UNIVERSIDAD POLITÉCNICA DE CARTAGENA (UPCT)

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIONES (ETSIT)

Grado en Ingeniería Telemática

ANÁLISIS DE TEXTOS MEDIANTE TÉCNICAS NLP PARA LA CATEGORIZACIÓN DE USUARIOS

Autor
Andrés Zapata García

Director
Javier Vales Alonso

Cartagena
Julio 2021

Ficha del proyecto

Tipo de proyecto	Específico
Curso académico	2020-2021
Autor del proyecto	Andrés Zapata García
Director del proyecto	Javier Vales Alonso
Departamento	Tecnologías de la Información y las Comunicaciones
Área de conocimiento	Ingeniería Telemática
Título en español	Análisis de textos mediante técnicas NLP para la categorización de usuarios
Título en inglés	User categorization using text analysis NLP methods
Objetivos	1. Familirizarse con los métodos de aprendizaje automático basados en técnicas de NLP (natural language processing) 2. Implementar un procedimiento basado en estas técnicas que permita la clasificación automática de textos y con ello la categorización de usuarios 3. Entrenar y probar dicho método en bases de datos de textos de test, por ejemplo, twitter user data (disponible en https://data.world/data-society/twitter-user-data)
Fases	1. Estudio base de machine learning 2. Estudio de redes neuronales densas 3. Estudio métodos NLP 4. Selección de algoritmo 5. Implementación de algoritmo 6. Pruebas y validación 7. Documentación

Resumen

El Procesado del Lenguaje Natural es una de las principales tendencias en el ámbito del aprendizaje máquina y la inteligencia artificial. Una vez que somos capaces de procesar la información a nivel computacional, lograr que una máquina comprenda el lenguaje de un humano nos abre un amplio abanico de posibilidades. Por ejemplo, nuestra tarea de clasificación de usuarios mediante el análisis de sentimientos puede ser de gran utilidad a la hora de saber cuál es la reacción de un determinado público ante un acontecimiento de interés.

El objetivo del proyecto se trata de, dado un tweet redactado como entrada, poder predecir si expresa un sentimiento positivo o negativo, en función del contenido. Esta tarea, a priori sencilla para una persona, supone un desafío para un computador: existen multitud de variables dentro del lenguaje que se traducirán en millones de parámetros en nuestro problema.

Basándonos en distintos modelos matemáticos, cuya base común es el uso de redes neuronales, aportaremos distintas métricas que nos ayuden a concluir qué modelo nos devuelve unos resultados más satisfactorios. Pasaremos de soluciones más tradicionales dentro del NLP como son las redes neuronales convolucionales, hasta otras más innovadoras como ELMo o BERT.

Abstract

Natural Language Processing is one of the main trends in the field of machine learning and artificial intelligence. Once we are able to process information at a computational level, enabling a machine to achieve the understanding of human language opens up a range of possibilities. For instance, our task of classifying users by means of sentiment analysis might be useful figuring out the response of an audience to an event of interest.

The aim of the project is to, given a tweet as input, be able to predict if the expressed sentiment is positive or negative, depending on the content. This task, which is easy for a person, entails a challenge for a computer: there are multiple variables in language which will be translated into millions of parameters in our problem.

Based on several mathematical models, based on the use of neural networks, we will provide different metrics which will help to conclude what model give us the most satisfactory results. We will explore more traditional solutions in NLP, such as convolutional neural networks, and more innovative ones, as ELMo or BERT.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción y objetivos

Hoy en día, saber analizar y tratar la información para obtener un valor añadido se ha convertido en una herramienta muy valiosa. Quizás, al hablar de tratamiento de datos podemos pensar en las empresas cuyo negocio está basado en esto, donde se traduce en un beneficio económico. Sin duda, esta se ha convertido en una de las principales motivaciones en el avance de la ciencia de datos. No obstante, los beneficios que podemos obtener de esta ciencia también pueden repercutir en nosotros como individuos y sociedad.

Un ejemplo de esto se puede observar durante el terremoto de Haití en 2010. Dado que los desastres de tal magnitud provocan grandes movimientos de la población, la ONG sueca Flowminder decidió analizar la actividad de 1900 millones de usuarios de teléfono móvil durante 42 días antes y 341 días después del terremoto. Mediante el análisis de estos datos, se buscó el objetivo de entender y predecir los movimientos de la gente afectada, para así poder proveer ayuda humanitaria de manera más eficiente y apoyar la reconstrucción social [11].

Las posibilidades son infinitas. El ámbito en el que nos vamos a centrar, el procesamiento del lenguaje natural, si bien es muy distinto al ejemplo que hemos comentado, tiene una base común que comparten: el tratamiento de un conjunto de información para conseguir un objetivo determinado. En nuestro caso, las aplicaciones son diversas: reconocimiento de entidades, resumen de textos, traducción automática o sistemas conversacionales, entre otras.

Podemos definir el procesamiento del lenguaje natural -en inglés, Natural Language Processing, NLP- como el área de desarrollo y aplicación que se enfoca en cómo los computadores pueden ser utilizados para entender el lenguaje natural con el objetivo de realizar diversas tareas. El NLP está basado en distintas disciplinas, tales como la ciencia de computadores y la información, la lingüística, las matemáticas o la inteligencia artificial [12].

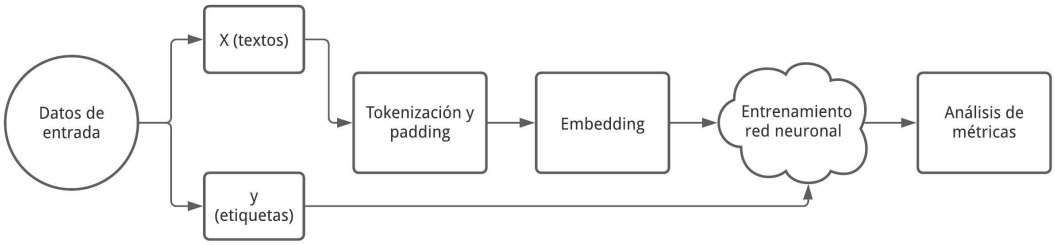


Figura 1.1: Diagrama del proceso a seguir

En nuestro proyecto, la tarea que llevaremos a cabo será la de análisis de sentimientos, pudiendo realizar así una categorización de usuarios en función a los parámetros deseados. Para ello, construiremos un modelo que, dada una entrada (un texto redactado) nos devuelva una salida (categoría a la que pertenece dicho texto).

Las entradas se toman de un conjunto de datos (dataset), que está formado por tweets ya etiquetados que expresan un sentimiento positivo o negativo. Entre la variedad de datasets que podemos encontrar disponibles, a la hora de empezar el proyecto también se tuvo en cuenta otro conjunto de datos que identificaba si un tweet expresaba sentimiento de odio o no, pero fue descartado debido a una peor calidad de los datos y su extensión demasiado reducida.

Por otra parte, la herramienta base que utilizaremos serán las redes neuronales, en las cuales utilizaremos algoritmos como GloVe, ELMo o BERT. Todo esto será detallado en los siguientes capítulos.

Para tener una idea general del flujo de trabajo que vamos a seguir a la hora de tratar los textos, seguiremos el diagrama que se muestra en la figura 1.1. Estos pasos serán detallados en los capítulos 4 y 5, cuando definamos el planteamiento del problema y lo desarrollemos.

Finalmente, mencionar que el código completo del proyecto se ha subido a un repositorio de GitHub, y se puede acceder a él mediante el siguiente enlace: `https://github.com/andreszpt/nlp-user-classification`.

Capítulo 2

Conceptos generales y herramientas

2.1. Machine learning y tipos

La base sobre la que se apoya una tarea de procesado del lenguaje natural como la nuestra es el *machine learning*. Machine learning -en español, aprendizaje máquina- es la ciencia de programar un computador para que este pueda aprender a realizar una tarea a partir de los datos de entrada, sin ser programado para realizar explícitamente dicha tarea [13].

Esta definición se vuelve clave para nuestra tarea de NLP. El modelo que deseamos crear no debe funcionar para unos datos específicos con unas características concretas, sino que debemos plantearlo de forma que pueda ir aprendiendo a clasificar a partir de esos propios datos de entrada. Este proceso se divide en dos fases: la fase de entrenamiento, donde nuestro modelo *aprende* en base a los ejemplos con los que alimentamos un modelo matemático, y la fase de evaluación o validación, donde podemos comprobar cómo generaliza para el resto de casos.

El aprendizaje se puede realizar de distintas formas, apareciendo así distintos tipos de machine learning. De forma general, podemos clasificar estos tipos en función a distintos criterios [13]:

- Si el sistema se entrena con supervisión humana o no: aprendizaje supervisado, no supervisado, semi-supervisado o aprendizaje por refuerzo.
- Si el sistema aprende incrementalmente durante el proceso: aprendizaje online o por lotes.
- Si el sistema trabaja comparando nuevos datos con datos conocidos o detectando patrones: aprendizaje basado en instancia o basado en modelo.

El problema que vamos a tratar lo podemos clasificar como aprendizaje supervisado, por lotes y basado en modelo.

En primer lugar, se trata de aprendizaje supervisado puesto que los datos con los que vamos a trabajar ya incluyen la solución que buscamos, el sentimiento en este caso, llamada *etiqueta*. Tomamos la decisión de trabajar con conjuntos de datos etiquetados ya que los algoritmos para trabajar con este tipo de datos son más accesibles para un trabajo de estas características. El proceso de etiquetado de datos es independiente a la tarea que vamos a realizar, y el proceso de obtención y manipulación de estos datos estará detallado en la sección 4.3.

Por otra parte, el aprendizaje se realiza por lotes, dado que nuestro sistema no será capaz de ser entrenado ni aprender una vez está funcionando. En su lugar, será entrenado utilizando todos los datos disponibles, de forma *offline*. Cuando se ha entrenado para dar una solución correcta, se lanza y ya no vuelve a ser entrenado. Aunque esto supone que el sistema ya funcionaría sin necesidad de tener que revisarlo, este proceso offline se traduce en el consumo de una gran cantidad de recursos que no podemos despreciar.

Por último, definimos el sistema como basado en modelo puesto que necesitamos, de alguna forma, generalizar nuestra solución para resolver de forma flexible el problema, realizando predicciones. Por ello, esta es una decisión de diseño obligada. En concreto, de las soluciones que vamos a explorar, todas están basadas en el uso de redes neuronales, cuyo funcionamiento entramos a describir en detalle en la sección siguiente.

2.2. Redes neuronales

Las redes neuronales son el modelo en el cual se va a basar el sistema que desarrollaremos en el proyecto. En las últimas décadas, el enfoque del machine learning hacia los problemas NLP se ha apoyado en modelos superficiales de grandes dimensiones y pocas características a entrenar (tales como máquinas de vectores de soporte o regresión logística). Más recientemente, las redes neuronales basadas en representaciones densas de vectores ha conseguido resultados destacados en distintas tares NLP [14].

Una red neuronal, también conocida como red neuronal artificial -ANN, por sus siglas en inglés- es una rama dentro del machine learning que tiene una gran relación con los algoritmos de aprendizaje profundo -deep learning-. Su nombre y estructura se inspira en el cerebro humano, simulando la forma en la que las neuronas biológicas se comunican entre ellas [15].

Si entramos en más detalle sobre cómo se inspiran unas en otras, ambas tratan de buscar un aprendizaje a través de la experiencia y la extracción de conocimiento genérico a partir de un conjunto de datos. Sin embargo, existen diferencias significativas entre

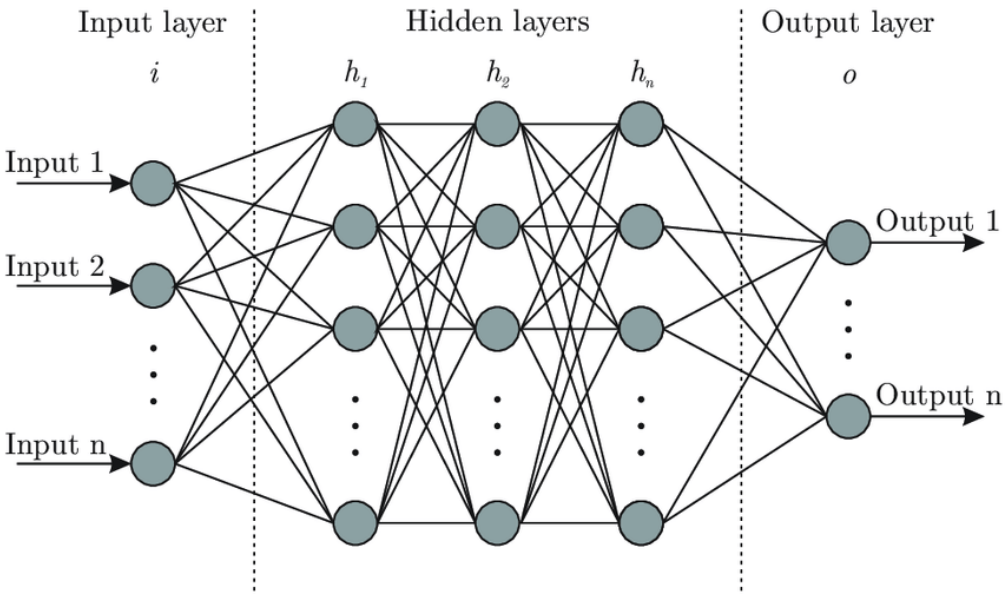


Figura 2.1: Esquema de las capas [1]

el cerebro biológico y los computadores convencionales. El cerebro está compuesto por millones de procesadores elementales o neuronas, interconectadas entre sí, formando redes, mientras que los computadores suelen implementar una arquitectura de tipo Von Neumann, basada en un microprocesador muy rápido que ejecuta una serie de instrucciones complejas en de forma fiable. La diferencia reside en que las neuronas biológicas no necesitan ser programadas, ya que aprenden a partir de los estímulos que reciben del entorno y operan siguiendo un esquema masivamente paralelo, distinto al procesamiento en serie de los computadores convencionales [16].

Las redes neuronales artificiales están compuestas por capas de nodos: una capa de entrada, una o más capas ocultas y una capa de salida. Cada nodo o neurona se conecta con otro, teniendo un peso y un sesgo asociados. Si la salida de cada nodo está por encima del sesgo especificado, se activará el nodo, enviando los datos a la siguiente capa. En caso contrario, no se pasará esta información. En la figura 2.1 podemos ver la representación gráfica de esta estructura [15].

La unidad básica que compone una red neuronal es la neurona, cuya estructura podemos ver en la figura 2.2. Se trata de una unidad que lee cada entrada x, y la multiplica por un peso determinado w. Estas multiplicaciones se suman, resultando en z. A partir de aquí, el resultado que obtendríamos sería una función lineal, algo que queremos evitar debido a las limitaciones de complejidad que presenta en el modelo. Por ello, hacemos uso de la función de activación f. La activación se trata de una función no-lineal, siendo las más populares algunas como *ReLu*, *softmax* o *sigmoid*. Mediante el uso de estas

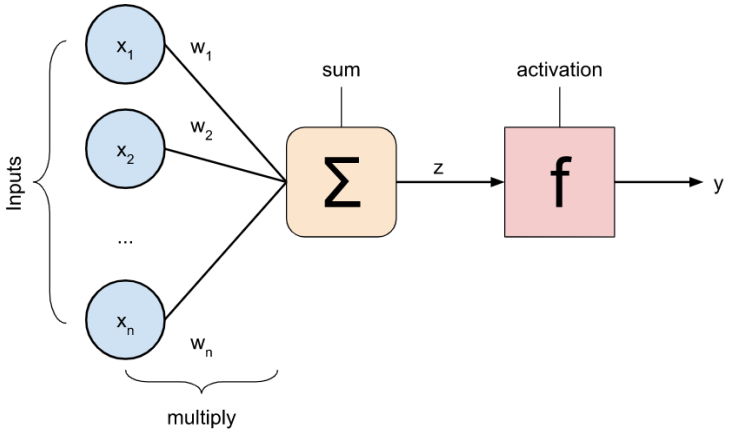


Figura 2.2: Representación de una única neurona [2]

activaciones la red neuronal es capaz de aprender funciones de transferencia no lineales, siendo esta su característica clave para problemas con una complejidad mayor.

Analíticamente:

$$y = f(z) = f(\sum_i w_i x_i) = f(w^T x) \tag{2.1}$$

Si queremos escribirlo de forma simplificada, quedaría de la siguiente forma:

$$y = f(wx) \tag{2.2}$$

La ecuación 2.2 guarda cierta similitud con la ecuación de la recta $y = wx + b$, habiendo omitido un parámetro, el sesgo (b). Este parámetro es aquel que nos permite desplazar la línea o predicción para poder predecir los datos de mejor manera. Cabe aclarar que en ocasiones la ecuación puede contener el parámetro b sumado o no. Se debe a que podemos *absorber* el sesgo con el peso, haciendo que siempre haya un input adicional, que sea constante a 1, siendo $x' = [x, 1]$, $w = [w_1, w_0]$, y teniendo entonces $y = w_1 x + w_0$ [2].

2.2.1. Redes neuronales convolucionales

Dentro de este ámbito, durante gran parte del proyecto nos centraremos en un subtipo, las redes neuronales convolucionales -CNN, por sus siglas en inglés-. Si bien es cierto que su éxito se atribuye principalmente al procesado de imágenes, debido a la naturaleza de la operación de convolución en 2 dimensiones, cabe destacar sus posibilidades dentro del

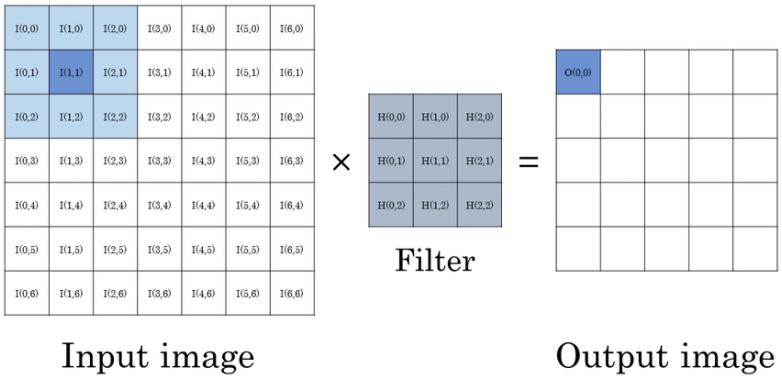


Figura 2.3: Convolución a una imagen 7x7 con un filtro 3x3 [3]

procesamiento del lenguaje natural. Para entender primero cómo funciona el proceso, lo aplicaremos a una imagen y después lo relacionaremos con NLP.

En primer lugar, una CNN es el conjunto de varias capas de convolución junto con funciones de activación no lineales, como *ReLu* o *tanh*, que se aplican a los resultados [17]. En las redes neuronales densas, conectamos cada neurona de entrada con cada neurona de salida, lo que se llama una capa completamente conectada. Este proceso cambia en las redes convolucionales. En este caso, lo que hacemos es realizar convoluciones en la capa de entrada para computar la salida. Cada capa aplica distintos filtros, normalmente cientos, y combina los resultados. Estos filtros se basan en la multiplicación y suma de cada valor por unos pesos determinados, propio de la convolución. El proceso se puede ver gráficamente en la figura 2.3.

En la fase de entrenamiento, la CNN aprende automáticamente los valores de los filtros basándonos en la tarea que queremos realizar. Por ejemplo, en una tarea de clasificación de imágenes, la red puede aprender a detectar bordes a partir de píxeles en la primera capa, usar esos bordes para detectar formas en la segunda capa, y utilizar estas formas para determinar características de más alto nivel, como rasgos faciales. La última capa consiste en un clasificador (ANN) que utiliza estas características para dar un resultado final.

En medio de estas capas, podemos encontrar unas capas de *pooling* (muestreo), cuya función es reducir las dimensiones de nuestra imagen, aplicando un filtro que, por ejemplo, solo se quede con los valores máximos dentro de una ventana determinada. Su uso se debe a que en este tipo de problemas, si utilizamos muchos filtros y las dimensiones son altas, la cantidad de datos puede escalar muy rápido. El funcionamiento lo podemos ver en la figura 2.4.

Finalmente, si juntamos estas capas, final aplanamos nuestra salida y la unimos con una red neuronal densa, podemos encontrarnos con una arquitectura de la forma en la

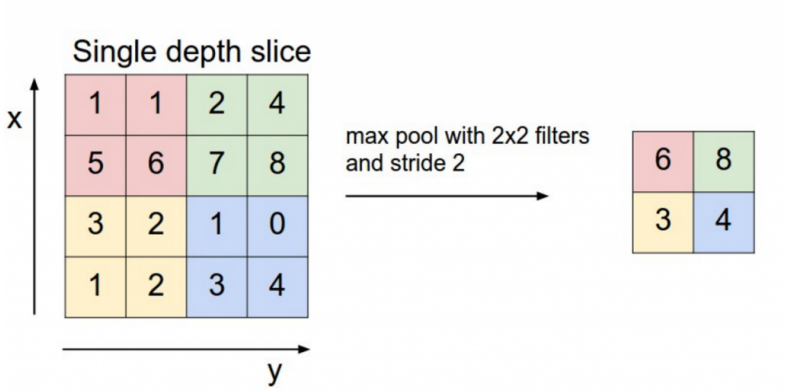


Figura 2.4: Pooling en una imagen 4x4 para pasar a una imagen 2x2 [4]

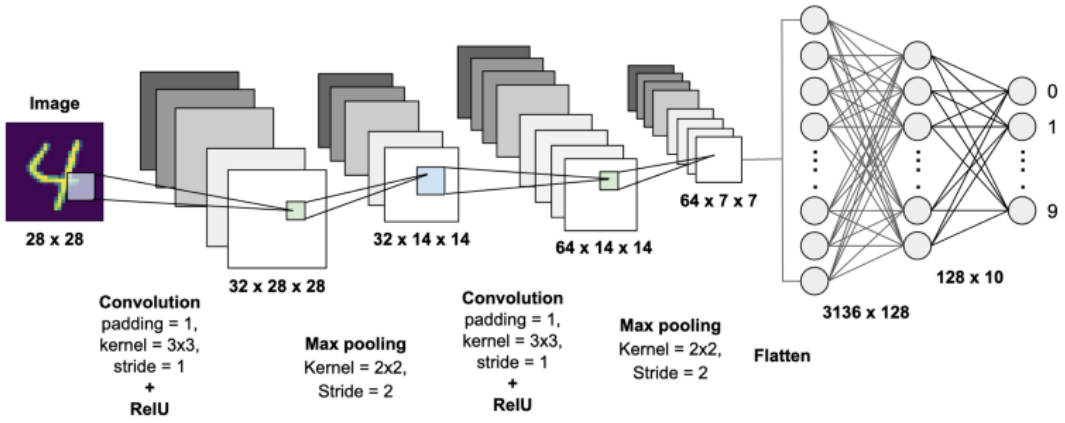


Figura 2.5: CNN para la clasificación de dígitos manuscritos MNIST [5]

que aparece en la figura 2.5. El término *padding* hace referencia a cómo tratar los bordes de las imágenes cuando el filtro que la recorre la sobrepasa, y el término *stride* se refiere a las unidades que se mueve el filtro mientras recorre la imagen.

Si bien esta arquitectura está orientada al procesado de imágenes, el procesado del lenguaje guarda muchas similitudes. En NLP, comenzamos con nuestras oraciones dadas, las cuales pasan un proceso de *tokenización*, consiguiendo la llamada *embedding matrix*, donde las filas son la representación vectorial de cada una de las palabras convertidas en tokens. Esta matriz oración la usaremos para convertir las palabras a vectores, haciendo que la oración se transforme en un conjunto de vectores, algo similar a una imagen. Este proceso es el llamado *embedding*, en el que entraremos en más detalle en las secciones 4.3 y 4.4. Una vez tenemos la oración vectorizada, podemos aplicar convoluciones de una dimensión (recorriendo varias palabras a la vez) y pooling. La figura 2.6 muestra el

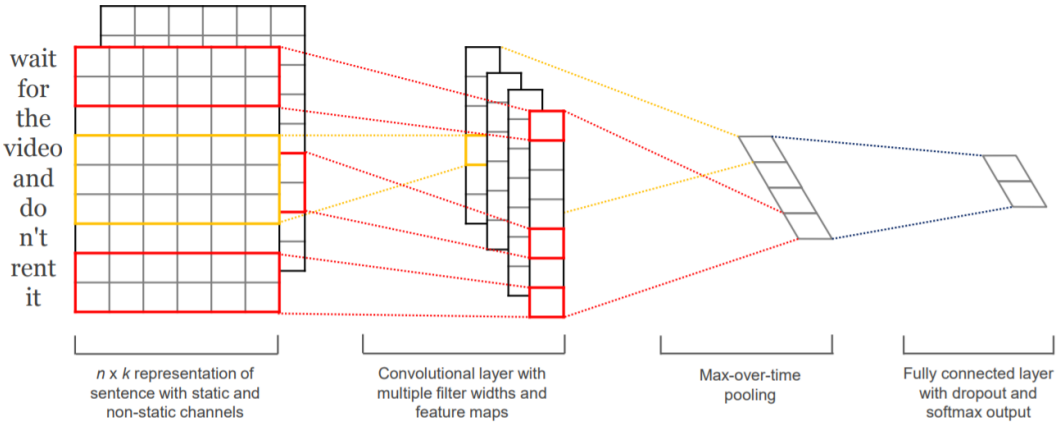


Figura 2.6: Arquitectura del modelo para una oración de ejemplo [6]

proceso con una oración de ejemplo.

2.3. Métodos NLP

El procesamiento del lenguaje natural debe realizarse de forma sistemática, dividiéndolo en partes, agregando elementos gramaticales e identificando elementos interesantes. Los siguientes son algunos de los elementos comunes al procesamiento en NLP [18]:

- **Tokenización**: Consiste en dividir el texto en palabras.
- **Normalización**: Estandariza todas las palabras, por ejemplo, convirtiendo todas las palabras en mayúsculas o minúsculas.
- **Eliminación de palabras redundantes**: Consiste en omitir o eliminar palabras que son redundantes, como artículos o preposiciones, que no aportan normalmente significado al texto.
- **Bolsa de palabras**: Una oración se considera como un conjunto de palabras, sin tener en cuenta la gramática ni el orden de las palabras.
- **N-gramas**: Secuencia continua de palabras adyacentes en una oración, necesarias para obtener el significado correctamente. Por ejemplo, machine learning es un bigrama.
- **TF (frecuencia del término)**: Número de veces que aparece una palabra en un mensaje o una oración; indica la importancia de esa palabra.

Todos estos elementos los iremos empleando a medida que trabajemos con nuestro dataset, ya que antes de alimentar nuestra red neuronal, será necesario pasar por un pre-procesamiento, donde trataremos los datos para optimizar los resultados.

2.4. Entorno de desarrollo y herramientas

Para realizar el proyecto, el lenguaje de programación seleccionado ha sido Python, principalmente por la variedad y calidad de paquetes orientados a la ciencia de datos y machine learning. Además, se trata de un lenguaje con una curva de aprendizaje accesible para alguien que nunca ha desarrollado con él.

A la hora de preparar el entorno de desarrollo, decidimos trabajar con un entorno virtual de Anaconda, que nos permite una fácil integración con los paquetes necesarios para el proyecto y poder trabajar con las versiones necesarias de cada uno. Principalmente, los paquetes que vamos a emplear son los siguientes:

- **Re** [19]: Es el módulo para trabajar con expresiones regulares que utilizamos en Pyhton. Una expresión regular es una secuencia de caracteres que forma un patrón de búsqueda, pudiendo así buscar si una determinada cadena de caracteres coincide con dicho patrón. El uso práctico que le daremos será el de hacer el preprocesado del texto mediante su limpieza, eliminando aquellos caracteres o expresiones que no nos interesan a la hora de analizar un texto: URLs, nombres de usuario, números o caracteres especiales.

- **Pandas** [20]: Se trata de una librería open source que permite el uso de estructuras de datos de forma fácil y potente, así como herramientas para análisis de datos. Su funcionalidad se basa en el uso de DataFrames, estructuras de datos que permiten la manipulación de datos y la indexación. En nuestro proyecto nos permitirá trabajar con los datasets de forma rápida y eficiente, y destaca su integración con el resto de paquetes.

- **Matplotlib** [21]: Es una librería destinada a crear visualizaciones estáticas, animadas o interactivas. Pese a su multitud de opciones, en nuestro proyecto el uso principal será aportar una visión de los resultados que obtenemos tras el entrenamiento de nuestras redes neuronales. En concreto, nos servirá para comparar los resultados de pérdidas y precisión entre el conjunto de datos de entrenamiento y validación. De una forma visual, podemos saber cómo están funcionando nuestros modelos.

- **Scikit-learn** [22] [23]: Es otra librería destinada al aprendizaje automático, incluyendo algoritmos de clasificación, regresión y análisis de grupos. Cuenta con un

gran número de algoritmos, aunque nosotros nos limitaremos a utilizar funciones para el tratamiento de datos, como la creación de subconjuntos de datos o el escalado de estos, ya que de los algoritmos se encargará una librería de más alto nivel, *Keras*, que comentaremos a continuación.

- **Keras** [24]: Se trata de una librería de código abierto pensada para trabajar con redes neuronales. Se ejecuta sobre TensorFlow, Microsoft Cognitive Toolkit o Theano. Nosotros trabajaremos sobre Tensorflow. Es una librería de más alto nivel, por lo que resulta más sencilla para el usuario, sin perder la potencia de Tensorflow. Proporciona bloques modulares para desarrollar los modelos de aprendizaje profundo que deseamos. Principalmente, en nuestro proyecto nos aprovecharemos de sus módulos Tokenizer y pad-sequences para transformar las palabras a secuencias y hacer que todas las oraciones tengan la misma longitud, respectivamente, así como de sus módulos de capas para formar nuestras redes neuronales.

Por último, cabe destacar el uso de los cuadernos de Jupyter, una herramienta que facilita el desarrollo y las pruebas del código desarrollado. Su funcionamiento se basa en el uso de celdas, que pueden ejecutarse de forma individual, aportando una gran flexibilidad. Estos cuadernos también se integran con IPython, un shell interactivo que ofrece una interfaz al usuario más manejable y fácil de usar.

Debido a la cantidad de recursos que consumen las simulaciones del proyecto, intentamos hacer pruebas en Google Colab, un entorno de ejecución de cuadernos tipo Jupyter desarrollado por Google que permite ejecutar y programar Python desde el navegador. Descartamos esta opción, debido a que necesitamos un entorno estable, en el que llevar un control de las versiones de cada paquete y que podamos tener control sobre las sesiones activas. Por ello, finalmente optamos por conectarnos de forma remota a un servidor de la Universidad, proporcionado por el tutor de este proyecto. Este equipo cuenta con una NVIDIA GeForce GTX Titan, una tarjeta gráfica muy potente de gama alta, pudiendo así hacer las simulaciones con mayor velocidad.

Capítulo 3

Trabajos relacionados

Para comprender mejor qué tipo de problemas son capaces de resolver las redes neuronales, en concreto las convolucionales, en este capítulo trataremos un artículo publicado en la Revista de Investigación en Tecnologías de la Información titulado *Modelo para detectar el uso correcto de mascarillas en tiempo real utilizando redes neuronales convolucionales* [25]. Además, veremos otros trabajos que trabajan con este tipo de problemas.

El objetivo del estudio es construir un modelo que, dada una imagen de una persona, sea capaz de detectar si se está usando la mascarilla o no, y en caso afirmativo saber si se está haciendo buen uso de ella.

3.1. Planteamiento

En primer lugar, es necesaria la creación de un dataset con imágenes etiquetadas de tres clases: rostros de personas que lleven mascarilla de forma correcta, que la lleven de forma incorrecta y que no la lleven. En este caso, la creación del dataset fue se hizo de forma manual, a través de imágenes de Google, guardando 1000 muestras para cada clase. Hecho esto, se definen hiperparámetros como la tasa de aprendizaje (0.0001), el número de iteraciones [5, 10, 15, 20, 25] y el tamaño del lote (32).

Para realizar el pre-procesamiento de las imágenes, se ajusta su tamaño a 224x224 píxeles, dado que este es el tamaño con el que trabaja MobileNetV2 e ImageNet. Por un lado, MobileNetV2 es una arquitectura CNN cuyo propósito es proporcionar un buen funcionamiento en dispositivos móviles, mientras que ImageNet es un banco de datos donde las imágenes se clasifican en 22.000 clases distintas, facilitando la tarea de clasificación.

Entonces, las imágenes se convierten a arrays, generando dos listas, una para las propias imágenes y otra para las etiquetas. El modelo base se crea utilizando MobileNetV2,

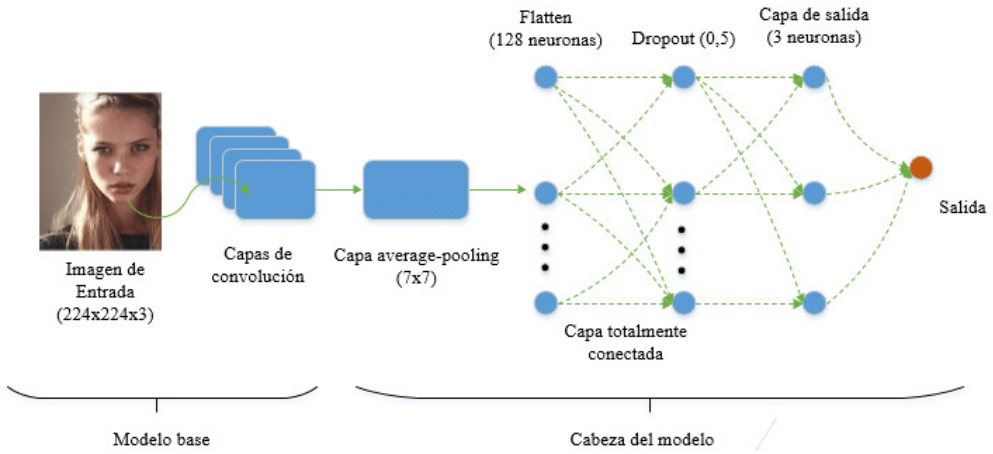


Figura 3.1: Arquitectura del modelo para una imagen dada

con los pesos de la red neuronal ImageNet y con el formato de entrada de 224x224x3. La tercera dimensión que multiplica las dimensiones de la imagen se debe a los 3 canales de color, ya que no se trata de blanco y negro.

En la salida del modelo base se encuentra la cabeza del modelo, que implementa una capa de *average pooling* junto con una capa de aplanamiento, que conecta con la red neuronal que clasifica los datos en 3 categorías, gracias a sus tres neuronas de salida.

Cabe mencionar que en este estudio se separa el total de muestras, dejando un 80 % para el conjunto de entrenamiento, y el 20 % restante para el conjunto de prueba. Hecho esto, el modelo construido es el que se muestra en la figura 3.1.

3.2. Resultados

En la figura 3.2 podemos ver cómo se comporta el modelo para 20 y 25 iteraciones. Cabe destacar que los resultados son realmente precisos, siendo el caso de mal uso de la mascarilla el que peor métricas obtiene, aunque no llega a bajar del 91 % de precisión.

Si entramos más en detalle en el entrenamiento de la red, si nos fijamos en la gráfica de la figura 3.3, donde se comparan los resultados de entrenamiento y validación, vemos cómo la precisión sube rápidamente en las 5 primeras iteraciones, mientras que las pérdidas bajan en ambos conjuntos. Éste es el comportamiento que esperamos en el entrenamiento de estos sistemas.

Este caso de estudio es un ejemplo concreto de cómo el uso de datos y redes neuronales puede ayudar a resolver un problema de la vida diaria de forma efectiva.

Tabla 4. Métricas del desempeño del modelo con veinte iteraciones.

Clase	Exactitud	Precisión	Exhaustividad	Valor F1
With mask	0.95	0.97	0.97	0.97
Without mask	0.95	0.97	0.88	0.92
Bad mask	0.95	0.91	0.99	0.95

Fuente: Elaboración propia.

Tabla 5. Métricas del desempeño del modelo con veinticinco iteraciones.

Clase	Exactitud	Precisión	Exhaustividad	Valor F1
With mask	0.94	0.96	0.98	0.97
Without mask	0.94	0.97	0.87	0.92
Bad mask	0.94	0.91	0.98	0.94

Fuente: Elaboración propia.

Figura 3.2: Métricas de las simulaciones del modelo para 20 y 25 iteraciones

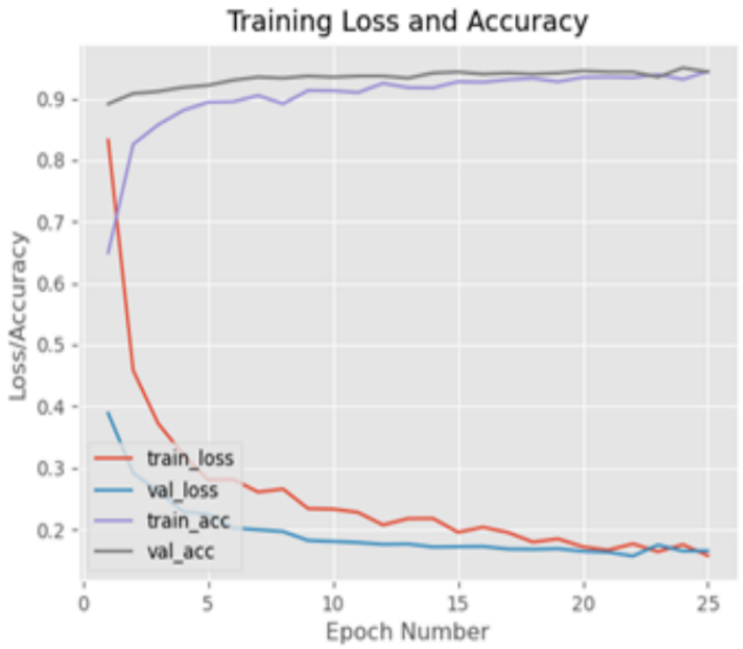


Figura 3.3: Gráfica de los resultados de entrenamiento y validación en 25 iteraciones

3.3. Otros trabajos

Sin entrar en tanto detalle, existen multitud de trabajos relacionados con tareas NLP que nos sirven para situarnos y conocer de qué somos capaces de hacer con este tipo de modelos.

Un caso muy similar al nuestro se trata en el artículo *Sentiment Analysis and Topic Detection of Spanish Tweets: A Comparative Study of NLP Techniques* [26]. El artículo se enfoca en el análisis de sentimientos y la detección de temas. Para realizar la clasificación se utiliza WEKA, una colección de algoritmos de machine learning que incluye algunos como regresión o selección de atributo. Para su desarrollo se utiliza un conjunto de datos de alrededor de 70 000 tweets, junto con 7000 tweets adicionales que ya habían pasado por las tareas de clasificación de sentimiento y tema. En concreto, se usó este último conjunto reducido, destinando 5000 tweets para entrenamiento y 2000 para evaluación. Las mejores precisiones obtenidas (58 % para temas y 42 % para sentimientos) demuestran que en este trabajo, a pesar de trabajar con clasificaciones multiclase, los algoritmos explorados no son muy efectivos, teniendo aún mucho margen de mejora.

Si nos alejamos de las tareas de clasificación, el artículo *Emotion-Cause Pair Extraction: A New Task to Emotion Analysis in Texts* [27] muestra otro tipo de tareas, en este caso la de extracción de pareja emoción-causa. Su objetivo es extraer todas las potenciales parejas de emociones junto con sus correspondientes causas dentro de un documento. El artículo propone una aproximación en dos pasos, donde primero se extraen las emociones y causas de forma individual, para luego realizar un proceso de emparejamiento y filtrado entre ellas. El conjunto de datos es el que se utiliza en *Event-Driven Emotion Cause Extraction with Corpus Construction* [28], que contiene casi 2000 documentos con una o más parejas de emoción-causa. Tras distintos experimentos con diversos modelos, observamos como las métricas de precisión rondan el 60 % y 70 %, dependiendo de la variante, lo que demuestra que la estrategia en dos pasos como la propuesta puede que no sea la mejor solución para el problema, ya que los errores producidos en el primer paso afectan inevitablemente al segundo.

Capítulo 4

Planteamiento del problema

4.1. Descripción de los datos

Como comentamos en la introducción del trabajo, el dataset que vamos a utilizar para el análisis de sentimientos va a ser el conjunto de entrenamiento empleado en el proyecto Sentiment140. Dicho proyecto nos proporciona un archivo .csv que podemos descargar desde su página web [29].

El conjunto contiene 1.6 millones de tweets, especificando 6 campos para cada uno:

1. **polarity**: La polaridad del tweet (0: negativo, 2: neutral, 4: positivo).
2. **id**: El identificador del tweet.
3. **date**: La fecha del tweet.
4. **query**: La consulta del tweet. Si no hay consulta, el valor será NO_QUERY.
5. **user**: El usuario que publicó el tweet.
6. **text**: El texto del tweet.

Como vemos, el dataset ya se encuentra etiquetado (lo podemos ver en el campo *polarity*). En este trabajo no entraremos en detalle sobre cómo ha sido el proceso de etiquetado de estos datos, aunque cabe mencionar que se realizó a través de un proceso automático mediante la API de Twitter, utilizando búsquedas por términos.

En el proyecto, con el propósito de reducir los tiempos de procesado y no quedarnos sin memoria, reducimos el conjunto muestreando un 10 % del total, teniendo así un dataset reducido de 160 000 tweets, que serán a su vez repartidos en conjuntos de

entrenamiento, validación y test. A pesar de la reducción de tamaño, sigue siendo una cantidad considerable para realizar nuestras tareas de machine learning.

Además de esto, también realizaremos una segunda tanda de entrenamientos, esta vez empleando un muestreo del 50 % de los datos. De esta manera, podemos comprobar de manera empírica en qué medida influye la cantidad de información que disponemos para trabajar.

4.2. Visualización de los datos

Para tener una idea más clara de los datos con los que vamos a trabajar, utilizamos distintas librerías de Python que nos permiten tener una visualización de los datos, y en concreto de forma orientada al NLP. En esta sección podremos encontrar las figuras que nos aportan esta información junto con el código implementado para obtenerlas. Para simplificar, esta visualización la aplicaremos al conjunto reducido al 10 % de los datos.

En primer lugar, importamos nuestros datos, sampleamos, y mapeamos los valores de y para poder visualizarlo con más claridad.

```
# Importing our data
dirname = path.dirname(__file__)
DATA_DIRECTORY = path.join(dirname,'..','data')
def read_sentiment_file(DATA_DIRECTORY, file):
    csv = pd.read_csv(
        path.join(DATA_DIRECTORY, file),
        names=['polarity', 'id', 'date', 'query', 'user', 'text'],
        encoding='latin-1')
    return csv
sentiment140 = read_sentiment_file(DATA_DIRECTORY, 'sentiment140_train.csv')
# Sampling our data
sentiment140 = sentiment140.sample(frac=0.1, random_state=7)
# Mapping 0: negative and 4: positive, for visualization
sentiment140['polarity'] = sentiment140['polarity'].map({0: 'negative', 4:
↪  'positive'})
```

En la figura 4.1 podemos ver la cantidad de tweets que tenemos en cada una de las dos categorías. Como se puede observar en el gráfico, existe un balance entre ambas, algo que será necesario para hacer funcionar bien nuestro modelo y obtener métricas razonables y fiables.

```
# Countplot of negative and positive polarity of texts
sns.set_palette(['red', 'green', 'blue'])
```

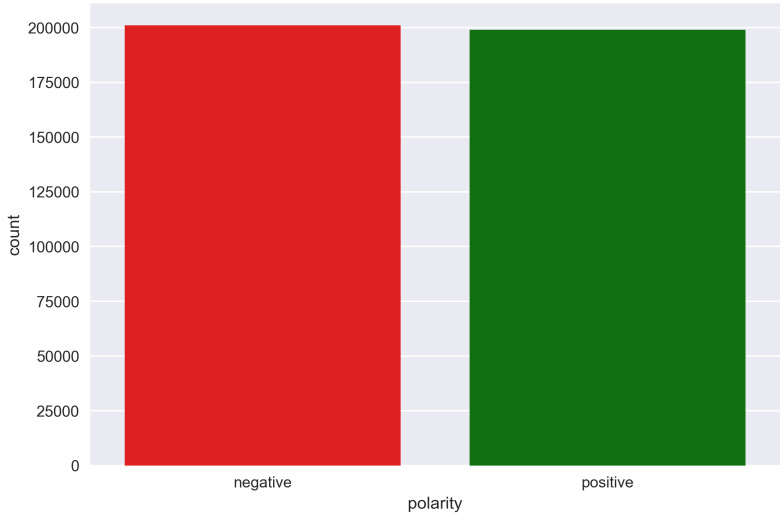


Figura 4.1: Número de tweets en las categorías 'negative' y 'positive'.

```
sns.countplot(x=sentiment140['polarity'])
plt.savefig("../figures/negative_positive.png", dpi=300)
```

Por otra parte, si pasamos a analizar los tweets de manera individual, podemos tener una idea de sus longitudes de caracteres y de palabras. Como vemos en la figura 4.2, el mayor número de datos se trata de textos que rondan los 50 caracteres y una longitud de 10 palabras.

```
# Number of characters and words per sentece
fig, axs = plt.subplots(1, 2, figsize=(14, 7))
axs[0].hist(sentiment140['text'].str.len(), color='skyblue', bins=10)
axs[0].set_title('Characters in each sentence')
axs[0].set_xlabel('Number of characters')
axs[0].set_ylabel('Count')
axs[1].hist(sentiment140['text'].apply(lambda x: len(x.split())),
↪  color='skyblue', bins=10)
axs[1].set_title('Words in each sentence')
axs[1].set_xlabel('Number of words')
axs[1].set_ylabel('Count')
plt.savefig("../figures/characters_words.png", dpi=300);
```

Si entramos en más detalle en el contenido de los textos, podemos visualizar ciertas características mediante paquetes destinados a ello, como NLTK. En la figura 4.3

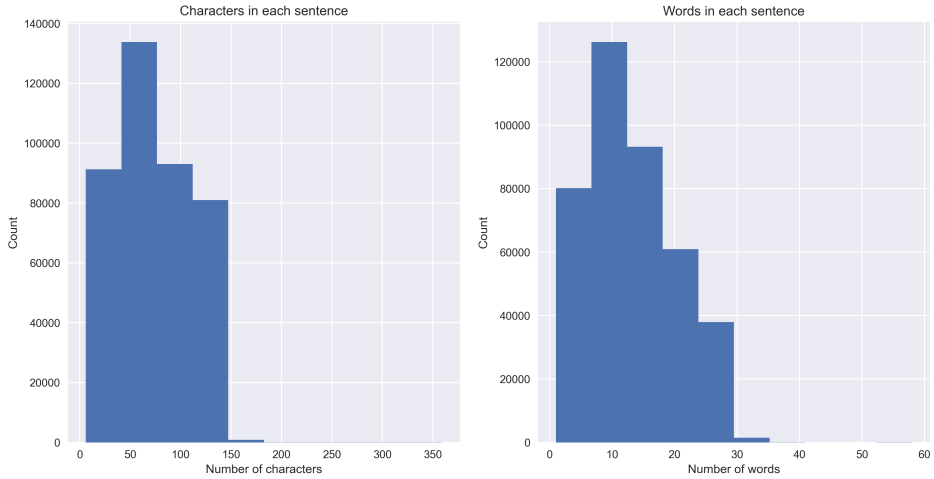


Figura 4.2: Número de tweets en función de sus caracteres y palabras.

podemos observar cuáles son las palabras vacías -stopwords en inglés- más frecuentes. Debemos tener en cuenta este tipo de palabras, ya que se tratan de términos que no aportan significado a una oración, por lo que no serán relevantes a la hora de clasificar nuestros textos. La figura 4.4 también nos muestra cuáles son las palabras más frecuentes de forma general. Para el desarrollo de de esta parte, que incluye la frecuencia de ciertas palabras y bigramas, nos hemos basado en el análisis de *neptuneblog* [30], adaptando el código que aparece a nuestro proyecto.

```
# English stopwords
nltk.download('stopwords')
stop=set(stopwords.words('english'))
# Most common stopwords in texts
corpus = []
words = sentiment140['text'].str.split()
words = words.values.tolist()
corpus = [word for i in words for word in i]
dic = defaultdict(int)
for word in corpus:
    if word in stop:
        dic[word] += 1
top = sorted(dic.items(), key=lambda x:x[1], reverse=True)[:10]
x, y = zip(*top)
plt.bar(x, y, color='skyblue')
plt.savefig("../figures/stopwords.png", dpi=300);
```

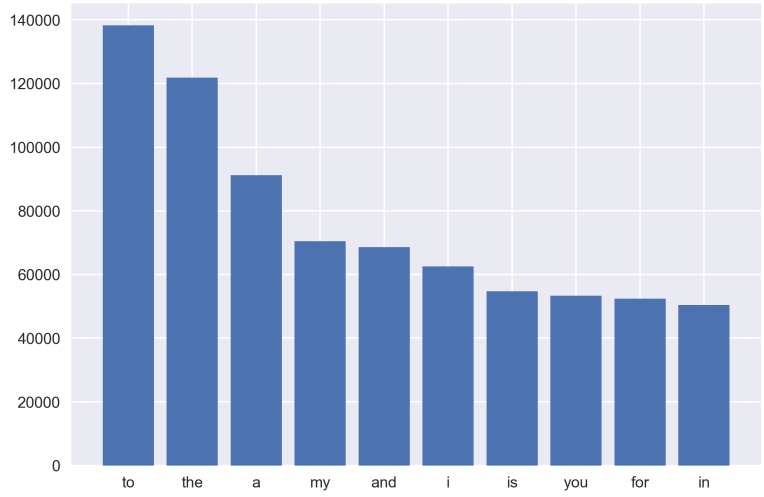


Figura 4.3: Stopwords más frecuentes.

```
# Most common words in texts
counter = Counter(corpus)
most = counter.most_common()
x, y = [], []
for word, count in most[:40]:
    if (word not in stop):
        x.append(word)
        y.append(count)
sns.barplot(x=y, y=x)
plt.savefig("../figures/most_common.png", dpi=300);
```

Usando esta misma herramienta, también podemos observar cuáles son los bigramas (grupos de dos palabras) que más aparecen en nuestros textos. Como se puede ver en la figura 4.5, los primeros puestos están ocupados por conjuntos de preposiciones o conjuntos de preposición y verbo, formando así formas impersonales de los verbos, como gerundios o infinitivos.

```
# Getting top bigrams in texts
corpus = sentiment140['text']
vec = CountVectorizer(ngram_range=(2, 2)).fit(corpus)
bag_of_words = vec.transform(corpus)
sum_words = bag_of_words.sum(axis=0)
words_freq = [(word, sum_words[0, idx])
```

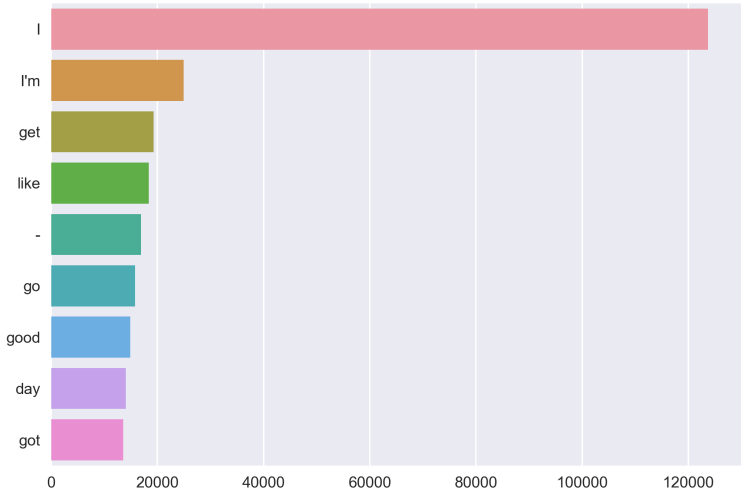


Figura 4.4: Palabras más frecuentes.

```
                for word, idx in vec.vocabulary_.items()]
words_freq = sorted(words_freq, key = lambda x: x[1], reverse=True)
top_10_bigrams = words_freq[:10]
x, y = map(list, zip(*top_10_bigrams))
sns.barplot(x=y, y=x)
plt.savefig("../figures/bigrams.png", dpi=300);
```

En último lugar, haremos uso de Scattertext [31]. Se trata de una herramienta para distinguir términos en textos y mostrarlos en un gráfico de dispersión interactivo. Se utiliza como un paquete de Python que ofrece la posibilidad de crear gráficas muy potentes. En nuestro caso, lo usaremos para comparar cuáles son los términos que más se emplean en función de si dicho texto es positivo o negativo. En la figura 4.6 se puede observar esta distribución. Como hemos podido señalar antes, aquí el balance entre palabras negativas y positivas también es equilibrado.

```
# Parsing texts to scatter plot
nlp = spacy.load("en_core_web_sm")
sentiment140['parsed'] = sentiment140.text.apply(nlp)
sentiment140.to_csv('sentiment140_parsed.csv', index=False)
corpus = st.CorpusFromParsedDocuments(sentiment140, category_col='polarity',
↪  parsed_col='parsed').build()
html = st.produce_scattertext_explorer(corpus,
                                       category='positive',
```

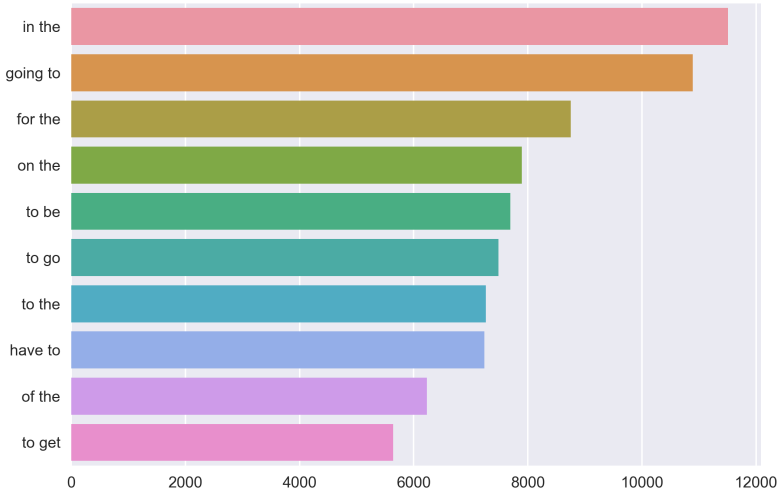


Figura 4.5: Bigramas más frecuentes.

```
                                    category_name='Positive',
                                    not_category_name='Negative',
                                    minimum_term_frequency=5,
                                    width_in_pixels=1000,
                                    transform=
                                    st.Scalers.log_scale_standardize)
file_name = 'ScattertextGraph.html'
open(file_name, 'wb').write(html.encode('utf-8'))
```

4.3. Pre-procesado de los datos

Como comentamos, los datos de partida se obtienen mediante un archivo .csv (del inglés *comma-separated values*). Por lo tanto, necesitamos importarlo a un dataframe para facilitar su manipulación utilizando distintos parámetros. El proceso de importación y muestreo ya lo hemos visto en la sección anterior.

Una vez tenemos nuestro dataset empaquetado en el dataframe y reducido, pasamos a extraer las características (vector X) junto con sus etiquetas (vector y):

```
X = sentiment140['text']
y = sentiment140['polarity']
```

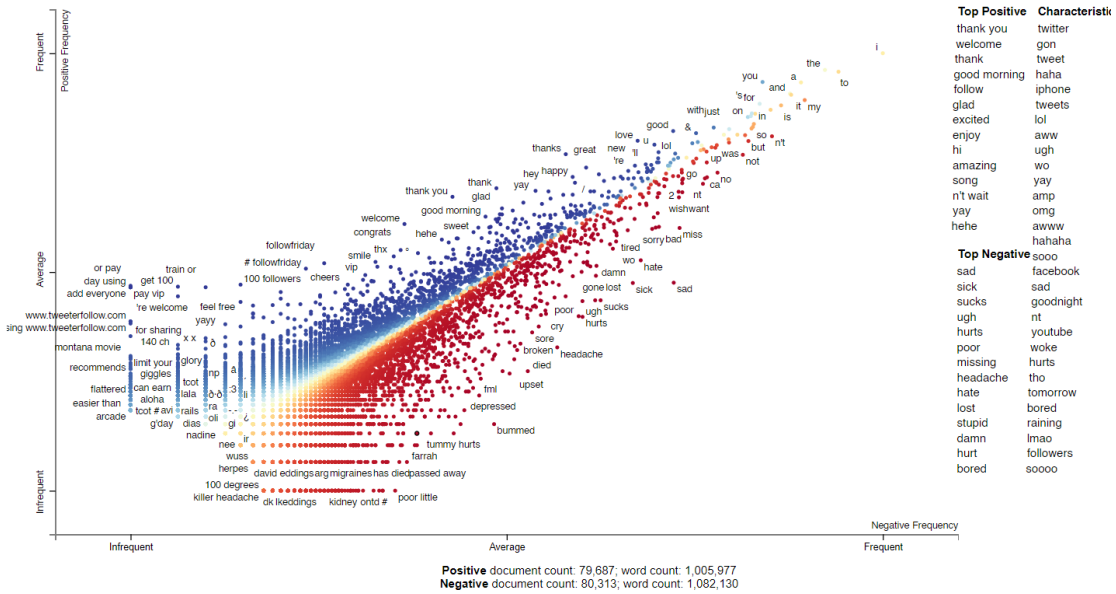


Figura 4.6: Frecuencia de palabras en función del sentimiento que expresan.

Seguidamente, limpiamos los textos mediante la función *clean_ data* y mapeamos los valores de las etiquetas a 0, 1 siendo negativo o positivo, respectivamente. Omitimos las etiquetas neutras, ya que a pesar de estar especificado en las características del dataset, no aparece ningún caso neutro.

La función que limpia los datos se encarga principalmente de pasar los textos a minúscula, eliminar las URL y eliminar los nombres de usuario (que empiezan con @). El mapeo será distinto al caso anterior, ya que nos interesa tener valores numéricos para realizar la clasificación binaria.

```
def clean_data(data):
    data = data.str.lower()
    data = data.apply(lambda x: re.sub(r'http\S+', '', x))
    data = data.apply(lambda x: re.sub(r'@\S+', '', x))
    data = data.apply(lambda x: re.sub('[^a-zA-Z]', ' ', x))
    return data
X = clean_data(X)
y = y.map({0: int(0), 4: int(1)})
```

El siguiente paso es crear nuestros conjuntos de validación y test, con la finalidad de comprobar si el modelo que vamos a entrenar generaliza bien para el resto de datos. La validación la iremos realizando a medida que se entrena, en cada iteración, mientras que el test lo haremos al final, cuando nuestro modelo ya está entrenado. Este sistema evita que

se pueda particularizar un modelo para el conjunto de validación (sobre-entrenamiento). Lo hacemos mediante la función *train_test_split* que proporciona Scikit-learn.

```
# Split the data to train and test, then train and val
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
↪  random_state=7)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
↪  test_size=0.2, random_state=7)
```

A continuación, debemos *tokenizar* los datos. El proceso de tokenización consiste en asignar un índice a cada palabra, siendo las palabras que más aparecen las que tendrán índices más bajos. De esta forma, creamos un diccionario (*word_index*) que nos relaciona cada palabra con un entero. Para elegir qué palabras pasamos por este proceso nos basamos en el conjunto de entrenamiento. Este proceso de indexación es un paso previo que necesitamos para la representación de los datos, que veremos en detalle en la siguiente sección.

```
def tokenize(X_train, X_val, X_test):
    tk = Tokenizer()
    # Update on the vocabulary based on the training set
    tk.fit_on_texts(X_train)
    # word_index is a dict that contains words (key) and index (value), sorted
    ↪  by frequency
    word_index = tk.word_index
    vocab_size = len(word_index) + 1
    return (tk, word_index, vocab_size)
tk, word_index, vocab_size = tokenize(X_train, X_val, X_test)
X_train = tk.texts_to_sequences(X_train)
X_val = tk.texts_to_sequences(X_val)
X_test = tk.texts_to_sequences(X_test)
```

4.4. Representación de los datos

El último paso antes de alimentar la red neuronal que se encargará de la tarea de clasificación es el de representar los datos de forma que la red pueda *entenderlos*. Estudiaremos 3 formas de trabajar esta información para obtener unos mejores resultados.

4.4.1. GloVe

Antes de hablar de GloVe, necesitamos introducir el concepto de *word embedding*. Se trata de una técnica NLP donde cada una de las palabras se representa como un vector de

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Figura 4.7: Probabilidades para determinadas palabras.

números reales. De esta forma, pasamos de un modelo en el que cada palabra de nuestros textos ocupa una dimensión, a otro donde las palabras pertenecen a un espacio vectorial continuo, donde el número de dimensiones es finito.

Esta técnica nos permite entonces tener unas dimensiones del problema definidas y delimitadas, pudiendo abordar el problema. Una oración con K palabras, donde cada palabra tiene N dimensiones pasa a ser un conjunto de datos de 2 dimensiones ($K * N$), pudiendo trabajar con redes neuronales convolucionales u otros métodos.

GloVe (Global Vectors for Word Representation) [32] se trata de un algoritmo de aprendizaje no supervisado cuyo objetivo es obtener representaciones vectoriales de palabras en un texto. Decidimos usar este modelo ya que ha sido entrenado con un conjunto de datos muy amplio y presenta características que facilitan la tarea de procesar el lenguaje natural.

Su entrenamiento se realiza teniendo en cuenta las estadísticas agregadas de co-ocurrencia de palabras en textos, y las representaciones resultantes ofrecen interesantes subestructuras lineales del espacio vectorial.

Cuando hablamos de co-ocurrencia, nos referimos a las relaciones que podemos encontrar en palabras que tienen significados similares, y por lo tanto, es más probable que aparezcan juntas en un texto. Para verlo de forma más clara, en la siguiente tabla de la figura 4.7 se pueden observar las probabilidades tras realizar una recopilación de 6000 millones de palabras.

Como podemos observar, el algoritmo se ha diseñado teniendo en cuenta que palabras como *ice* -hielo, en inglés- tienen más posibilidades de aparecer junto con *water* -agua- que junto con *fashion*.

Otra de las características que ofrece el algoritmo es la llamada *nearest neigbors*. La distancia euclídea es una métrica que se puede utilizar para medir la similitud entre palabras dentro del algoritmo, de forma que aquellas palabras que están más relacionadas entre sí son aquellas que están más próximas entre ellas.

Sin embargo, esta medida, a priori simple y eficiente, no nos sirve completamente para medir esta similitud que buscamos. Principalmente se debe a que una palabra puede estar relacionada con otra de varias formas, dependiendo del sentido, y para ilustrar

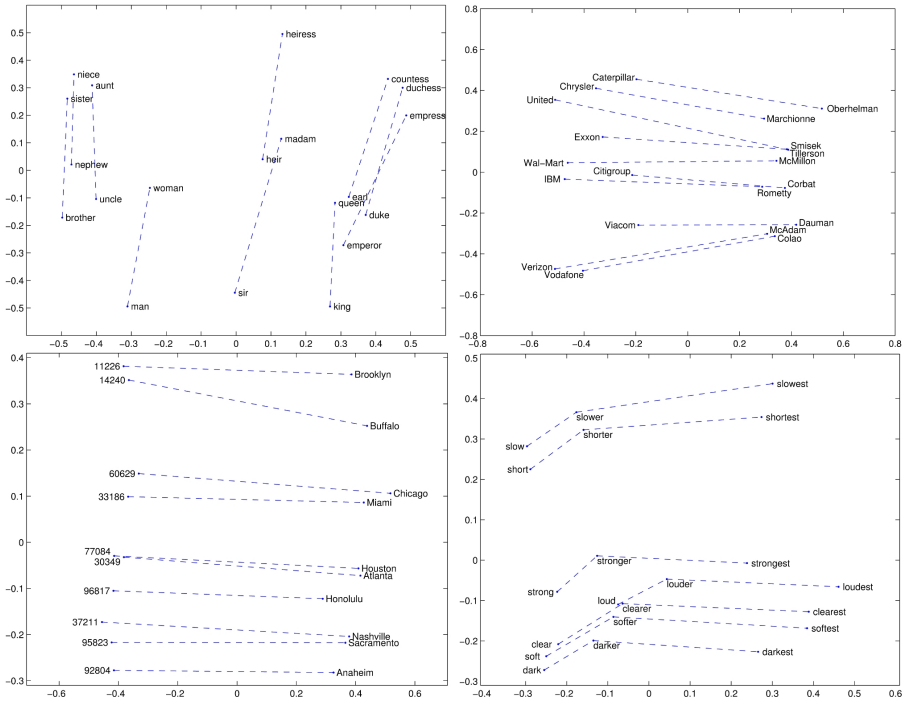


Figura 4.8: Representaciones gráficas de palabras similares.

esto, un único escalar es insuficiente. Por ello, para poder capturar estas relaciones sin reducirlos de tal forma, buscamos una similitud entre palabras a partir de la diferencia de sus vectores. Mediante estas *subestructuras lineales*, podemos conseguir capturar estas similitudes entre parejas de palabras, tal como vemos en la figura 4.8.

A la hora de implementarlo en nuestro modelo, el primer paso es aplicar un relleno a cada uno de los textos, de forma que todos tengan la misma longitud.

```
def longest_sentence(X_train, X_val, X_test):
    return max(
        len(sorted(X_train, key=len, reverse=True)[0]),
        len(sorted(X_val, key=len, reverse=True)[0]),
        len(sorted(X_test, key=len, reverse=True)[0]))
maxlen = longest_sentence(X_train, X_val, X_test)
X_train = pad_sequences(X_train, padding='post', maxlen=maxlen)
X_val = pad_sequences(X_val, padding='post', maxlen=maxlen)
X_test = pad_sequences(X_test, padding='post', maxlen=maxlen)
```

Hecho esto, necesitamos crear nuestra *embedding matrix*. Dicha matriz contiene todas las palabras que se encuentran en nuestro texto junto con la representación vectorial de GloVe (si está definida para dicha palabra). Para poder indexar cada palabra en una fila de la matriz era necesario el proceso de tokenización descrito anteriormente.

Sus filas serán el número de palabras únicas en el texto, mientras que sus columnas se corresponden con la dimensión de embedding. Este último parámetro será una elección de diseño, ya que podemos elegir entre las 50, 100, 200 y 300 dimensiones que ofrece GloVe, buscando un equilibrio entre rapidez de procesado y mejores resultados. Para nuestras pruebas utilizaremos el caso de 50 y 300 dimensiones. Cuantas más dimensiones, más datos con los que trabajar y más posibilidades de mejora tendremos.

```
EMBEDDING_DIM = 300
def create_embedding_matrix(vocab_size, EMBEDDING_DIM, word_index):
    EMBEDDING_DIM = EMBEDDING_DIM
    # embedding_index will be the dict containing words in word_index, but
    ↪  GloVe coefficients
    embedding_index = {}
    # embedding_matrix dimensions: vocab_size x embedding_dim
    embedding_matrix = np.zeros([vocab_size, EMBEDDING_DIM])
    if (EMBEDDING_DIM == 50):
        f = open(path.join(DATA_DIRECTORY, 'GloVe', 'glove.6B.50d.txt'),
        ↪  encoding='utf-8')
    else:
        f = open(path.join(DATA_DIRECTORY, 'GloVe', 'glove.6B.300d.txt'),
        ↪  encoding='utf-8')
    # Format is: word..value..value.. ..value. We split to make a list, and
    ↪  loading in embedding_index
    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embedding_index[word] = coefs
    f.close()
    # We iterate on word_index to fill the embedding_matrix with words from
    ↪  word_index
    # and values from embedding_index. If there is no value, it is filled with
    ↪  zeros
    for word, i in word_index.items():
        embedding_vector = embedding_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
    return embedding_matrix
embedding_matrix = create_embedding_matrix(vocab_size, EMBEDDING_DIM,
↪  word_index)
```

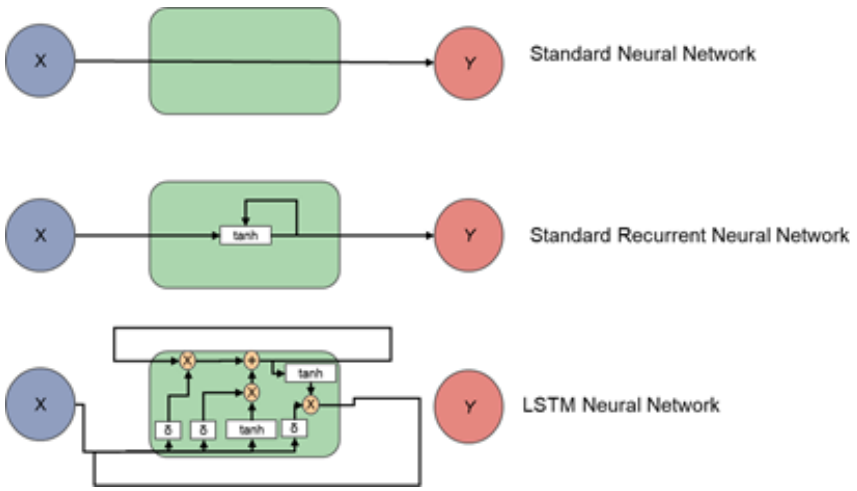


Figura 4.9: Comparativa entre ANN, RNN y LSTM [7]

4.4.2. ELMo

ELMo [33] se trata de un framework de word embedding que, a diferencia de GloVe, su objetivo es proporcionar *word embeddings contextualizados*, tratando así de fijarse en el significado que aporta cada palabra dentro de la oración. Esto de debe a que determinadas palabras, dependiendo del contexto, pueden tener significados diferente, necesitando por lo tanto alguna forma de obtener vectores diferentes.

Antes de entrar en detalle en el funcionamiento de ELMo, debemos explicar el concepto de LSTM, para lo que necesitamos también conocer de qué se trata una red neuronal recurrente (RNN) [34]. La característica principal de estas últimas es que permiten que, a diferencia de las redes neuronales convencionales, la información anterior persista introduciendo bucles, pudiendo así recordar estados previos y utilizar esta información para influir en la siguiente. La contra de este tipo de redes es que estas dependencias se limitan al corto plazo. Es aquí donde aparecen las redes LSTM, cuyo funcionamiento evita este problema. Aún sin entrar en mucho detalle, en la figura 4.9 podemos ver su arquitectura en comparación a las otras redes, consiguiendo tener en cuenta la información anterior tanto a corto como a largo plazo.

En lugar de utilizar un embedding fijo para cada palabra, haciendo un mapeado para cada palabra, ELMo utiliza un modelo de lenguaje pre-entranado. Utiliza una red LSTM bidireccional de múltiples capas (normalmente 2) entrenada para la tarea específica de crear estos embeddings. Su estructura sería tal como muestra la figura 4.10.

El modelo parte de los datos de entrada, a los que se aplica una red neuronal convolucional a nivel de caracter para obtener una representación vectorial primaria, que es de lo que se alimenta la primera capa. Realizar este embedding a nivel de caracteres supone

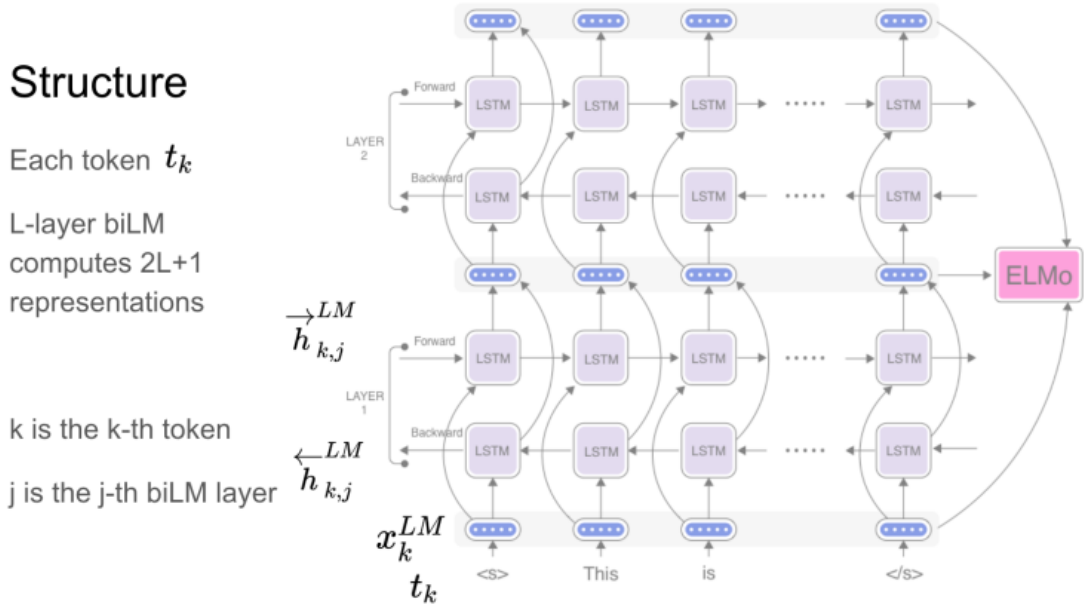


Figura 4.10: Estructura del funcionamiento de ELMo [8]

varias ventajas. Principalmente, permite capturar la estructura interna de cada palabra: sabemos que *arte* y *artista* estarán relacionados de alguna manera. Además, es robusto a palabras desconocidas, que no han aparecido durante el entrenamiento del modelo.

Como vemos en el esquema, cada capa recorre los datos de dos maneras: hacia delante y hacia atrás, en ambas direcciones. Las pasadas hacia delante tienen en cuenta las palabras anteriores de la oración, mientras la pasada hacia atrás tiene en cuenta las palabras futuras. Estos estados intermedios se concatenan y producen un vector intermedio para cada palabra. De esta forma tenemos una representación vectorial para cada palabra que de alguna forma *sabe* cuál es el contexto anterior como futuro de la oración.

Otra característica del modelo es que utiliza modelos de lenguaje de capas múltiples. De esta forma, el vector intermedio resultante mencionado sirve de entrada para la siguiente capa, que hace la misma operación. A mayor procesado, conseguimos representar una semántica más abstracta (llegando a temas o sentimientos), en comparación a capas inferiores donde podemos capturar características menos complejas.

Finalmente, la representación que obtenemos se realiza mediante una combinación ponderada de los L vectores intermedios + 1, siendo L el número de capas al que le sumamos los vectores de palabras en bruto que utilizamos en la entrada del sistema. [8]

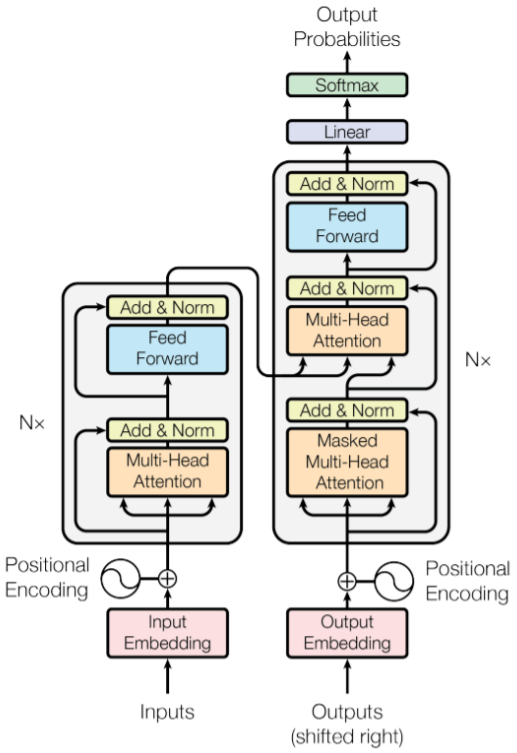


Figura 4.11: Arquitectura de un transformer [9]

4.4.3. BERT

BERT (Bidirectional Encoder Representations from Transformers) [35] se trata de otro modelo para representar los datos, de manera que puedan ser alimentados a la red neuronal y ser clasificados con una mayor precisión. Al igual que ELMo, se trata de un sistema que tiene en cuenta el contexto de las palabras [36], pero en vez de hacerlo mediante el uso de redes LSTM, este se basa en el entrenamiento bidireccional de un Transformador, un modelo de atención conocido que aprende relaciones contextuales entre palabras (o sub-palabras) en un texto. La ventaja que aporta es que en lugar de tomar una entrada en cada momento, estas entradas pueden ser paralelizadas, lo que resulta en un mejor rendimiento.

En principio, un transformador incluye dos mecanismos diferenciados: un codificador que lee el texto de entrada y un decodificador que produce una predicción para la tarea. Dado que la tarea de BERT es únicamente modelar el lenguaje de los textos, únicamente será necesario el mecanismo de codificador. En la figura 4.11 podemos ver su arquitectura.

A diferencia de modelos direccionales, que leen el texto de entrada de forma secuencial, el codificador del transformador lee la secuencia de palabras completa a la vez.

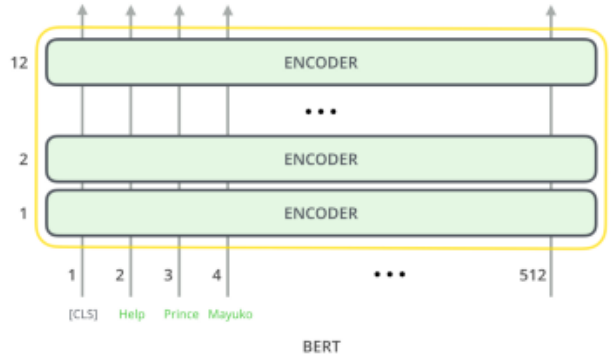


Figura 4.12: Arquitectura de BERT [10]

Esta característica permite al modelo aprender el contexto de una palabra basadándose en aquello que la rodea (derecha e izquierda). Mediante este procesado, el modelo nos permite extraer las características de los textos, facilitando las tareas de clasificación. Si observamos la figura 4.12, muestra su arquitectura desde una perspectiva general.

En concreto, BERT tiene 2 variantes: BERT base, que cuenta con 12 codificadores, y BERT large, con 24 codificadores. Ambos fueron entrenados a partir de datos sin etiquetar extraídos de BooksCorpus con 800 millones de palabras, y la Wikipedia en inglés, con 2500 millones de palabras. En nuestro proyecto trabajaremos con la variante BERT base. Como ya se ha pre-entrenado, solo es necesario afinar el modelo tal como veremos en el siguiente capítulo.

Capítulo 5

Desarrollo del problema

En la solución que planteamos para resolver el problema de clasificación de textos, estudiaremos 3 modelos, que se detallan en las siguientes secciones. Como comentamos, el primero de ellos cuenta con dos variantes, por lo que serán 4 casos a tener en cuenta. Los resultados que obtenemos de cada uno de ellos se explican en el siguiente capítulo.

5.1. CNN+GloVe

El primer modelo es el que está basado en redes neuronales convolucionales (CNN) junto con la representación GloVe de 50 y 300 dimensiones. Como explicamos en los primeros capítulos, los estudios han mostrado que es una aproximación que ofrece buenos resultados para este tipo de tareas.

Centrándonos en el proyecto, la arquitectura del modelo tiene el aspecto que se puede observar en la figura 5.1. Mencionar que la herramienta que ha sido utilizada para la visualización gráfica de la red neuronal ha sido Net2Vis [37]. El funcionamiento es sencillo; introduciendo el código de Keras que hemos utilizado para crear nuestra red podemos descargar vía .pdf o .svg una visualización intuitiva de las capas.

Para simplificar, la arquitectura que se muestra es la que empleamos en nuestro dataset reducido al 10 %. En este subconjunto, la longitud máxima de las oraciones es de 50, por lo que serán nuestras filas de entrada. En el caso de que el subconjunto sea mayor, esta longitud máxima puede variar, por lo que cambiarían las filas de entrada, pero la estructura sería la misma.

En este caso, comenzamos con una capa de *Embedding*, que hace uso de los pesos que hemos definido siguiendo GloVe, tal como describimos en el capítulo anterior. Las dimensiones de entrada para cada texto serán de 50 filas (el número máximo de palabras

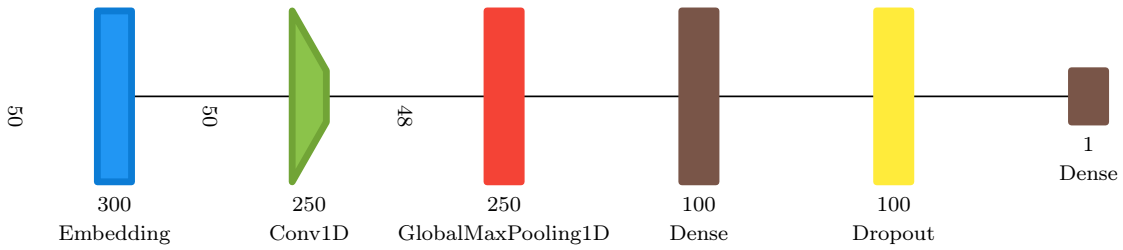


Figura 5.1: Arquitectura del modelo CNN+GloVe.

en cada tweet tras el pre-procesado) y 50 o 300 columnas (dimensiones obtenidas mediante word embedding), dependiendo de nuestra elección. Destacar que los textos entran a la red en lotes de 64 para un procesado más rápido.

Seguidamente, los textos pasan a la capa de convolución de una dimensión, aplicando convoluciones por filas, lo que hace quedarnos con 250 columnas, debido a la decisión de utilizar 250 filtros, que se irán ajustando conforme al entrenamiento. A la salida pasamos a tener 48 filas, perdiendo dos debido a el padding de los bordes.

Hecho esto, esta información pasa a la capa de *Global Max Pooling* de una dimensión, donde reducimos la la cantidad de datos, quedándonos con el valor máximo de la fila, aquel que representa un peso mayor. Esta capa también sirve para pasar a una única dimensión nuestros textos, aplanándolos y servir de entrada a la siguiente capa.

La siguiente capa es una capa densa, la entrada de la red neuronal artificial que realiza la tarea de clasificación de datos en función de lo conseguido en capas anteriores. Utilizaremos 100 neuronas, cuyos pesos y sesgos irán variando en función del entrenamiento. A esto le juntamos una capa de *Dropout*, cuya función se basa en deshabilitar las neuronas con una probabilidad dada (en nuestro caso 0.5), con el objetivo de evitar el *overfitting*. El *overfitting* -sobreajuste, en español- se trata del efecto que se produce cuando nuestra red se comporta tan bien para el conjunto de entrenamiento, que llega a no generalizar bien para el resto: aprende las características más concretas del conjunto de entrenamiento y pierde la capacidad de predecir datos diferentes a éstos.

Finalmente, terminamos la red con una única neurona cuya función de activación es la llamada *sigmoid*, cuyo objetivo es resultar en un número entre 0 y 1, que determinará la probabilidad de ser positivo.

Desde el punto de vista de código, la construcción del código se detalla a continuación:

```
# Architecture for CNN + GloVe model
model = Sequential()
model.add(Embedding(
    input_dim=vocab_size,
    output_dim=EMBEDDING_DIM,
```

```
    weights=[embedding_matrix],
    input_length=maxlen,
    trainable=False))
model.add(Conv1D(
    filters=250,
    kernel_size=3,
    activation='relu'))
model.add(GlobalMaxPooling1D())
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.compile(
    optimizer='sgd',
    loss='binary_crossentropy',
    metrics=['accuracy', metrics.Precision(name='precision'),
    ↪  metrics.Recall(name='recall')])
model.summary()
```

En lo que se refiere al entrenamiento de la red, definimos un callback del tipo *early stopping*, como segunda medida para el evitar el mencionado *overfitting*, deteniendo el entrenamiento cuando las pérdidas en el conjunto de validación no mejoren después de 15 *epochs*.

El código utilizado es el siguiente:

```
# Training and validation for CNN + GloVe model
early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=15,
    restore_best_weights=True)
history = model.fit(
    X_train,
    y_train,
    batch_size=64,
    epochs=100,
    validation_data=(X_val, y_val),
    callbacks=[early_stopping])
```

5.2. Utilización de ELMo

Para la implementación de este modelo de representación de datos, utilizamos el repositorio de GitHub keras_elmo_bert, del usuario epoch8. Mediante sus dos scripts *elmo_tokenizer* y *elmo_layer*, nos aporta un envoltorio del modelo para ser utilizado

como una capa de Keras, la librería que hemos utilizado a lo largo de este proyecto. A su vez, el repositorio está basado en el modelo de ELMo de tensorflow-hub, proporcionado por Google.

Decidimos utilizar este repositorio ya que el uso de Tensorflow de forma directa supone una curva de aprendizaje demasiado pronunciada, dado que todo el trabajo se ha basado en su implementación de Keras.

Antes de crear el propio modelo, necesitamos transformar los textos a un formato que sea entendible, al igual que hacíamos en GloVe. En este caso, lo realizamos mediante el tokenizador de ELMo, especificando una longitud máxima para poder aplicar padding a aquellos textos que tengan una menor longitud. Para saber cuál es la máxima longitud de nuestras oraciones lo podemos hacer mediante el método *max()* integrado en Python, tal como se muestra a continuación:

```
from elmo_tokenizer import ELMO_tokenizer
max_seq_length = max(X.str.len())
tokenizer = ELMO_tokenizer(max_seq_length)
train_tokens = tokenizer.predict(X_train.tolist())
train_label = y_train.tolist()
val_tokens = tokenizer.predict(X_val.tolist())
val_label = y_val.tolist()
test_tokens = tokenizer.predict(X_test.tolist())
test_label = y_test.tolist()
```

Una vez tenemos los datos en el formato correcto, necesitamos definir la arquitectura del modelo, tal como puede observar en la figura 5.2.

Como vemos, aunque la estructura parece más sencilla que el modelo anterior, esto se debe a que en este caso la capa *ElmoLayer* contiene todas las operaciones que hemos detallado en el capítulo anterior (embedding a nivel de carácter, procesado de LSTM, etc).

De esta forma, contamos con una capa de Input, que toma cada una de las oraciones con una dimensión máxima de 138 palabras. Esta dimensión de entrada es distinta a la del modelo de CNN+GloVe dado que el proceso de tokenización es distinto.

Cada oración pasa a la capa de ELMo, donde se van creando los vectores intermedios que finalmente se ponderan para dar lugar a un vector final que reúne las características de cada palabra, teniendo en cuenta el contexto en la oración. Este vector tiene una dimensión de 1024.

Tendríamos por lo tanto un vector de dimensiones (138, 1024) para cada oración. Sin embargo, entre las opciones de salida que ofrece la implementación, en nuestro caso utilizamos la salida *default*. Esta opción lo que hace es una media de todas las palabras

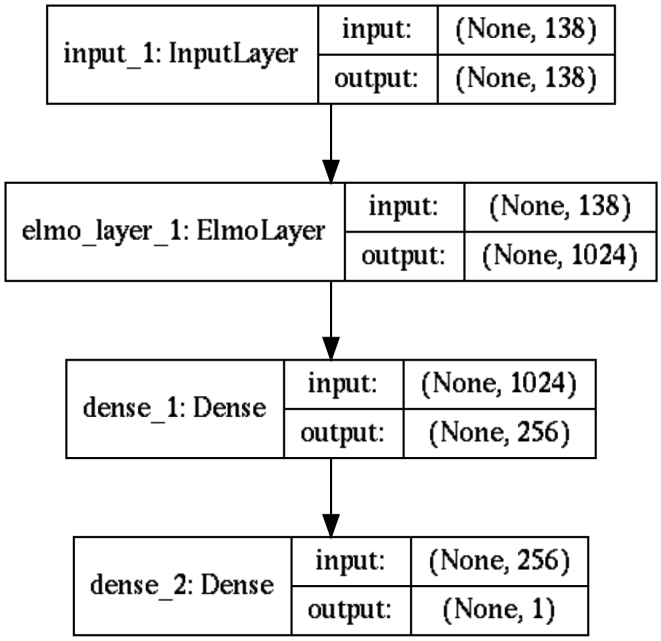


Figura 5.2: Arquitectura del modelo ELMo

de la oración contextualizadas, pasando así a una dimensión (1024) para cada oración, habiendo aplanado nuestros textos.

Finalmente, cada oración reducida a 1024 dimensiones pasa a una red neuronal artificial con una única capa de 256 neuronas, cuya tarea es realizar la clasificación de los textos en función a las oraciones procesadas por ELMo. Contamos con una neurona de salida con la capa de activación *sigmoid*, que se encarga de dar la predicción de las probabilidades para la clasificación binaria.

A nivel de código nos basamos en la implementación de *elmo_ layer* y la incorporamos como una capa de Keras:

```
from elmo_layer import ElmoLayer
def build_model_elmo(max_seq_length):
    input_tokens = Input(shape=(max_seq_length,), dtype="string")
    elmo_output = ElmoLayer(trainable=True, tf_hub=elmo_path,
    ↪  output_representation='default')(input_tokens)
    dense = Dense(256, activation='relu')(elmo_output)
    pred = Dense(1, activation='sigmoid')(dense)
    model = Model(inputs=input_tokens, outputs=pred)
    model.compile(
        loss='binary_crossentropy',
        optimizer='adam',
```

```
        metrics=['accuracy', metrics.Precision(name='precision'),
        ↪  metrics.Recall(name='recall')])
    model.summary()
    return model
elmo_model = build_model_elmo(max_seq_length)
```

Al igual que en el modelo de CNN, utilizaremos *early stopping* en el proceso de entrenamiento para evitar el overfitting. También fijamos el tamaño de lote a 32, inferior que en el modelo anterior para reducir el consumo de RAM. Destacar el gran consumo de recursos que requiere este modelo, contando con 93 863 509 parámetros, de los cuales 262 661 son entrenables para el dataset reducido al 10 %.

```
early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True)
history_elmo = elmo_model.fit(
    train_tokens,
    train_label,
    validation_data=(val_tokens, val_label),
    epochs=20,
    batch_size=32,
    callbacks=[early_stopping])
```

5.3. Utilización de BERT

Al igual que en ELMo, para la implementación de este modelo hacemos uso del repositorio de GitHub antes mencionado (keras_elmo_bert) debido a su fácil integración con Keras.

Como en el resto de sistemas, primero necesitamos tokenizar nuestros datos, de forma que podamos pasarlo a la capa de entrada. En este caso, transformamos nuestras características y etiquetas, y las pasamos a tokens en función de la longitud máxima de las oraciones, aplicando padding.

Lo que diferencia a BERT es que el resultado de este proceso son 4 conjuntos de datos. En los anteriores casos únicamente teníamos nuestras caracterísitcas (tokens) y etiquetas (labels).

Explicamos cada uno para el caso de entrenamiento, pero se realiza de forma análoga para el conjunto de test:

- **train_input_ids**: Contiene los identificadores de cada palabra, en función de los

índices asignados. Si la oración no llega a la longitud máxima definida, se rellena con ceros.

- **train_input_masks**: Contiene una máscara que identifica qué la cantidad de información que contiene cada entrada. Si existe información en una posición, aparece un 1, y un 0 en caso contrario.
- **train_segment_ids**: Contiene el identificador a nivel de oración dentro de nuestro texto. Como hemos eliminado la puntuación, en nuestro caso siempre será cero, ya que contamos con una sola oración.
- **train_labels**: Contiene las etiquetas del sentimiento positivo o negativo que expresa el texto.

```
from bert_tokenizer import BERT_tokenizer
# Create datasets (Only take up to max_seq_length words for memory)
train_text = X_train.tolist()
train_text = [' '.join(t.split()[0:max_seq_length]) for t in train_text]
train_text = np.array(train_text, dtype=object)[:, np.newaxis]
train_label = y_train.tolist()
val_text = X_val.tolist()
val_text = [' '.join(t.split()[0:max_seq_length]) for t in val_text]
val_text = np.array(val_text, dtype=object)[:, np.newaxis]
val_label = y_val.tolist()
test_text = X_test.tolist()
test_text = [' '.join(t.split()[0:max_seq_length]) for t in test_text]
test_text = np.array(test_text, dtype=object)[:, np.newaxis]
test_label = y_test.tolist()
tokenizer = BERT_tokenizer(bert_path, max_seq_length)
(train_input_ids, train_input_masks, train_segment_ids, train_labels) =
↪  tokenizer.predict(train_text, train_label)
(val_input_ids, val_input_masks, val_segment_ids, val_labels) =
↪  tokenizer.predict(val_text, val_label)
(test_input_ids, test_input_masks, test_segment_ids, test_labels) =
↪  tokenizer.predict(test_text, test_label)
```

Una vez tenemos nuestros conjuntos de datos, necesitamos definir las capas del modelo. La arquitectura se puede observar en la figura 5.3.

Como vemos, tomamos tres fuentes de entrada, que ya hemos definido previamente: *input_ids*, *input_masks* y *segment_ids*.

Estas tres fuentes son la entrada de la capa de BERT, en la cual definimos que el número de capas de ajuste será 2. Como comentamos, BERT se trata de un modelo preentrenado en el que solo hay que realizar el llamado *fine-tuning*. Esto quiere decir que

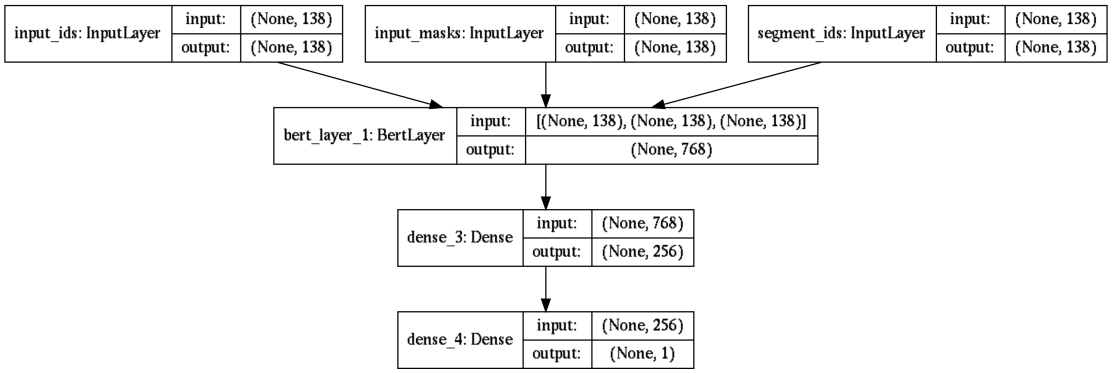


Figura 5.3: Arquitectura del modelo BERT

de las 12 capas de codificación que tiene, utilizaremos únicamente las dos últimas para ajustar sus pesos.

Al igual que en el caso de ELMo, dentro de las opciones que podemos elegir para la representación de la salida, elegimos la *mean_ pooled*, lo que quiere decir que una vez tenemos el embedding de cada una de las palabras de la oración, hacemos una representación media de toda la oración teniendo su contenido, aplanando así la salida con una dimensión de (768).

Esta salida aplanada nos sirve como entrada para la red neuronal artificial de 256 neuronas que hace la función de clasificación, con una neurona de salida que predice la probabilidad positiva.

```
from bert_layer import BertLayer
def build_model_bert(max_seq_length):
    in_id = Input(shape=(max_seq_length,), name="input_ids")
    in_mask = Input(shape=(max_seq_length,), name="input_masks")
    in_segment = Input(shape=(max_seq_length,), name="segment_ids")
    bert_inputs = [in_id, in_mask, in_segment]
    bert_output = BertLayer(n_fine_tune_layers=2, tf_hub=bert_path,
    ↪  output_representation='mean_pooling', trainable=True)(bert_inputs)
    dense = Dense(256, activation='relu')(bert_output)
    pred = Dense(1, activation='sigmoid')(dense)
    model = Model(inputs=bert_inputs, outputs=pred)
    model.compile(
        loss='binary_crossentropy',
        optimizer='adam',
        metrics=['accuracy', metrics.Precision(name='precision'),
        ↪  metrics.Recall(name='recall')])
    model.summary()
    return model
```

```
bert_model = build_model_bert(max_seq_length)
```

Finalmente, el entrenamiento lo realizamos de forma similar a los casos anteriores, fijando un *early_ stopping* y otros parámetros como el tamaño de lote (32) y los conjuntos de entrenamiento y validación. Al igual que en ELMo, se trata de un algoritmo muy costoso computacionalmente, con 110 103 354 parámetros, de los cuales 198 657 entrenables, para el conjunto del 10 % de los datos.

```
early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True)
history_bert = bert_model.fit(
    [train_input_ids, train_input_masks, train_segment_ids],
    train_labels,
    validation_data=([val_input_ids, val_input_masks, val_segment_ids],
    ↪  val_labels),
    epochs=20,
    batch_size=32,
    callbacks=[early_stopping])
```

Capítulo 6

Presentación de resultados

6.1. Métricas empleadas

Antes de evaluar los resultados, necesitamos definir qué métricas definimos para evaluar el rendimiento de los modelos [38].

6.1.1. Loss

La métrica más importante que estudiaremos -en la que nos basamos para ajustar nuestro algoritmo cada iteración- es la pérdida -loss, en inglés-. En nuestro problema, entrenamos nuestro algoritmo para que clasifique los textos en positivos o negativos. Esto quiere decir que para cada muestra, otorga una probabilidad -entre 0 y 1- de pertenecer a la categoría positiva.

El objetivo es encontrar una función que mida cómo de buena o mala es la predicción de la probabilidad, en función de su etiqueta. Esta función es la llamada *función de pérdidas*: devuelve valores altos para malas predicciones y valores bajos para buenas predicciones.

Típicamente, para una clasificación binaria como la de nuestro trabajo, la función de pérdidas usada es la *binary cross-entropy loss*. Analíticamente:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i)) \tag{6.1}$$

Siendo y la etiqueta (0: negativo, 1: positivo), y $p(y)$ la probabilidad predicha de expresar un sentimiento positivo para cada una de las N muestras.

Por lo tanto, en la fórmula vemos que, para cada muestra positiva ($y = 1$), se va

restando a H_p el logaritmo de la probabilidad de ser positivo. De la misma manera, para cada muestra negativa ($y = 0$), se substrae a nuestro resultado el logaritmo de la probabilidad de ser negativo.

6.1.2. Accuracy

La exactitud -accuracy, en inglés- es el cociente de predicciones correctas entre el número de predicciones totales.

$$Accuracy = \frac{N\ predicciones\ correctas}{N\ predicciones\ totales} \tag{6.2}$$

Si bien es cierto que es una métrica importante, no será el único indicador que tengamos en cuenta, ya que no nos aporta una visión global del comportamiento del modelo. Por ejemplo, si nuestro conjunto estuviera formado por 99 muestras positivas y 1 negativa, un modelo que siempre predijera que el resultado es positivo obtendría una precisión del 99 %, algo que no sería realista, ya que el modelo no se estaría ajustando a los datos realmente. Para ello, también introducimos las métricas que detallamos a continuación, que se obtienen a partir de la matriz de confusión.

6.1.3. Confusion matrix

Antes de describir las siguientes métricas, necesitamos introducir el concepto de matriz de confusión. La matriz de confusión es aquella que nos relaciona las muestras positivas y negativas con las predicciones positivas y negativas, tal como muestra la figura 6.1. Las columnas muestran la cantidad de valores (positivos o negativos) predichos en función de las filas, siendo las filas los valores reales (positivos o negativos). Distinguimos entonces:

- **Verdaderos positivos (VP)**: El valor real y su predicción son positivos.
- **Verdaderos negativos (VN)**: El valor real y su predicción son negativos.
- **Falsos positivos (FP)**: El valor real es negativo y su predicción es positiva.
- **Falsos negativos (FN)**: El valor real es positivo y su predicción es negativa.

6.1.4. Precisión

La precisión -precision, en inglés- se trata de la proporción de textos positivos ($y = 1$) entre aquellos que he clasificado como positivos ($\hat{y} = 1$). Trasladando esto la matriz de

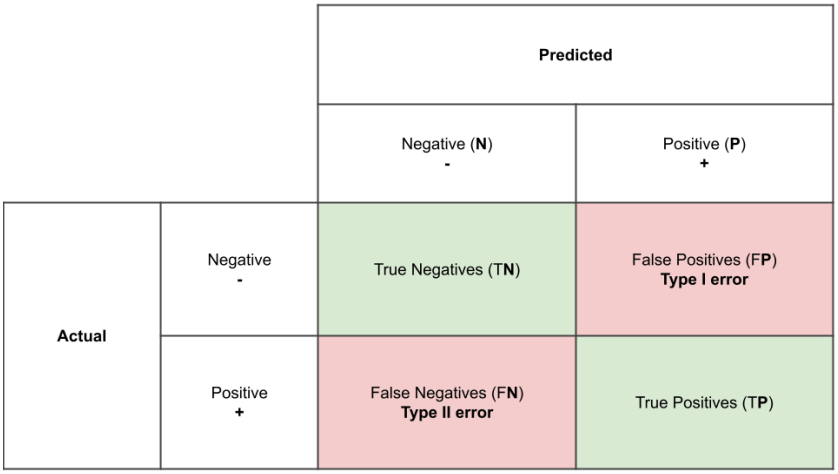


Figura 6.1: Matriz de confusión.

confusión, es el cociente entre verdaderos positivos y la suma de verdaderos positivos y falsos positivos:

$$Precision = \frac{VP}{VP + VN} \tag{6.3}$$

6.1.5. Sensibilidad

La sensibilidad -recall, en inglés- se trata de la proporción de textos clasificados como positivos ($\hat{y} = 1$) entre todos los que son realmente positivos ($y = 1$). Trasladando esto la matriz de confusión, es el cociente entre verdaderos positivos y la suma de verdaderos positivos y falsos negativos:

$$Recall = \frac{VP}{VP + FN} \tag{6.4}$$

6.1.6. F1 Score

La puntuación F1 -F1 score, en inglés- es la media harmónica entre precisión y sensibilidad. Al existir un trade-off ellas, la métrica F1 ofrece una interpretación cuantitativa del mismo. Su rango se encuentra entre [0, 1]. Indica cómo de preciso es el algoritmo (cuánto clasifica de forma correcta) y cómo de robusto es (cuántas muestras pierde en el proceso).

Lo que hace la fórmula es castigar los valores extremos. Por ejemplo, un clasificador con una precisión de 1.0 y una sensibilidad de 0.0 tendrá una media simple de 0.5, pero

	Loss	Accuracy	Precision	Recall	F1-Score
Training	0.4693	0.7763	0.7819	0.7638	0.7727
Validation	0.4968	0.7562	0.7685	0.7339	0.7508
Test	0.4995	0.7535	0.7654	0.7261	0.7452

Cuadro 6.1: Métricas para CNN+GloVe 50 dimensiones, 10 % de datos

	Loss	Accuracy	Precision	Recall	F1-Score
Training	0.4248	0.8018	0.8086	0.7905	0.7994
Validation	0.4564	0.7847	0.7936	0.7697	0.7815
Test	0.4604	0.7819	0.7888	0.7672	0.7779

Cuadro 6.2: Métricas para CNN+GloVe 50 dimensiones, 50 % de datos

una puntuación F1 de 0. En nuestros modelos, buscaremos una F1 score teniendo en cuenta la precisión y la sensibilidad.

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{6.5}$$

6.2. Tablas de resultados

A continuación se muestran las tablas de resultados que se han obtenido tras el entrenamiento de los distintos modelos con los distintos conjuntos de datos. Las métricas usadas son las descritas en la sección anterior.

Como vemos, contamos con 8 tablas, en función de si el modelo es CNN+GloVe (50 ó 300 dimensiones), ELMo o BERT, así como dependiendo de si trabajamos con el dataset reducido al 10 % o al 50 %.

Se puede ver de forma clara cómo los resultados mejoran dependiendo de la complejidad del modelo: el peor caso sería la combinación de CNN+GloVe, que cuenta con 2 653 901 parámetros (de los cuales 62 951 entrenables), mientras que el caso que obtiene las mejores métricas sería BERT, el cual cuenta con 110 302 011 parámetros (de los cuales 198 657 son entrenables). Esta información se refiere al caso de muestreo del 10 %.

	Loss	Accuracy	Precision	Recall	F1-Score
Training	0.4186	0.8099	0.8131	0.8028	0.8079
Validation	0.4608	0.7794	0.786	0.7685	0.7772
Test	0.4666	0.7769	0.7812	0.7649	0.7730

Cuadro 6.3: Métricas para CNN+GloVe 300 dimensiones, 10 % de datos

	Loss	Accuracy	Precision	Recall	F1-Score
Training	0.4021	0.8166	0.8234	0.8059	0.8146
Validation	0.4271	0.8016	0.8096	0.7888	0.7991
Test	0.4303	0.8002	0.8067	0.7871	0.7968

Cuadro 6.4: Métricas para CNN+GloVe 300 dimensiones, 50 % de datos

	Loss	Accuracy	Precision	Recall	F1-Score
Training	0.412	0.8085	0.812	0.8007	0.8063
Validation	0.4535	0.7904	0.8026	0.7708	0.7864
Test	0.4564	0.7904	0.7986	0.7729	0.7855

Cuadro 6.5: Métricas para ELMo, 10 % de datos

	Loss	Accuracy	Precision	Recall	F1-Score
Training	0.3887	0.8227	0.8267	0.8165	0.8216
Validation	0.4223	0.8044	0.8092	0.7968	0.8030
Test	0.4282	0.8021	0.8055	0.7939	0.7997

Cuadro 6.6: Métricas para ELMo, 50 % de datos

	Loss	Accuracy	Precision	Recall	F1-Score
Training	0.4176	0.8076	0.8092	0.803	0.8061
Validation	0.434	0.8006	0.7933	0.8135	0.8033
Test	0.4366	0.7951	0.7864	0.8064	0.7963

Cuadro 6.7: Métricas para BERT, 10 % de datos

	Loss	Accuracy	Precision	Recall	F1-Score
Training	0.4072	0.8128	0.8145	0.8099	0.8122
Validation	0.4116	0.8108	0.8066	0.8179	0.8122
Test	0.418	0.807	0.802	0.8127	0.8073

Cuadro 6.8: Métricas para BERT, 50 % de datos

Ocurriría de forma equivalente para el caso del muestreo al 50 %.

Por otra parte, de forma general los resultados mejoran cuanta mayor es la muestra del dataset que tomamos: si comparamos las precisiones para los casos de validación y test, podemos ver cómo mejora una media de 2.5 % contando todos los modelos estudiados.

6.3. Gráficas de resultados

Para observar los resultados de forma más clara, a continuación podemos ver distintas gráficas que nos relacionan las métricas de pérdidas y precisión en función del modelo y la cantidad de datos seleccionada.

Estas gráficas nos permiten comprobar cómo se ha desarrollado el proceso de aprendizaje mediante las representaciones del entrenamiento y validación.

Debemos destacar que, tal como comentamos anteriormente, utilizamos un *callback* de tipo *early stopping*. En concreto, ponemos a *true* uno de sus atributos llamado *restore_best_weights*. Este mecanismo lo que hace es, no solo detener el entrenamiento cuando los resultados de validación no mejoran tras un número de *epochs*, sino que también asigna al modelo los pesos obtenidos en la iteración con mejores resultados. Esto último nos asegura no introducir *overfitting* aunque se vea en la representación gráfica.

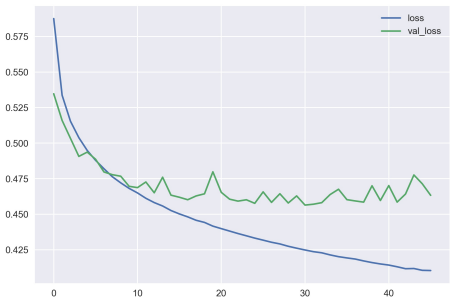
6.3.1. Loss

Idealmente, si hablamos de pérdidas del modelo, conforme pasan las iteraciones ambas deben converger hacia un mínimo. Sin embargo, en algunos casos podemos ver cómo las gráficas llegan a un punto donde se siguen reduciendo las pérdidas en el entrenamiento, pero en el conjunto de validación comienzan a incrementarse. Esto se trata de la representación gráfica del *overfitting* mencionado anteriormente.

Cabe destacar la utilización del optimizador *sgd* en los casos de CNN+GloVe, a diferencia de los casos del ELMo y BERT, donde utilizamos el optimizador *adam*. Sin entrar en mucho detalle, el segundo se trata de un optimizador más eficiente, pues hace que el modelo converja con mayor rapidez. Sin embargo, en el primer modelo decidimos utilizar *sgd* para que la convergencia sea más lenta, y poder así ver de forma más clara cómo se reducen las pérdidas con el paso de las iteraciones.

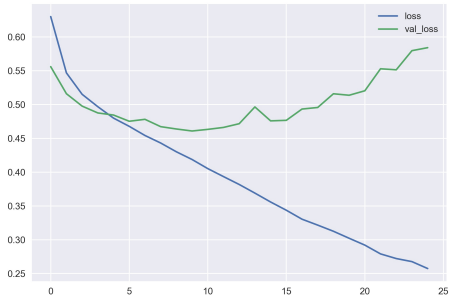
En las figuras 6.2, 6.3, 6.4 y 6.5 podemos observar la representación de estas pérdidas en función de los *epochs* transcurridos. Un *epoch* en el ámbito de las redes neuronales hace referencia a un ciclo completo en el que utilizamos por completo todos nuestros datos de entrenamiento para realizar exactamente una pasada.

(a) 10 % de datos.

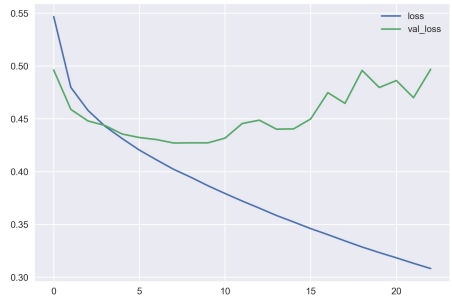


(b) 50 % de datos.

Figura 6.2: Pérdidas para CNN+Glove 50 dimensiones.

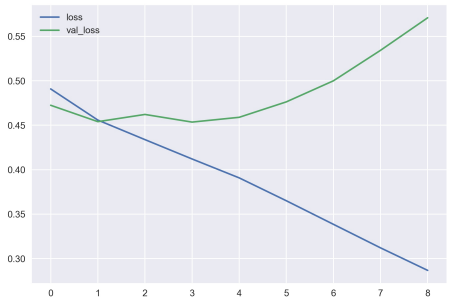


(a) 10 % de datos.

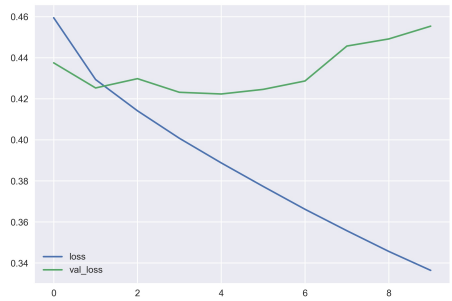


(b) 50 % de datos.

Figura 6.3: Pérdidas para CNN+Glove 300 dimensiones.

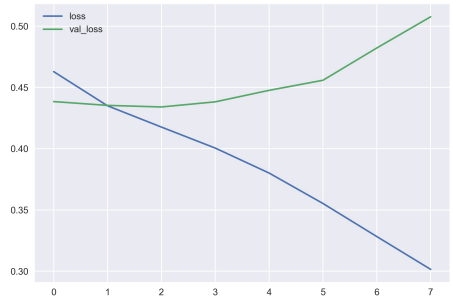


(a) 10 % de datos.

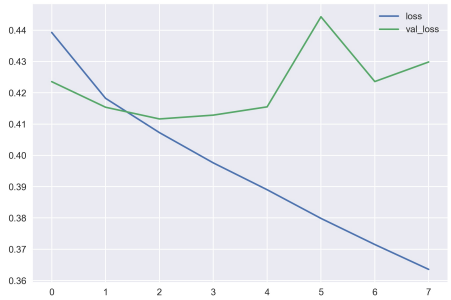


(b) 50 % de datos.

Figura 6.4: Pérdidas para ELMo.

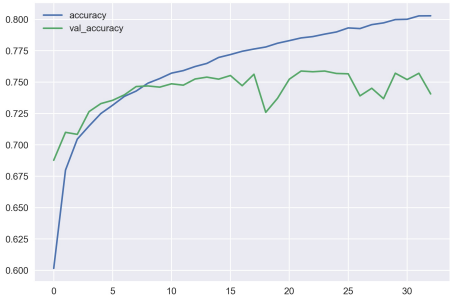


(a) 10 % de datos.

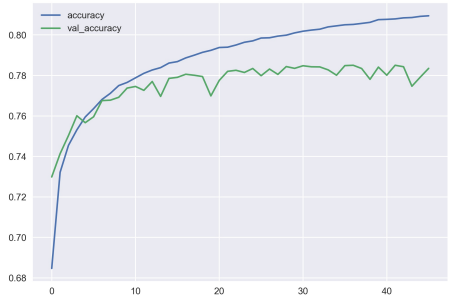


(b) 50 % de datos.

Figura 6.5: Pérdidas para BERT.



(a) 10 % de datos.



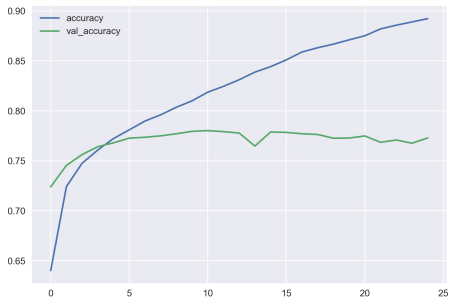
(b) 50 % de datos.

Figura 6.6: Precisión para CNN+Glove 50 dimensiones.

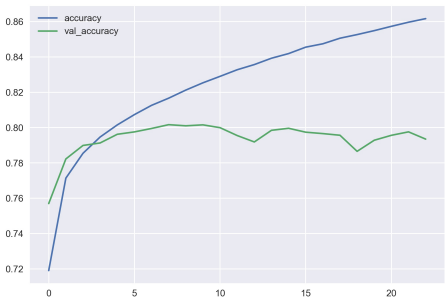
6.3.2. Accuracy

A la hora de estudiar la precisión del modelo, en las figuras 6.6, 6.7, 6.8 y 6.9 se puede observar que no se produce ninguna mejora sustancial desde que comenzamos el entrenamiento hasta que se detiene debido al *early stopping*. Esto se debe a que estamos trabajando con modelos pre-entrenados que ya desde las primeras iteraciones ofrecen buenos resultados.

Como vemos, en el mejor de los casos, trabajando con BERT y el 50 % de los datos, la precisión llega al 81 % en el conjunto de validación. Lo podemos considerar un buen resultado, teniendo en cuenta las dificultades que pueden presentar las tareas de procesado del lenguaje natural, y en concreto una como la de clasificación de textos.

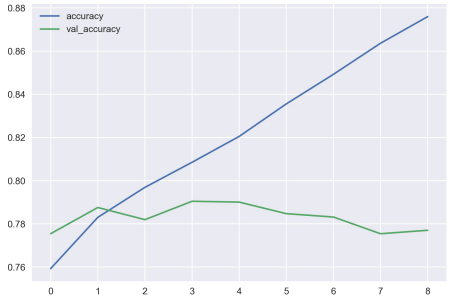


(a) 10 % de datos.

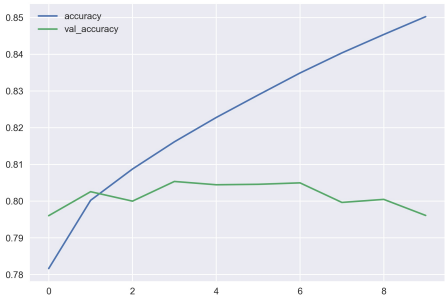


(b) 50 % de datos.

Figura 6.7: Precisión para CNN+Glove 300 dimensiones.

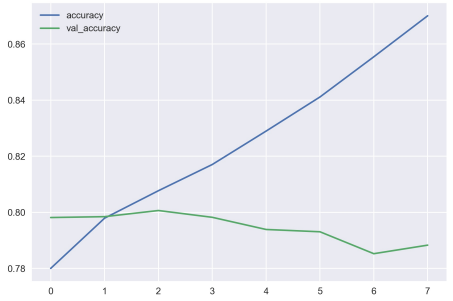


(a) 10 % de datos.

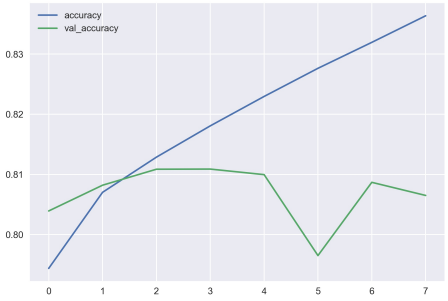


(b) 50 % de datos.

Figura 6.8: Precisión para ELMo.



(a) 10 % de datos.



(b) 50 % de datos.

Figura 6.9: Precisión para BERT.

Capítulo 7

Conclusión y líneas futuras

A lo largo de la realización de este Trabajo de Fin de Grado, hemos podido obtener las siguientes conclusiones.

En primer lugar, hemos comprobado cómo los modelos basados en redes neuronales son una alternativa muy prometedora dentro del procesado del lenguaje natural, y en concreto en tareas de clasificación de textos. Las precisiones obtenidas, que rondan el 80 %, se tratan de resultados aceptables para una tarea de estas características, sobre todo si tenemos en cuenta otros experimentos relacionados, como los que aparecen en el artículo *Convolutional Neural Networks for Sentence Classification* [6], donde se hace un análisis de distintos modelos para distintos conjuntos de datos. Además, es posible que el sistema desarrollado no supere las precisiones obtenidas debido a posibles fallos en el proceso de etiquetado. Como comentamos, el etiquetado se realizó mediante algoritmos NLP semi-supervisados y no supervisados, por lo que pueden existir errores en este proceso.

También hemos podido observar cómo los datos de partida de nuestro problema, los textos, deben ser transformados a un formato que sirva de entrada a un modelo matemático. Para esto, hemos explorado distintas alternativas, lo que nos ha dado lugar al estudio de distintos modelos. Tras ver los resultados, hemos podido comprobar que aquellos con una mayor cantidad de parámetros nos suelen ofrecer unos mejores resultados. Destacar también que al ser modelos entrenados o pre-entrenados, desde el principio ya obteníamos unos resultados relativamente aceptables, en los que únicamente era necesario ajustar ciertos pesos.

Esta conclusión nos lleva a hablar de una de las limitaciones que nos hemos encontrado durante el desarrollo: la elevada capacidad de cómputo necesaria para entrenar nuestros modelos. Como hablamos, trabajar con millones de parámetros trae consigo un inconveniente, y es que resulta inaccesible trabajar con un ordenador convencional. Sobre

todo para los casos de ELMo y BERT, donde se volvió indispensable el uso de un servidor remoto con una tarjeta gráfica de gama alta para llevar a cabo las simulaciones. Esto se trata de algo a lo que no todo el mundo tiene acceso y debe ser tenido en cuenta.

No obstante, cabe mencionar que la diferencia de los tiempos de entrenamiento no es del todo proporcional con las diferencias en los resultados de cada modelo. El ejemplo más claro se muestra si comparamos el peor modelo (CNN+GloVe 50 dimensiones, 10 %) frente al modelo con mejores resultados (BERT, 50 %). Respectivamente, los tiempos de entrenamiento son alrededor de 3 minutos y en torno a 20 horas, respectivamente, mientras que la diferencia de precisión en el conjunto de test ronda el 5 %. Quizás esa diferencia de precisión pueda ser crítica en algunos casos, pero cabe destacar que será a costa de un tiempo mucho mayor.

Si hablamos de líneas futuras de desarrollo, quizás la más clara sea pasar de una clasificación binaria (positivo o negativo) a una que cuente con múltiples categorías, como pueden ser distintos sentimientos como alegría, ironía, odio o tristeza, o bien distintos grados dentro de lo positivo o lo negativo. La principal barrera para este tipo de trabajos es encontrar conjuntos de datos lo suficientemente amplios y de calidad como para poder entrenar una red. A efectos prácticos, podríamos mantener el mismo flujo de trabajo y arquitectura, haciendo únicamente algunos pequeños cambios para adaptar nuestras etiquetas y nuestras salidas a un modelo con múltiples categorías.

Por otra parte, también habría sido interesante ver qué resultados obtenemos al trabajar con el dataset completo, lo que sería el doble de datos de nuestro dataset con la menor reducción. Desafortunadamente, al intentar este escenario, se produce un cuello de botella en la cantidad memoria necesaria para trabajar con un número tan elevado de parámetros. Se trata de una limitación de nuestro hardware, pero que podría llegar a ser posible en otro trabajo con un servidor aún más potente.

Dado que todas nuestras soluciones han estado basadas en el aprendizaje supervisado, otra línea posible sería explorar modelos basados en aprendizaje semi-supervisado o sin supervisar. Para este caso, el enfoque sería totalmente distinto, pudiendo utilizar otro tipo de bases de datos sin etiquetar, pero debido a la extensión del proyecto no hemos podido abarcar este tipo, aunque sí que sería interesante para comparar qué soluciones se ajustan mejor a una tarea de estas características.

A nivel académico, este proyecto ha supuesto mi primer contacto con Python y el mundo de la ciencia de datos, así como el ámbito del aprendizaje máquina y el procesado del lenguaje natural. He podido trabajar con multitud de librerías dentro de este lenguaje, pudiendo experimentar con datos reales y resultados aplicados a un caso práctico.

Ha sido muy interesante para mí trabajar en este campo, principalmente por las grandes posibilidades que ofrece. Si bien este trabajo se ha limitado al NLP, he podido comprobar la versatilidad que tienen los algoritmos de machine learning estudiados y

las innovaciones que se van desarrollando con el tiempo. Sin duda, es un tema que dará mucho que hablar en el futuro cercano.

Bibliografía

[1] Facundo Bre, Juan Gimenez, and Víctor Fachinotti. Prediction of wind pressure coefficients on building surfaces using artificial neural networks. *Energy and Buildings*, 158:4, 11 2017.

[2] Imperial College. Neural networks. `https://www.doc.ic.ac.uk/~nuric/teaching/imperial-college-machine-learning-neural-networks.html`.

[3] Chaim Baskin, Natan Liss, Avi Mendelson, and Evgenii Zheltonozhskii. Streaming architecture for large-scale quantized neural networks on an fpga-based dataflow platform. 07 2017.

[4] Stanford University. Cs231n convolutional neural networks for visual recognition. `https://cs231n.github.io/convolutional-networks/#pool`.

[5] Meghna Asthana. Introduction to convolutional neural networks. `https://medium.com/analytics-vidhya/introduction-to-convolutional-neural-networks-c50f41e3bc66`.

[6] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics.

[7] GFT. Cómo usar redes neuronales (lstm) en la predicción de averías en las máquinas. `https://blog.gft.com/es/2018/11/06/como-usar-redes-neuronales-lstm-en-la-prediccion-de-averias-en-las-maquinas/`.

[8] Masato Hagiwara. Improving a sentiment analyzer using elmo — word embeddings on steroids. `http://www.realworldnlpbook.com/blog/improving-sentiment-analyzer-using-elmo.html`.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2017.

[10] Andreas Pogiatzis. Nlp: Contextualized word embeddings from bert. https://towardsdatascience.com/nlp-extract-contextualized-word-embeddings-from-bert-keras-tf-67ef29f60a7b.

[11] Linus Bengtsson Xin Lu and Petter Holme. Predictability of population displacement after the 2010 haiti earthquake. https://www.flowminder.org/resources/publications/predictability-of-population-displacement-after-the-2010-haiti-earthquake/.

[12] Gobinda G. Chowdhury. Natural language processing. *Annual Review of Information Science and Technology,*, 37:51–89, 2003.

[13] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, Sebastopol, CA, 2017.

[14] Wei Wang and Jianxun Gang. Application of convolutional neural network in natural language processing. In *2018 International Conference on Information Systems and Computer Aided Education (ICISCAE)*, pages 64–70, 2018.

[15] IBM. Neural networks. https://www.ibm.com/cloud/learn/neural-networks.

[16] Jose Miguel Fernandez Raquel Flórez López. *Las Redes Neuronales Artificiales.* 2009.

[17] WILDML. Understanding convolutional neural networks for nlp. http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/.

[18] Universitat Oberta de Catalunya. Procesamiento del lenguaje natural (nlp). http://datascience.recursos.uoc.edu/es/procesamiento-del-lenguaje-natural-nlp/.

[19] Guido Van Rossum. *The Python Library Reference, release 3.8.2.* Python Software Foundation, 2020.

[20] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[21] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[23] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[24] François Chollet et al. Keras. `https://keras.io`, 2015.

[25] Hugo Andrade, Soraya Sinche, and Pablo Hidalgo. Modelo para detectar el uso correcto de mascarillas en tiempo real utilizando redes neuronales convolucionales. *Revista de Investigación en Tecnologías de la Información*, 9:111–120, 01 2021.

[26] Antonio Fernández Anta, Philippe Morere, Luis Chiroque, and A. Santos. Sentiment analysis and topic detection of spanish tweets: A comparative study of nlp techniques. *Procesamiento de Lenguaje Natural*, 50:45–52, 03 2013.

[27] Rui Xia and Zixiang Ding. Emotion-cause pair extraction: A new task to emotion analysis in texts. In *ACL*, 2019.

[28] Lin Gui, Dongyin Wu, Ruifeng Xu, Qin Lu, and Yu Zhou. Event-driven emotion cause extraction with corpus construction. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1639–1649, Austin, Texas, November 2016. Association for Computational Linguistics.

[29] Alec Go, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision. `http://help.sentiment140.com/home`, 2009.

[30] Shahul ES. Exploratory data analysis for natural language processing: A complete guide to python tools. `https://neptune.ai/blog/exploratory-data-analysis-natural-language-processing-tools`, 2021.

[31] Jason S. Kessler. Scattertext: a browser-based tool for visualizing how corpora differ. 2017.

[32] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[33] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

[34] Christopher Olah. Understanding lstm networks. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[35] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[36] Rani Horev. Bert explained: State of the art language model for nlp. `https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270`.

[37] Alex Bäuerle, Christian van Onzenoodt, and Timo Ropinski. Net2vis – a visual grammar for automatically generating publication-tailored cnn architecture visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 27(6):2980–2991, 2021.

[38] Daniel Godoy. Understanding binary cross-entropy / log loss: a visual explanation. `https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a`.