask_zero=True)

model = Sequential()
model.add(embedding_layer)
model.add(SpatialDropout1D(0.5))
model.add(RNN_layer(RNN_DIM, dropout=0.5, recurrent_dropout=0.5))
model.add(Dense(50, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print(model.summary())
```

Figura 4.30: RNN-GRU: Modelo final para el caso con *word embeddings* entrenables.

```
Model: "sequential"
=====
Layer (type)          Output Shape         Param #
=====
embedding (Embedding) (None, 135, 300)      1032600
spatial_dropout1d (Spatial Dropout1D)        (None, 135, 300)      0
gru (GRU)                (None, 50)           52800
dense (Dense)             (None, 50)           2550
=====
Total params: 1087950 (4.15 MB)
Trainable params: 1087950 (4.15 MB)
Non-trainable params: 0 (0.00 Byte)
```

Figura 4.31: Sumario del modelo para el caso con *word embeddings* entrenables.

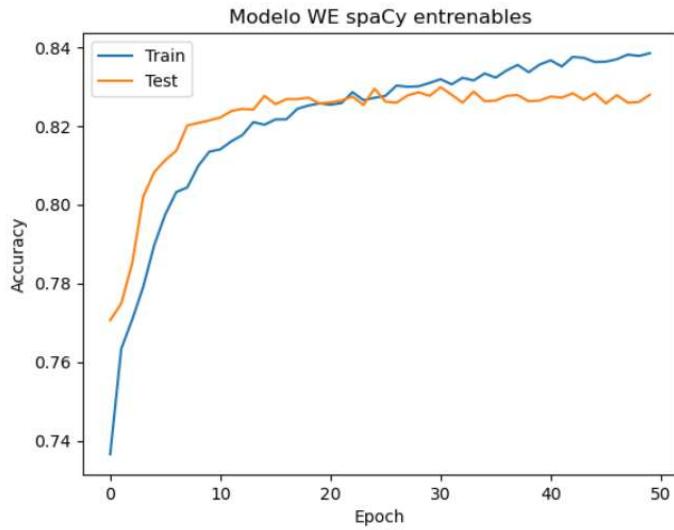


Figura 4.32: RNN-GRU: *Accuracy* para el caso con *word embeddings* entrenables.

Resultados:

- Mejor época: 30
- *Accuracy*: 82,98%

4.4.3. Resultados finales con RNN

Si resumimos los resultados obtenidos con RNN (LSTM y GRU) y con los 3 métodos considerados, obtenemos la siguiente tabla:

	Propio	spaCy	Fine-tuning
RNN-LSTM	84,56%	84,80%	83,47%
RNN-GRU	84,45%	84,84%	82,98%

Tabla 4.3: *Accuracies* obtenidos con los 6 modelos basados en RNN.

Los resultados obtenidos por los 3 modelos son bastante parecidos, con valores de *accuracy* alrededor del 84%, siendo el mejor el modelo con *word embeddings* de spaCy.

4.4.4. sciBERT

Se entrena un modelo BERT del tipo sciBERT. Recordemos que sciBERT es un modelo preentrenado sobre texto científico, con un corpus tomado de artículos científicos de *Semantic Scholar*. sciBERT tiene su propio vocabulario, scivocab, construido para trabajar mejor con el corpus

El entrenamiento tiene las siguientes características:

- Primero se intenta entrenar el modelo sobre un PC portátil, pero el entrenamiento es muy lento y al final se decide entrenarlo en Google Colab, sobre un entorno de ejecución de tipo GPU.
- El tokenizador se implementa con la clase *AutoTokenizer* de la librería *transformers*, con el conjunto preentrenado '*allenai/scibert_scivocab_uncased*'.

```
tokenizer = AutoTokenizer.from_pretrained('allenai/scibert_scivocab_uncased',
                                         from_pt=True,
                                         do_lower_case=True)
```

Figura 4.33: sciBERT: Tokenizador.

- El modelo se implementa a partir de la clase *BertForSequenceClassification* de la librería *transformers*, también con el conjunto preentrenado '*allenai/scibert_scivocab_uncased*'.

```
model = BertForSequenceClassification.from_pretrained('allenai/scibert_scivocab_uncased',
                                                       num_labels=len(label_dict),
                                                       output_attentions=False,
                                                       output_hidden_states=False)
```

Figura 4.34: sciBERT: Modelo.

- Se entrena en 5 épocas. Este entrenamiento tarda aproximadamente 100 minutos.

Los resultados obtenidos en las 5 épocas pueden verse en la figura 4.35.

```
Epoch 1
Training loss: 0.8209288782974845
Validation loss: 0.9314319804720701
F1 Score (Weighted): 0.8001601222790199
Accuracy: 0.835716212570812

Epoch 2
Training loss: 0.7406937621808721
Validation loss: 0.9626450323613046
F1 Score (Weighted): 0.8049420200938799
Accuracy: 0.835716212570812

Epoch 3
Training loss: 0.6712329908551692
Validation loss: 0.9443678833551507
F1 Score (Weighted): 0.8045166162335101
Accuracy: 0.83247909360669

Epoch 4
Training loss: 0.6372855173093995
Validation loss: 0.9443678833551507
F1 Score (Weighted): 0.8045166162335101
Accuracy: 0.83247909360669

Epoch 5
Training loss: 0.6345366817348599
Validation loss: 0.9443678833551507
F1 Score (Weighted): 0.8045166162335101
Accuracy: 0.83247909360669
```

Figura 4.35: sciBERT: Resultados del entrenamiento.

Por lo tanto, tenemos que el valor más grande de *accuracy* se da en la primera y segunda época, con un valor de 83,57%.

Capítulo 5

Conclusiones

5.1. Conclusiones generales

Si repetimos las tablas con los resultados con ML y con DL, tenemos:

Enfoque	MultinomialNB		SGDClassifier		RandomForestClassifier	
	Sin lematizar	Lematizado	Sin lematizar	Lematizado	Sin lematizar	Lematizado
BoW	76,68%	76,81%	82,98%	82,85%	72,77%	72,77%
TF-IDF	73,72%	73,63%	81,10%	81,21%	72,77%	72,77%
2-grams	77,19%	77,01%	84,88%	84,64%	72,77%	72,77%
3-grams	78,71%	78,77%	84,64%	84,61%	72,77%	72,77%
4-grams	79,31%	79,37%	84,70%	84,47%	72,89%	73,02%
5-grams	79,52%	79,52%	84,67%	84,58%	73,04%	73,02%

Tabla 5.1: *Accuracies* obtenidos con los modelos ML.

	Propio	spaCy	fastText
CNN	83,47%	84,07%	83,78%

Tabla 5.2: *Accuracies* obtenidos con los 3 modelos basados en CNN.

	Propio	spaCy	Fine-tuning
RNN-LSTM	84,56%	84,80%	83,47%
RNN-GRU	84,45%	84,84%	82,98%

Tabla 5.3: *Accuracies* obtenidos con los 6 modelos basados en RNN.

Y el modelo sciBERT, con un *accuracy* de 83,57%.

Ordenamos estos resultados en una sola tabla, de mejor a peor *accuracy* obtenido.

Tecnología	Modelo		Accuracy
ML	2-grams	SGD	84,88%
DL	RNN-GRU	spaCy	84,84%
DL	RNN-LSTM	spaCy	84,80%
ML	4-grams	SGD	84,70%
ML	5-grams	SGD	84,67%
ML	3-grams	SGD	84,64%
DL	RNN-LSTM	Propio	84,56%
DL	RNN-GRU	Propio	84,45%
DL	CNN	spaCy	84,07%
DL	CNN	FastText	83,78%
DL	BERT	sciBERT	83,57%
DL	RNN-LSTM	Fine-tuning	83,47%
DL	CNN	Propio	83,47%
DL	RNN-GRU	Fine-tuning	82,98%
ML	BoW	SGD	82,98%
ML	TF-IDF	SGD	81,21%
ML	Todos	NB	73,63% - 79,52%
ML	Todos	RF	72,77% - 73,04%

Tabla 5.4: Accuracies obtenidos con los 6 modelos basados en RNN.

Así, podemos extraer las siguientes conclusiones:

1. El *accuracy* de estos modelos abarca un rango que va del 72,77% (Random Forest) hasta el 84,84% (RNN-GRU con spaCy).
2. Los mejores modelos son los basados en n-gramas con SGD y los de spaCy con RNN.
3. Los peores modelos son los basados en *Random Forest*.
4. Para los modelos en ML, los de n-gramas obtienen mejores resultados.
5. Para los modelos de ML, BoW supera a TF-IDF.
6. En DL, Los RNN-LSTM son los mejores, después van los RNN-GRU y, por último, los CNN. Sin embargo, las diferencias son mínimas.
7. El modelo sciBERT, a pesar de ser tan novedoso, no consigue mejores resultados que modelos más simples.

Con todo esto, y en vistas a una eventual puesta en producción de un modelo, en principio parecería que el modelo elegido sería el SGD con bigramas, el RNN-LSTM o el RNN-GRU, que son los que obtienen mejor *accuracy*. Pero al poner un modelo en

producción el rendimiento no es el único factor a tener en cuenta, ya que puede haber otros factores que pueden ser importantes. Por ejemplo, el despliegue de un modelo con GloVe de spacy ha de cargar en memoria el *word embedding*, que puede llegar a ocupar mucha memoria y, por lo tanto, puede no ser adecuado para según qué aplicaciones.

5.2. Trabajo futuro

Como no puede ser de otra manera en un TFM, siempre hay aspectos del mismo que se podrían mejorar, o que se podrían llevar un paso más adelante. En este TFM, planteo dos trabajos futuros que pueden ser la continuación del trabajo desarrollado:

- Entrenar más modelos basados en *Transformers*, para ver si podemos mejorar los resultados obtenidos.
- Entrenar modelos que, además del texto del campo conclusiones, tengan en cuenta las medidas físicas que contiene el fichero medidas.csv.

Bibliografía

- Géron, A.: *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow*, Third Edition. 2023. O'Reilly Media.
- Starmer, J.: *The StatQuest Illustrated Guide to Machine Learning*. 2022. Joshua Starmer.
- Serrano, L. G.: *Grokking Machine Learning*. 2021. Manning Publications.
- Topol, E.: *Deep Medicine*. 2019. Basic Books.
- Müller, A. C.; Guido, S.: *Introduction to Machine Learning with Python*. 2017. O'Reilly Media.
- Tunstall, L.; Werra, L; Wolf, T.: *Natural Language Processing with Transformers*. 2022. O'Reilly Media.
- Kochmar, E.: *Getting Started with Natural Language Processing*. 2021. Manning Publications.
- Krohn, J.: *Deep Learning Illustrated*. 2020. Pearson Education.
- Goodfellow, I.; Bengio, Y.; Courville, A.: *Deep Learning*. 2016. Massachusetts Institute of Technology.
- Raschka, S.: *Python Machine Learning*. 2015. Packt Publishing.
- Trask, A. W.: *Grokking Deep Learning*. 2019. Manning Publications.
- Torres, J.: *Python Deep Learning*. 2020. Marcombo.
- Beltagy, I.; Lo, K.; Cohan, A.: *SciBERT: A Pretrained Language Model for Scientific Text*. 2019. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pages 3615–3620, Hong Kong, China. Association for Computational Linguistics.