UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Resumen de textos con un modelo secuencia-a-secuencia: varias aproximaciones

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Matías Peirano Maletta

Tutor: Jon Ander Gómez Adrián

Curso 2019-2020

Agradecimientos

Son varias las instituciones y personas que han contribuido, de manera directa o indirecta, a la realización de este trabajo. En particular quiero agradecer:

En primer lugar a la empresa Solver M.L. y a todos mis compañeros de trabajo la oportunidad que me ha brindado además de ofrecerme todos los recursos disponibles para realizar este proyecto y aprender de él.

En segundo lugar a mi tutor Jon Ander Gómez por la ayuda, consejos y dedicación que me ha prestado durante la elaboración de este trabajo de fin de grado.

En tercer lugar a mi amigo Lucas Renovell Ferrer por ser mi mentor en la informática, por darme la oportunidad de poner en práctica las competencias adquiridas en la universidad en casos reales y por saciar mi curiosidad con respuestas detalladas y completas a todas mis preguntas.

Finalmente, a mis padres por el apoyo incondicional que me han brindado no solo durante la realización de este proyecto sino durante toda mi vida académica.

Resum

En l'àmbit de l'aprenentatge automàtic profund, els models neuronals seqüència-a-seqüència han demostrat obtenir molt bons resultats en la resolució de tasques de processament del llenguatge natural. En aquest treball ens centrem específicament en la resolució d'un problema d'aquest camp: resumir textos fent servir aquests models. S'explica quines dificultats hem trobat en cada fase del procés i com bregar amb elles. A més es mostren diferents aproximacions per a resoldre el problema, tant en el processament de dades com en la construcció del model. Posteriorment es comparen les diferents versions entre sí i amb l'estat de l'art fent servir els criteris ROUGE i BLEU.

Paraules clau: seq2seq, machine learning, deep learning, text summarization

Resumen

En el ámbito del aprendizaje profundo, los modelos neuronales secuencia-a-secuencia han demostrado obtener muy buenos resultados en la resolución de tareas de procesamiento del lenguaje natural. En este trabajo nos centramos específicamente en la resolución de un problema de este campo: resumir textos haciendo uso de dichos modelos. Se explica qué dificultades hemos encontrado en cada fase del proceso y como lidiar con ellas. Además se muestran distintas aproximaciones para resolver el problema, tanto en el procesamiento de datos como en la construcción del modelo. Posteriormente se comparan las distintas versiones entre sí y con el estado del arte haciendo uso de los criterios ROUGE y BLEU.

Palabras clave: secuencia a secuencia, aprendizaje automático, aprendizaje profundo, resumen de textos

Abstract

In the field of deep learning, sequence-to-sequence neural models have been shown to achieve very good results in solving natural language processing tasks. In this work we focus specifically on solving a problem in this field: summarizing texts using such models. It explains what difficulties we have encountered in each phase of the process and how to deal with them. In addition, different approaches to solve the problem are shown, both in data processing and model construction. Subsequently, the different versions are compared with each other and with the state of the art using the ROUGE and BLEU criteria.

Key words: seqüència a seqüència, aprenentatge automàtic, aprenentatge profund, resum de text

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

La inteligencia artificial (IA) ha sufrido dos grandes inviernos. Aunque las fechas de estos varían dependiendo del autor, aproximadamente se puede comprender el primero desde 1974 hasta 1980 y el segundo desde 1984 hasta principios de los años 2000. Pero gracias a los avances de hardware que han derivado en una gran capacidad de computo y gracias a la recolección y procesamiento de grandes cantidades de datos, este campo a resurgido de nuevo y evoluciona de una manera frenética.

La disciplina, aunque relativamente reciente (años 50), es muy extensa y tiene sus propias ramificaciones. Este proyecto se categoriza en el subcampo del *Machine Learning* (ML), en concreto en el *Deep Learning* (DL), que se caracteriza por utilizar arquitecturas más complejas en comparación con el primero.

En este trabajo haremos uso del DL para resolver la tarea de resumir textos. Esta puede considerarse un caso especial del problema de traducción automática en el que el lenguaje de entrada y de salida son el mismo pero la longitud del texto como salida es notablemente inferior. Es decir, se desea que **el modelo aprenda a comprender la semántica del texto de entrada para así poder representarla de una manera más breve y concisa.**

No solo nos centraremos en el procedimiento de la construcción del modelo sino que también expondremos los pasos necesarios para la carga de datos, el procesamiento previo de estos antes del entrenamiento, la inferencia y la evaluación de los resultados mediante métricas específicas; comentando los problemas encontrados durante la realización de los experimentos y qué opciones y variantes tiene cada modelo.

Abordaremos el problema desde distintos niveles de abstracción, haciendo uso de varias librerías de ML entre las cuales se incluyen Tensorflow, Keras y una muy reciente pero prometedora: *hugging face* (transformers). De esta manera se ofrece, por una parte, una explicación detallada de los modelos más sencillos que conforman las bases del campo, precursores de modelos más avanzados. Por otra parte, se expone de manera superficial los modelos más complejos que componen el estado del arte. Esto permite dominar los fundamentos del campo a la vez que se tiene una visión general de la literatura más actual.

1.1 Motivación

Dentro del ámbito que acabamos de describir encontramos dos formas de realizar los resúmenes, de manera **extractiva** y de manera **abstractiva**. En la primera los resúmenes se generan seleccionando las partes del texto de entrada elegidas como las más representativas mientras que en la segunda la salida es libre en cuanto a que puede estar formada por cualquier palabra del vocabulario y en cualquier orden. En esta última aproximación se fuerza al sistema a comprender la semántica y la estructura de las frases.

Esto último se acerca a lo que sucede en los modelos de lenguaje, modelos que pretenden resolver los problemas de *Natural Language Processing* (NLP) de una manera genérica y no entrenando un modelo distinto para cada uno de ellos: responder preguntas, comprensión lectora, traducción, resumir textos, etc. Más concretamente estos modelos intentan calcular la probabilidad de que una palabra del vocabulario sea la siguiente producida dadas las palabras anteriores. Para poder realizar esta tarea correctamente el modelo necesita entender no solo la estructura de las frases sino también el significado que estas tienen. Por lo tanto los avances realizados en el resumen abstractivo de textos se pueden aplicar directamente o indirectamente a los modelos de lenguaje.

Esta capacidad de resolver varios problemas de NLP con un único modelo más general en vez de varios modelos específicos es una propiedad muy perseguida en el campo del ML ya que da un gran paso en la dirección de conseguir una inteligencia artificial fuerte. Este tipo de inteligencia artificial promete ser capaz de resolver problemas de distintas áreas conceptuales, tal y como lo hacemos los seres humanos. Hasta la fecha todas las IA se clasifican como débiles ya que están especializadas en resolver una única tarea.

1.2 Objetivos

El objetivo principal de este trabajo es **reunir y tratar en un solo documento todos los aspectos a tener en cuenta a la hora de enfrentarse a la tarea de resumir textos**, siendo este procedimiento extensible a otras tareas en el campo del NLP. De esta forma pretendemos que este estudio se pueda utilizar como una guía a seguir o punto de partida para futuras aproximaciones a este problema o problemas similares. A continuación desglosamos el objetivo principal en distintas partes constituyentes.

En primera instancia uno de los objetivos es comprender los conceptos propios del campo e identificar cuales son los procedimientos a seguir para la realización de esta tarea. A posteriori esto nos permite definir mejor el resto de objetivos. Consecutivamente se pretende **comprender qué arquitecturas se usan para la creación de modelos de resumen de texto** y porqué. Además se contempla estructurar el proyecto de tal manera que se puedan seguir los mismos procedimientos para distintos tipos de problemas dentro del campo con cambios mínimos. Seguidamente se comentan las etapas generales que componen este flujo de trabajo.

1. **Carga de datos:** Explicar las distintas fuentes de datos disponibles en un caso real y cómo cargar estos en nuestro programa.
2. **Preprocesamiento de datos:** Tratar los datos mediante librerías como NLTK y Gensim para diversos fines como mejorar el aprendizaje del modelo, reducir el coste espacial y temporal y eliminar muestras corruptas.
3. **Creación de modelos:** Aprender qué tipo de arquitecturas se emplean para entrenar modelos para resumir textos, cómo crear estas haciendo uso de librerías de ML como Tensorflow y Keras, qué tipo de capas existen y cómo utilizarlas.
4. **Entrenamiento:** Comprender la importancia de los hiperparámetros y que papel juegan a la hora de entrenar los modelos. También comentar opciones de visualización del entrenamiento y guardado de modelos que ofrecen las librerías.
5. **Inferencia:** Diseñar una estructura común para realizar inferencia en los distintos tipos de modelos independientemente de sus arquitecturas o hiperparámetros, además de comentar distintos algoritmos con sus beneficios y desventajas.
6. **Evaluación:** Exponer cuales son las métricas preferidas para la evaluación de esta tarea y porqué no se deben utilizar las comunes a todos los modelos.

Además del objetivo principal también consideramos el aprendizaje y acercamiento del alumno a las herramientas y conceptos propios del campo, así como a los problemas y soluciones que se encuentran en la literatura. Adicionalmente se contempla la creación manual de distintos modelos para posteriormente compararlos entre ellos y con los resultados del estado del arte.

1.3 Estructura de la memoria

- **Capítulo 1 ~Introducción:** El primer capítulo del trabajo introduce al lector el tema a tratar además de explicar la motivación del alumno para realizar este proyecto. Adicionalmente expone los objetivos a conseguir y describe la estructura del trabajo.
- **Capítulo 2 ~Estado del arte:** En esta capítulo se comenta la historia del campo y los avances más relevantes respecto a este estudio. Comentario sobre las tecnologías más avanzadas del campo del NLP, en concreto en la tarea de resumir textos: modelos de atención, transformers, etc.
- **Capítulo 3 ~Descripción de la tecnología empleada:** En esta sección presentamos las herramientas seleccionadas para la realización de este trabajo y ejemplificamos algunas de ellas para su mejor entendimiento.
- **Capítulo 4 ~Descripción del sistema:** Este capítulo expone el proceso a seguir para el desarrollo, entrenamiento y evaluación de un modelo creado para resolver la tarea de resumir textos.

- **Capítulo 5 ~Resultados:** En este capítulo se describen los distintos tipos de modelos creados así como los resultados obtenidos por estos.
- **Capítulo 6 ~Conclusiones:** En la última sección de este trabajo se presentan las conclusiones obtenidas respecto al proceso de enfrentarse a la tarea así como respecto a los resultados obtenidos por los modelos.
- Adicionalmente se puede encontrar la bibliografía y un glosario de términos con sus definiciones al final de este trabajo.

CAPÍTULO 2
Estado del arte

La velocidad con la que se producen nuevos avances en el campo del ML es frenética. En el momento de empezar este trabajo, los modelos que ocupan las dos primeras posiciones del estado del arte en la tarea de resumir textos de manera abstractiva aún no existían. Esto provoca que la cantidad de artículos sobre el tema sea muy extensa y haya una amplia variedad de enfoques distintos para intentar resolver este problema. A continuación se presentan los modelos y arquitecturas más relevantes y con mejores resultados de la literatura.

2.1 Redes neuronales recurrentes

Por la naturaleza secuencial del texto, una aproximación básica es el uso de redes neuronales recurrentes (RNR), estas procesan las palabras o tokens uno a uno para generar un contexto general de toda la frase en cuestión. En oposición a las redes neuronales prealimentadas (*feedforward neural networks*), las RNR presentan bucles de retroalimentación, es decir, la salida de la red en la iteración n se introduce como entrada adicional en la iteración $n + 1$. Esto permite propagar la información durante varios pasos dentro de la secuencia procesada.

Mediante el uso de estas redes se puede construir una arquitectura *encoder-decoder*. En esta arquitectura un primer módulo procesa los datos de entrada y construye un contexto general de toda la secuencia. Seguidamente este contexto se introduce como entrada del segundo módulo, además de un término de iniciación, para empezar a generar una secuencia de salida. Tanto el *encoder* como el *decoder* constan de al menos una RNR.

Un problema muy preocupante que presentaban las primeras versiones de RNR era el *Vanishing Gradient Problem* [1]. Este problema se da en las redes neuronales profundas, es decir, aquellas con un elevado número de capas. A efectos prácticos, una RNR se puede considerar una red *feedforward* con tantas capas como longitud de la secuencia procesada, es por esto que dicho problema aparece indistintamente del tipo de red.

Existen al menos 20 tipos distintos de RNR pero la predilecta hasta la fecha es la LSTM (*Long Short Term Memory*) [2], esta fue propuesta en 1997 y desde entonces la gran mayoría de modelos con RNR implementan esta variante. Esto se debe a que esta red permite conservar la información de iteraciones arbitraria-

mente lejanas a la actual. La LSTM consta de una célula, una puerta de entrada, una puerta de salida y una puerta de olvido. La célula almacena la información entre iteraciones mientras que las puertas regulan el flujo de dicha información.

En 2014 Kyunghyun Cho et al. [3] propusieron una variante simplificada de la LSTM, la GRU, que aumentaba su eficiencia manteniendo su eficacia. La *Gated recurrent units* (GRU) se diferencia de la LSTM por no tener puerta de salida, puerta que guarda un estado interno de la red. Aunque se ha demostrado que la LSTM es estrictamente superior en la tarea de modelado [4], la GRU está ganando terreno al ser más eficiente y obtener los mismos resultados en cierto tipo de problemas.

2.2 Modelos de atención

A pesar de la introducción de las LSTM y las GRU, el problema de interdependencia de palabras alejadas entre sí persistía. Los modelos no eran capaces de captar dichas relaciones entre posiciones distanciadas. Esto se consiguió solucionar mediante un mecanismo llamado atención. Esta técnica revolucionaria consiste en almacenar los contextos que se producen en cada iteración del *encoder* para después pasarlos conjuntamente al *decoder* de tal manera que el modelo aprenda a "prestar atención" a las partes de la secuencia de entrada que más le convenga.

2.3 Transformers

Aunque se había solucionado en parte el problema de dependencias entre palabras alejadas entre sí dentro del texto, la naturaleza recurrente de estas redes hacia imposible su escalabilidad al no ser fácilmente paralelizables. Es aquí donde Vaswani et al. [5] proponen un modelo basado únicamente en mecanismos de atención, evitando cualquier tipo de recurrencia o convolución. Este modelo, además de ser menos exigente computacionalmente, mejoró el estado del arte obtenido hasta la fecha.

2.4 BERT

Bidirectional Encoder Representations from Transformers (BERT) [6] es un modelo basado en *transformers* diseñado para crear representaciones de las secuencias condicionadas por los contextos de la capa anterior y la siguiente, esto permite conseguir resultados del estado del arte simplemente añadiendo una capa de salida para la tarea en cuestión.

2.5 BART

Esta aproximación consta de dos partes, en primer lugar se corrompe el texto de entrada y después se entrena un modelo para que aprenda a reconstruirlo. Este usa una arquitectura basada en transformers lo que provoca que sea altamente paralelizable. Una de las novedades que presenta BART [7] es el uso de un *encoder* bidireccional y un *decoder* con flujo de izquierda a derecha. Esto puede verse como una generalización de BERT, GPT [] y otros arquitecturas recientes.

2.6 Get To The Point

Esta propuesta [8] presenta una arquitectura pointing-generator la cual permite copiar datos directamente del texto de entrada gracias al *pointing* a la vez que producir nuevos mediante el *generator*.

2.7 PEGASUS

La novedad que introduce *PEGASUS* [9] es enmascarar las frases importantes del documento de entrada para que el modelo aprenda a generarlas a partir del resto de frases.

2.8 ProphetNet

La principal diferencia que presenta *ProphetNet* [10] es la predicción de n-gramas en lugar de un único término por iteración, esto incentiva al modelo a pensar a futuro, evitando el overfiting en el caso de encontrar fuertes correlaciones locales.

2.9 ERNIE-GEN

ERNIE-GEN [11] propone 3 metodos para mejorar la habilidad generativa del modelo:

- *Span-by-span generation pre-training task* para que el modelo genere unidades sintácticamente completas en vez de una única palabra.
- *Infilling generation mechanism* para evitar el problema de *Exposure Bias*.
- *Multi-Granularity Target Fragments* para alentar la dependencia del *decoder* en el *encoder* en la fase de preentrenamiento.

CAPÍTULO 3
Descripción de la tecnología empleada

Ahora procederemos a enumerar y explicar las herramientas seleccionadas para el desarrollo de este proyecto. Aunque se han utilizado diversas librerías, solo se comentan en detalle aquellas que están más relacionadas con la tarea que nos ocupa.

Es preciso mencionar que el lenguaje de programación empleado es Python. Este se ha establecido como el lenguaje predilecto en el ámbito de la ciencia de datos. Aunque Python es un lenguaje de propósito general y hay otros más específicos para este fin como R u Matemática, posee unas características que hacen que sea el preferido por la mayoría de usuarios:

- Su sintaxis es clara y sencilla, esto deriva en que sea uno de los lenguajes más recomendados para iniciarse en la programación. Lo cual permite que su comunidad crezca día a día.
- Al ser un lenguaje de propósito general cuenta con miles de librerías para realizar todo tipo de tareas. Por un lado, esto permite la inclusión de módulos de ciencia de datos en programas ya existentes. Por otro lado, esta característica ofrece la posibilidad de crear una solución completa (carga de datos, interacción con otros sistemas, interfaz gráfica...) para un problema y no limitarse únicamente a tareas de ciencia de datos.
- Se suele comentar que al ser un lenguaje interpretado su velocidad de cómputo es menor que la de otros lenguajes como Java o C++, pero Python cuenta con librerías internas escritas en C para agilizar procesos y operaciones comunes. Algunas de las herramientas que comentamos más adelante hacen uso de estas funciones para optimizar sus procesos.
- Gracias a las características descritas en los puntos anteriores Python cuenta con una gran comunidad, lo cual es otro de sus puntos fuertes ya que en el caso de encontrarte atascado en tu programa puedes navegar por foros como StackOverflow [1] y encontrar rápidamente otros desarrolladores con los mismos problemas que tú y las soluciones que la comunidad ha aportado para resolverlos.

[1] http://stackoverflow.com/

Dado que el lenguaje de programación empleado es Python, como entorno de desarrollo integrado se ha elegido PyCharm. Este software esta disponible tanto para Windows como para Linux, característica deseable ya que durante el desarrollo de este proyecto se ha trabajado en ambos sistemas.

El orden en el que se presentan las distintas librerías es deliberado. La intención es presentar las herramientas según su orden de utilización en el proyecto y en herramientas con aparición simultánea, presentarlas según su nivel de abstracción, de menor a mayor.

3.1 NLTK

Natural Language ToolKit (NLTK) como su nombre indica es una herramienta para el desarrollo de NLP. Esta provee gran cantidad de utilidades entre las que se encuentran bases de datos léxicas como WordNet y librerías para analizar, clasificar, etiquetar y tokenizar texto. Además contiene funciones para evaluar el texto según determinadas métricas como *BLEU* [13].

La funcionalidad más utilizada de esta librería de código abierto es *word_tokenize*, que se encuentra directamente en la raíz del paquete NLTK. Esta permite tokenizar una frase mediante expresiones regulares para conseguir una mejor división del texto. Esto es posible gracias al conocimiento sobre el idioma del texto a tratar. Por ejemplo en inglés se dividen ciertas contracciones como "don't" en "do" y "n't".

Otra utilidad muy interesante es *pos_tag*, esta permite inferir la categoría gramatical de los componentes de la oración en cuestión. Esta información es muy valiosa para realizar otras funciones como *lemmatize*, la cual de por sí lematiza las palabras basándose en un corpus. Pero si además se le entrega la información gramatical de la palabra, esta devolverá una raíz con la misma categoría. Es decir, si, por ejemplo, la palabra en cuestión es "trabajador" y la categoría indicada por la función *pos_tag* es "sustantivo", *lemmatize* devolverá "trabajador" en vez de "trabajar".

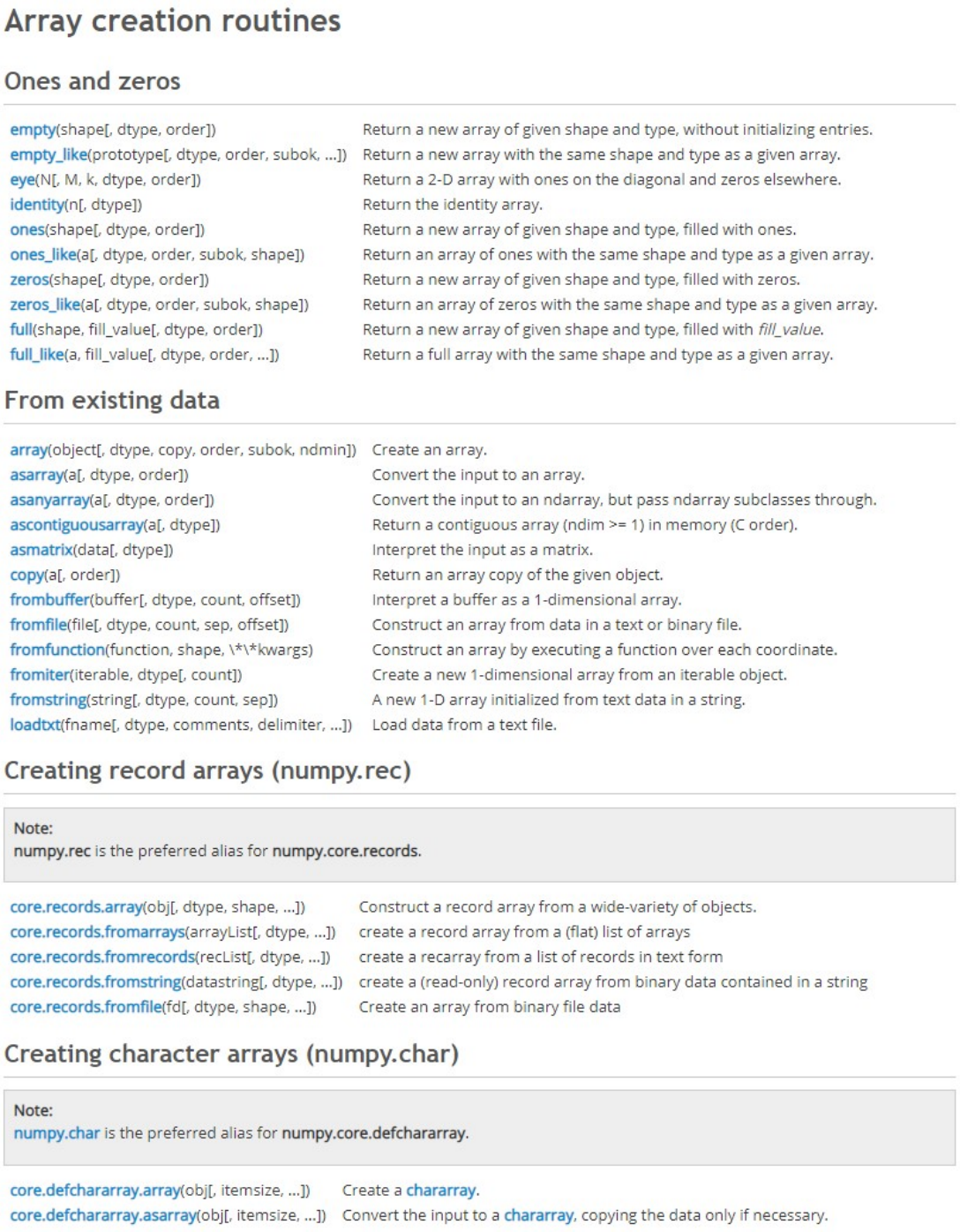
Es preciso nombrar la existencia de SpaCy. Esta es una librería similar a *NLTK* en cuanto a funcionalidad se refiere aunque promete ser más rápida y además añade algunas utilidades muy interesantes como el uso de *pipelines* para configurar un flujo de procesado de datos de forma sencilla y la capacidad de poder cargar directamente la información necesaria de distintos idiomas para procesar nuestros documentos.

3.2 Numpy

Numpy es una biblioteca matemática que extiende las capacidades que provee Python puro sobre los vectores y matrices. Su núcleo está escrito en C por lo tanto consigue una gran eficiencia a la hora de trabajar con grandes volúmenes de datos. Gracias a esto *Numpy* se ha convertido en una librería ubicua en el ámbito de la ciencia de datos. Consecuentemente la mayoría de las herramientas que

presentaremos a continuación implementan *Numpy* internamente, lo cual provoca que los objetos introducidos y devueltos entre funciones de distintas librerías sean compatibles y no necesiten conversión de tipos.

A modo de ejemplo se expone una lista de rutinas para la creación de arrays que ofrece Numpy:



Array creation routines

Ones and zeros

empty(shape[, dtype, order])	Return a new array of given shape and type, without initializing entries.
empty_like(prototype[, dtype, order, subok, ...])	Return a new array with the same shape and type as a given array.
eye(N[, M, k, dtype, order])	Return a 2-D array with ones on the diagonal and zeros elsewhere.
identity(n[, dtype])	Return the identity array.
ones(shape[, dtype, order])	Return a new array of given shape and type, filled with ones.
ones_like(a[, dtype, order, subok, shape])	Return an array of ones with the same shape and type as a given array.
zeros(shape[, dtype, order])	Return a new array of given shape and type, filled with zeros.
zeros_like(a[, dtype, order, subok, shape])	Return an array of zeros with the same shape and type as a given array.
full(shape, fill_value[, dtype, order])	Return a new array of given shape and type, filled with *fill_value*.
full_like(a, fill_value[, dtype, order, ...])	Return a full array with the same shape and type as a given array.

From existing data

array(object[, dtype, copy, order, subok, ndmin])	Create an array.
asarray(a[, dtype, order])	Convert the input to an array.
asanyarray(a[, dtype, order])	Convert the input to an ndarray, but pass ndarray subclasses through.
ascontiguousarray(a[, dtype])	Return a contiguous array (ndim >= 1) in memory (C order).
asmatrix(data[, dtype])	Interpret the input as a matrix.
copy(a[, order])	Return an array copy of the given object.
frombuffer(buffer[, dtype, count, offset])	Interpret a buffer as a 1-dimensional array.
fromfile(file[, dtype, count, sep, offset])	Construct an array from data in a text or binary file.
fromfunction(function, shape, **kwargs)	Construct an array by executing a function over each coordinate.
fromiter(iterable, dtype[, count])	Create a new 1-dimensional array from an iterable object.
fromstring(string[, dtype, count, sep])	A new 1-D array initialized from text data in a string.
loadtxt(fname[, dtype, comments, delimiter, ...])	Load data from a text file.

Creating record arrays (numpy.rec)

Note:
numpy.rec is the preferred alias for **numpy.core.records**.

core.records.array(obj[, dtype, shape, ...])	Construct a record array from a wide-variety of objects.
core.records.fromarrays(arrayList[, dtype, ...])	create a record array from a (flat) list of arrays
core.records.fromrecords(recList[, dtype, ...])	create a recarray from a list of records in text form
core.records.fromstring(datastring[, dtype, ...])	create a (read-only) record array from binary data contained in a string
core.records.fromfile(fd[, dtype, shape, ...])	Create an array from binary file data

Creating character arrays (numpy.char)

Note:
numpy.char is the preferred alias for **numpy.core.defchararray**.

core.defchararray.array(obj[, itemsize, ...])	Create a chararray.
core.defchararray.asarray(obj[, itemsize, ...])	Convert the input to a chararray, copying the data only if necessary.

Figura 3.1: Lista de (algunas) funciones Numpy para crear arrays.

3.3 Pandas

Pandas es una herramienta para el análisis, manipulación y visualización de datos. Esta utiliza *Numpy* y crea un nivel de abstracción superior con el que permite manejar estructuras de datos más complejas y no solo de tipo numérico, presentando además un gran rendimiento. Adicionalmente contiene funcionalidades para la carga de datos directamente desde archivos, lo cual facilita el proceso cuando trabajamos con los tipos de archivos más comunes.

Una de las utilidades más ventajosas de Pandas es su clase *Dataframe*, una estructura bidimensional con campos etiquetados y potencialmente de cualquier tipo, similar a la una tabla de SQL. Una práctica muy común en el desarrollo de ML es cargar el dataset en un objeto de tipo Dataframe y utilizar este para el procesado de los datos. Esta manera de trabajar ofrece la ventaja de contener todos los datos en un único objeto en el cual están asociados por filas y por columnas, lo cual asegura (o al menos facilita) la consistencia en operaciones de inserción, modificación, borrado, etc.

De igual forma, Pandas provee la clase *Series*, un array unidimensional capaz de almacenar valores de cualquier tipo (por cada objeto todos los elementos deben ser del mismo tipo). Los objetos *Dataframe* y *Series* están fuertemente relacionados ya que acceder a las filas o a las columnas de un *Dataframe* resulta en un *Series*. Es decir, se puede pensar que un *Datafrfame* está compuesto de *Series*. Esto supone una ventaja ya que a la hora de transformar los datos en objetos reconocibles por librerías de ML como *Tensorflow*, sus funciones aceptan objetos de tipo *Series* al ser estos iterables.

3.4 Matplotlib

Esta biblioteca permite visualizar estructuras de datos como listas, vectores y matrices. Al igual que *Pandas*, *Matplotlib* también esta construida sobre *Numpy*, por lo cual podemos visualizar los objetos de esta última sin necesidad de realizar ningún tipo de conversión intermedia.

Aunque pueda parecer que una librería de visualización no aporta mucho al proceso de desarrollo de un modelo, estas son una parte casi indispensable en él. Las distintas funcionalidades que ofrecen las herramientas de visualización permiten observar la forma que presentan los datos de nuestro corpus, detectar posibles valores atípicos e incluso vislumbrar posibles relaciones entre distintos atributos de nuestros datos. Todo esto ofrece una ayuda sustancial a la hora de crear nuestro modelo y elegir los hiperparámetros óptimos para este.

Para ejemplificar lo comentado en el párrafo anterior, a continuación se muestra una gráfica con las longitudes de los datos de entrada y salida del dataset comentado en la sección 4.1:

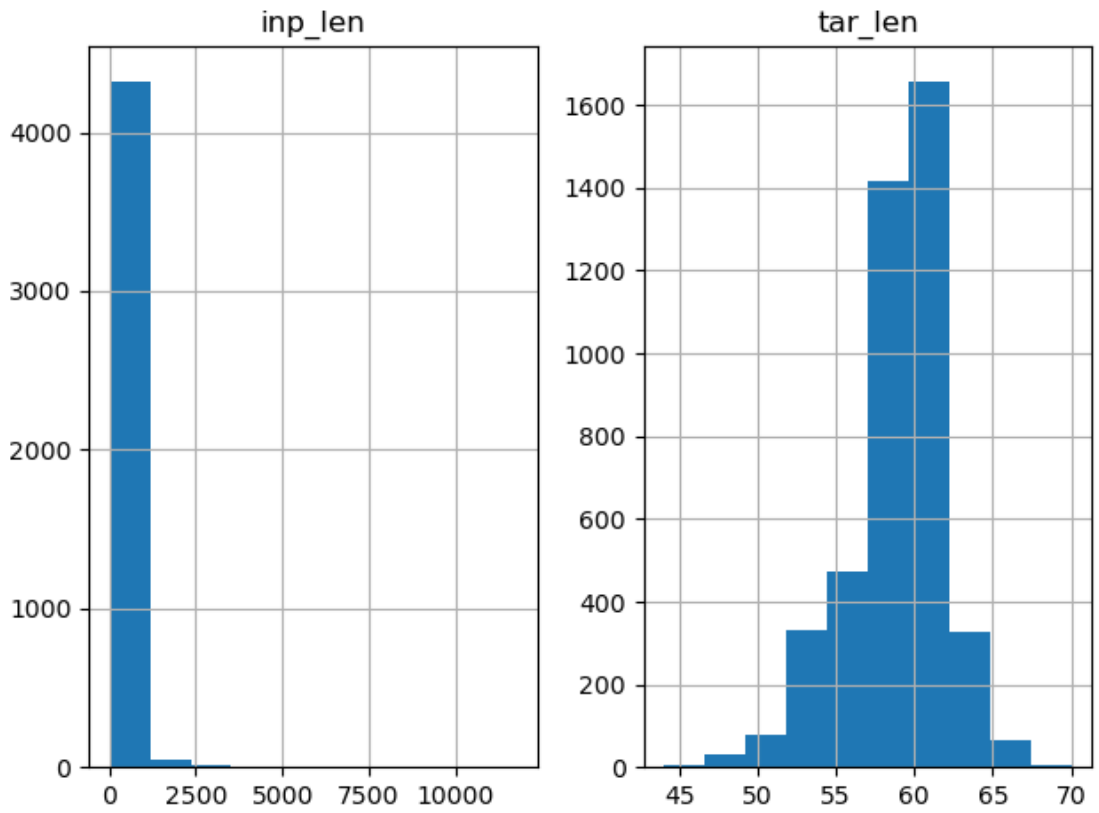


Figura 3.2: Longitudes de los datos de entrada y salida del dataset utilizado en este proyecto.

3.5 Gensim

Gensim es una librería diseñada para efectuar NLP sobre grandes cantidades de datos. Una de sus características más interesantes es la capacidad de crear *Word Embeddings* directamente desde algunos de sus objetos. Aunque estos elementos se explican detalladamente más adelante, por el momento basta saber que con su creación podemos obtener representaciones de palabras o tokens con información semántica, característica que puede mejorar mucho la calidad de nuestro modelo.

3.6 Tensorflow y Keras

En una primera instancia se pretendía explicar *Tensorflow* y *Keras* de manera individual pero consideramos más correcto presentarlos de manera conjunta por los motivos que se exponen más adelante.

3.6.1. Tensorflow

Tensorflow es una plataforma de Machine Learning de código abierto la cual ofrece una cantidad inmensa de funcionalidades para este fin, tanto de bajo como

de alto nivel. Está desarrollada por Google y cuenta con una amplia comunidad. Esta herramienta es muy versátil y flexible ya que permite tanto entrenar los modelos de DL como realizar inferencia con ellos.

A consecuencia de la gran cantidad de cálculos que necesitan efectuar los modelos durante su entrenamiento, *Tensorflow* está construido para poder ejecutarse sobre GPU, TPU e incluso de manera distribuida, ya sea utilizando distintos recursos del mismo dispositivo (dos tarjetas gráficas) o mediante la utilización de distintos nodos en la misma red.

Dada la versatilidad y potencia de esta herramienta, puede llegar a ser un poco abrumadora, sobretodo para los principiantes. Por eso las primeras guías que se recomiendan seguir en la propia pagina de Tensorflow no se adentran en *Tensorflow* puro sino que utilizan *Keras*.

3.6.2. Keras

Keras es un librería *wrapper* de redes neuronales que se puede ejecutar sobre diversas librerías back-end ofreciendo una interfaz común más sencilla y amena para realizar tareas de ML en un entorno de alto nivel. Esta biblioteca soporta *Tensorflow*, *Theano* y *CNTK* como back-ends, aunque la combinación más popular y la que se ha usado en este proyecto es *Keras + tensorflow*.

3.6.3. Tensorflow + Keras

Desde el año 2017 *Tensorflow* ofrece *Keras* como parte de su núcleo, es decir, al instalarse Tensorflow también se instala Keras. Esto se debe a dos motivos principalmente. Primero, la compatibilidad entre ambas librerías es total ya que cuando se libera una nueva versión, esta es estable y evita al usuario tener que dedicar tiempo a comprobaciones de compatibilidad entre librerías separadas. Segundo, dado que Keras presenta una mayor abstracción, por un lado permite a los usuarios principiantes la asimilación progresiva de las características de Tensorflow y, por otro, facilita y agiliza la construcción de modelos con topologías comunes y ofrece funcionalidades extra para el entrenamiento e inferencia con dichos modelos.

3.7 Hugging Face Transformers

Primero que todo es preciso aclarar la nomenclatura de esta herramienta puesto que es relativamente nueva y navegando en internet podemos encontrarnos que se usan los términos “Hugging Face” y “Transformers” indistintamente.

Huggin Face es el nombre que recibe la empresa de desarrolladores los cuales, además de otras librerías, han construido *Transformers*. No se ha de confundir esta biblioteca con la arquitectura homónima de modelos de redes neuronales basada en mecanismos de atención [5].

La característica más atractiva de esta biblioteca de alto nivel es que ofrece modelos neuronales del estado del arte ya preentrenados listos para su utiliza-

ción. Además permite ajustar los hiperparámetros de los modelos para su fácil manipulación o la extracción de alguna de sus capas para realizar *Transfer Learning*.

3.8 Py-rouge

Como indica su nombre, esta pequeña librería escrita en Python nos permite calcular la métrica *rouge*, que se subdivide en cuatro variantes: Rouge-N, Rouge-L, Rouge-W y Rouge-S; todas estas se explican detalladamente en la sección 4.8.1. Más concretamente permite efectuar una comparación uno a muchos por cada secuencia de texto a evaluar y, en este paradigma de uno a muchos, ofrece la opción de tomar el mejor resultado obtenido o realizar la media entre todos ellos. Es preciso comentar que en el repositorio de paquetes de Python (PyPI por sus siglas en inglés) podemos encontrar paquetes con nombres muy similares a este que realizan la misma función, por ello se presenta el enlace directo al paquete comentado.

3.9 Otras librerías

Acabamos de describir las herramientas más importantes para la tarea de resumir textos, ahora daremos una explicación superficial del uso de otras librerías para que la descripción de las tecnologías sea más completa.

Cabe destacar el uso de la biblioteca *Chardet*. Esta permite detectar la codificación de un texto, ya sea una cadena de caracteres o de bytes. Esto resulta sumamente útil cuando los datos recogidos para generar el dataset que vamos a emplear provienen de fuentes distintas. Un ejemplo de esto último sería la utilización de técnicas de *web scraping*.

Otra herramienta no imprescindible pero muy interesante y ventajosa es *Tqdm*. En el ámbito del ML se trabaja con grandes volúmenes de datos ya que los modelos necesitan una cantidad sustanciosa de ejemplos para aprender los patrones comunes y poder generalizar dicha información a casos nunca vistos. La gran mayoría de las veces no entrenamos los modelos directamente con el dataset que hemos descargado sino que este sufre un preprocesado antes de ser apto para el entrenamiento. Dada la magnitud de los datos, este proceso puede ser tedioso y no se sabe con exactitud cuando finalizará. La librería *Tqdm* se especializa en resolver este problema. Se puede acoplar con pequeñas modificaciones a los bucles en nuestro código para que muestre información como numero de iteraciones total, tiempo transcurrido, tiempo restante estimado, media de iteraciones por segundo, etc.

Por último nombraremos el uso de la biblioteca *re* (abreviación de expresiones regulares en inglés). Esta viene incorporada por defecto en Python y provee una gran cantidad de funciones para el tratamiento de cadenas. Permite buscar patrones y secuencias de texto para posteriormente reemplazarlas, eliminarlas y/o modificarlas. Un uso muy habitual de esta librería es la eliminación de las

stop words y de símbolos no deseados, una tarea muy común durante la etapa de preprocesamiento de los datos.

Cabe destacar que este proyecto no es una recopilación y concatenación de funciones importadas de librerías externas. Además de explorar esta aproximación también se han escrito métodos propios para realizar algunas de las funciones que ofrecen dichas librerías. Esto ha permitido comprender más profundamente el flujo de procesado de los datos y los factores a tener en cuenta al efectuarlo.

CAPÍTULO 4
Descripción del sistema

Una vez comentada la tecnología empleada procederemos a detallar el sistema y su configuración. Como se ha explicado anteriormente, el trabajo se ha desarrollado en un entorno multiplataforma por lo que se ha tenido que adaptar debidamente el código para funcionar independientemente del sistema operativo y su gestión de directorios.

Para solucionar este problema hemos aplicado dos medidas. Por un lado hemos utilizado rutas relativas en vez de absolutas. Por otro lado, para escribir los separadores de carpetas de las rutas, se obtiene en tiempo de ejecución el separador utilizado por el sistema: "\" para Windows y "/" para Linux y MAC. Utilizamos la instrucción "os.sep" para guardar el separador en una variable y utilizarla posteriormente.

Durante el desarrollo del código del proyecto se ha seguido el estilo y formato *pythonico*. Python provee unas guías y recomendaciones para que el estilo y formato de los programas escritos en este lenguaje sigan unas reglas comunes. Estas reglas se recogen en los llamados PEP (Python Enhancement Proposals).

La nomenclatura utilizada en el código ha sido en favor a la legibilidad y entendimiento del comportamiento de este. Únicamente se han utilizado comentarios donde así se ha requerido, evitando sobrecargar los métodos con texto redundante.

4.1 Dataset

El dataset utilizado en este proyecto es News Summary [14]. Este corpus consiste en dos archivos en formato CSV obtenidos mediante *Web Mining*. Es decir, haciendo uso de un script disponible en este enlace se han extraído noticias de una retahíla de paginas web hindúes.

Uno de estos archivos contiene 4514 elementos cuyos atributos son, en orden: autor, fecha, titulo, link, resumen y texto de la noticia. El otro arvhico contiene 98280 elementos constituidos por titulo y texto de la noticia. Cabe destacar que los archivos se encuentran mal formateados lo que implica un esfuerzo adicional en el proceso de carga de datos.

4.2 Carga de datos

A la hora de enfrentarse al problema de resumir textos, el primer paso a realizar en código es la carga de los datos. Este paso no es exclusivo de esta tarea sino que es común a todos los problemas de ML. El primer paso a realizar es identificar de que fuente vamos a extraer los datos, principalmente podemos distinguir entre dos vertientes: la carga desde fichero(s) o mediante una librería para tal fin. A continuación exponemos los procedimientos a seguir en cada una de ellas.

4.2.1. Carga desde fichero

Si los datos de nuestra tarea están almacenados en archivos, es preciso conocer de qué tipo de archivo se trata (es decir, que extensión tiene el fichero) para seleccionar una librería adecuada para su extracción. En casos reales de resumen de texto podemos encontrar una diversidad inmensa de tipos de archivos. A continuación nombramos los que consideramos más comunes y populares: CSV (Comma-Separated Values), JSON (JavaScript Object Notation), XLS (Microsoft Excel), XML (eXtensible Markup Language), SQL (Structured Query Language), TXT (texto plano).

En el caso particular de este proyecto los datos se encuentran almacenados en ficheros CSV. Aunque a continuación se comenten los procedimientos para cargar los datos a partir de este tipo de ficheros, los pasos a seguir son válidos para muchos otros tipos con pequeñas modificaciones. Para cargar los datos en el código se emplea la librería *pandas*, en concreto la función *read_csv*. Esta recibe la ruta del fichero csv a cargar (además de otros posibles parámetros) y devuelve un objeto con los datos de este. De igual manera podemos encontrar las funciones *read_json*, *read_excel*, *read_sql*, etc.

4.2.2. Carga mediante librería

A causa de la gran popularidad de algunos datasets, muchas de las librearías de ML implementan módulos para cargar estos de forma más fácil y cómoda. Incluso podemos encontrar bibliotecas creadas específicamente para este fin. Pero la popularidad no es el único motivo de la formación de estas librerías. Por un lado estas pueden ofrecer datasets ya preprocesados, evitándole así al usuario tener que hacerlo manualmente, una tarea que puede demorarse bastante dado el orden de magnitud de algunos datasets actuales [1]. Por otro lado se asegura la reproducibilidad del sistema (al menos en la carga de datos) ya que el código es determinista y no depende de la implementación específica de cada usuario de la carga de datos y el posible preprocesado.

Aunque el dataset que hemos utilizado en este proyecto está disponible únicamente mediante archivos por lo que no hemos hecho uso de librearías de carga de datasets, podemos comentar brevemente el procedimiento a realizar en el caso de estar disponible el dataset en dichas bibliotecas.

[1] https://www.kdnuggets.com/polls/2015/largest-dataset-analyzed-data-mined.html

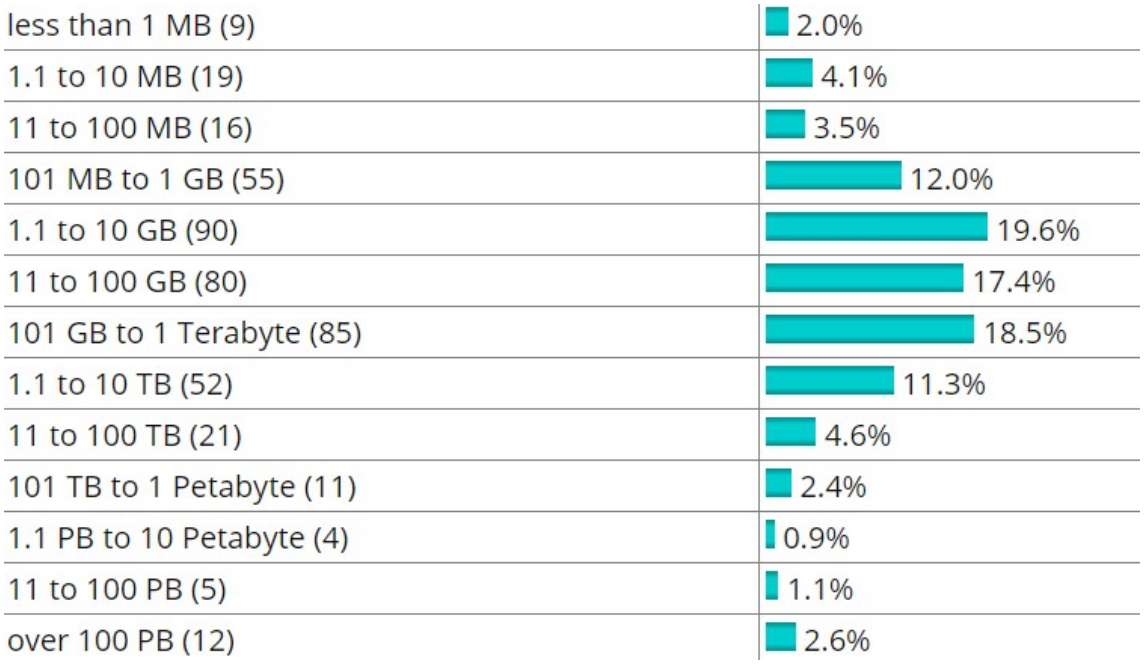


Figura 4.1: Tamaños de dataset.

4.2.3. Guardado de objetos para ahorrar tiempo

Adicionalmente creemos correcto comentar también en esta sección una técnica que se ha usado durante el desarrollo del trabajo y que surgió por necesidad. Como se ha comentado anteriormente, los datasets presentan tamaños considerables y preprocesar estos no es una tarea instantánea. Dado que la construcción de la solución a un problema es un proceso iterativo y por lo tanto se lanzan muchas ejecuciones del código durante la prueba, si en cada ejecución se preprocesan los datos antes de entrenar al modelo, se pierde una cantidad de tiempo abismal.

Por este motivo se decidió guardar en memoria no volátil los objetos que más tardaban en generarse para así agilizar el preprocesamiento, el cual pasó del orden de minutos al orden de segundos. En concreto el proceso que más tiempo consume es la generación de Word Embeddings. Para qué sirven y como se usan se explicará posteriormente, por el momento solo nos interesa destacar que tienen un alto coste computacional.

Aunque los Word Embedding dependen del vocabulario utilizado y este a su vez depende del filtrado de palabras que se haya realizado durante el procesamiento, una vez estabilizado este proceso, los Word Embeddings son invariantes y pueden reutilizarse entre ejecuciones. Para guardar estos objetos en disco duro se decidió utilizar la librería *pickle*. La metodología que ofrece es muy sencilla. Por un lado, para guardar el objeto se usa la función *dump*, que recibe la ruta del archivo donde se guardará y el objeto en sí. Por otro lado, para cargar el objeto se usa la función *load*, que recibe la ruta del fichero y devuelve el objeto que este contiene.

4.3 Preprocesamiento de datos

Es común pensar que la parte más importante de un problema de DL es la creación del modelo y el correcto ajuste de sus hiperparámetros pero se ha demostrado que la calidad de los datos influye al mismo nivel. El procesamiento de los datos que se van a utilizar para entrenar dicho modelo es una etapa crucial y no se debería omitir a la hora de enfrentarnos a ninguna tarea. Ponemos énfasis en este último comentario ya que en diversas ocasiones podemos encontrar trabajos faltantes de este proceso.

En internet hay muchas guías que prometen enseñarte a solventar una tarea de ML en 5 minutos. Esto es muy atractivo para los nuevos en el campo pero dista mucho de la realidad. Para no abrumar al lector principiante estas guías omiten etapas del problema como el preprocesamiento ya que el dataset utilizado esta preprocesado previamente. Esto permite una curva de aprendizaje más suave pero a veces crea mapa mental carente de tareas importantes como es el preprocesamiento de datos. Una vez aclarado este punto procedemos a explicar el tratamiento de datos previo al entrenamiento realizado en este proyecto.

4.3.1. Eliminación de valores nulos

El paso inmediatamente posterior a la carga del dataset con *pandas* es eliminar las filas donde haya valores nulos ya que omitir este paso deriva en dos posibles problemas. Primero, intentar tratar un valor nulo donde se esperaba una cadena de texto no vacía puede provocar un fallo en la ejecución del código. Segundo, si entrenamos un modelo con campos vacíos que no aportan nada de información, este tardará más en aprender o incluso si la cantidad de valores nulos es muy alta podría llevar al modelo a no converger nunca.

4.3.2. Conversión a minúsculas

Consecutivamente procedemos a pasar la totalidad del texto a letras minúsculas, esto es estrictamente necesario para evitar crear en el vocabulario más de una entrada por palabra. Si se omitiera este paso, podría darse el caso de que, por ejemplo, la palabras “perro” y “Perro” estén ambas incluidas en el vocabulario. Como en última instancia el texto es convertido a números, no interesa tener varias representaciones de la misma palabra.

4.3.3. Filtrado

Explorando manualmente los archivos que contienen el dataset observamos que hay cadenas de texto que no nos interesa conservar ya que no aportan información semántica. Hay varias aproximaciones para tratar este problema, unas son más agresivas y otras más laxas; el tamaño y contenido del vocabulario de nuestro sistema depende directamente de cual se elija. A continuación se listan las posibles cadenas a eliminar (o sustituir) del corpus.

- Caracteres en blanco: elementos tales como tabuladores, saltos de línea o más de un espacio se deben eliminar ya que pueden afectar negativamente en la creación del vocabulario.

- Texto en formato *Cascade Style Sheet* (CSS): debido a que para la generación del dataset se utilizaron técnicas de Web Scraping, podemos encontrar trozos de código CSS en algunas líneas de los archivos.

- Direcciones de paginas web: estas pueden aparecer por el mismo motivo que el código CSS o porque las propias noticias las incluyan.

- Direcciones de correo: consideramos necesario eliminar o reemplazar estos elementos por dos razones; primero, no aportan información semántica ni sintáctica y, segundo, evitamos sesgos o aparición de datos privados (dependiendo de la procedencia del dataset).

- Hashtags y menciones de Twitter: aunque estos elementos sí podrían aportar información semántica, el hecho de que no se separen las palabras con un espacio llevaría al programa a considerarlas como una palabra nueva en vez de una secuencia de las ya existentes. Esto provocaría un aumento innecesario del vocabulario.

- Números: la cantidad de números distintos en un corpus puede ser notoria, más aún si tenemos en cuenta que "1" y "1.0" son tokens distintos a efectos prácticos. Esto deriva en un aumento drástico del tamaño del vocabulario por lo cual conviene eliminarlos o sustituirlos por un token común.

- Signos de puntuación: La eliminación de estos depende considerablemente de la tarea a realizar ya que ofrecen una información muy valiosa gramaticalmente.

- Caracteres raros: un efecto secundario de la extracción de datos usando Web Scraping es que los textos pueden presentar distintas codificaciones. Si en el momento de extracción no es identifican correctamente y se convierten a un formato común, por ejemplo UTF-8, se pueden generar cadenas con caracteres raros.

4.3.4. Tokenización

Además del filtrado del texto también es recomendable realizar otros procesamientos para que los datos entregados al modelo sean de mayor calidad. El siguiente paso a realizar es la tokenización. Este proceso traduce una cadena de texto a una lista de las palabras que este contiene. La aproximación más simple para ejecutar este proceso es dividir (*split*) el texto por espacios, esto puede provocar efectos no deseados al considerar como palabras distintas una misma palabra con o sin prefijo/sufijo. Ejemplo en inglés (idioma de nuestro dataset): *they — they'll*. Ejemplo en castellano: *(tu) entrega — (tu) entrégalo*.

4.3.5. Agrupación de términos

A continuación procedemos a aplicar técnicas de stemming/lematización. Estas convierten palabras con morfología similar (stemming) o con significado similar (lematización) en un único token. De esta manera conseguimos reducir considerablemente el tamaño de vocabulario sin apenas perder información.

Por una parte, el stemming reduce los términos a seudoraíces (raíces no necesariamente iguales a sus raíces morfológicas). Existen diversos algoritmos para realizar esta tarea pero el más popular y ampliamente usado es el algoritmo de Porter [15]. Este se basa en la eliminación de sufijos comunes. Este método es más rápido pero menos preciso ya que palabras con significados distintos pueden converger en la misma seudoraíz.

Por otra parte, la lematización es el proceso por el cual las palabras se transforman en su unidad semántica constituyente, es decir, en su lema. Esta unidad es representativa de todo su grupo de palabras. Aunque este método es más exigente computacionalmente, ofrece mejores resultados que el stemming.

4.3.6. Etiquetado de límites de frase

En tareas que implican la generación de textos, es necesario indicar explícitamente donde empieza y donde acaba cada frase. Para esto se añaden unos tokens especiales al principio y al final de cada frase, por ejemplo: "*_START_*" y "*_END_*" respectivamente. El modelo al entrenarse los tratará como una palabra más del vocabulario y aprenderá que todas las oraciones empiezan por "*_START_*" y terminan con "*_END_*".

Este paso es importante ya que una vez tengamos el modelo entrenado, para realizar la inferencia necesitamos introducir un token inicial a partir del cual empiece a generar la frase. Si no dispusiéramos del token de inicio deberíamos elegir un token inicial no trivial y estaríamos introduciendo un sesgo en la inferencia. De igual forma, si el vocabulario no contara con un token de finalización, el modelo no tendría la capacidad de indicar que la frase que estaba construyendo ha finalizado.

4.3.7. Resumen

Se expone a continuación un diagrama para generar una imagen mental del flujo de procesamiento de los datos.

4.4 Representación de los datos

Lo primero que debemos elegir es cual va a ser nuestra unidad mínima de información. Se puede trabajar a nivel de caracteres o a nivel de palabra e, independientemente de esta elección, podemos generar representaciones término a término o agrupar varios en n-gramas. En general no existe una aproximación mejor, por lo tanto esta elección va a depender de cada dataset y tarea a realizar,

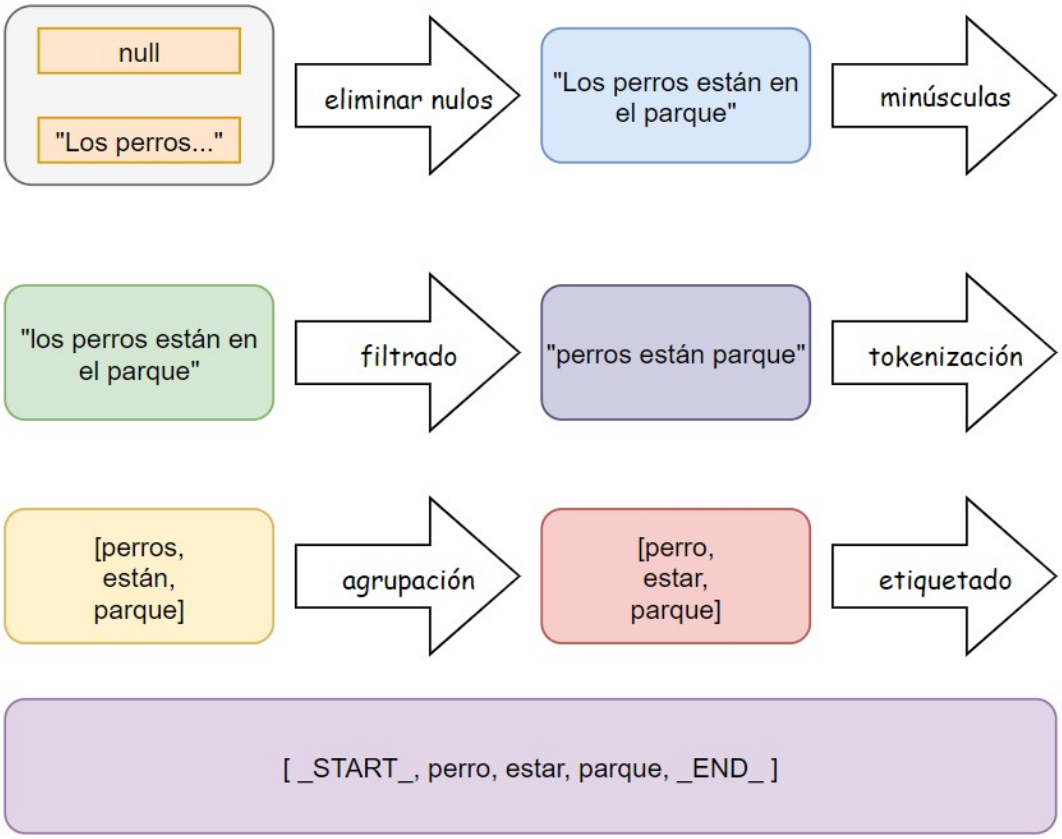


Figura 4.2: Flujo de preprocesado.

debiéndose realizar en cada caso distintas pruebas para averiguar qué unidad de información resulta más conveniente.

Dado que los ordenadores no entienden de palabras y texto sino de números, la siguiente tarea a realizar es convertir la unidades mencionadas en el párrafo anterior a representaciones numéricas. Para esta transformación existen diversas soluciones pero las más comúnmente utilizadas son: *One Hot Encoding*, *Bag of Words* y *Word Embeddings*. A continuación se ofrece una explicación detallada de cada uno de ellos para posteriormente compararlos y comentar cual se debería utilizar según la tarea a realizar y el tamaño de nuestro dataset.

4.4.1. One Hot Encoding

Esta técnica básicamente crea un vector del tamaño del vocabulario por cada palabra. Este tendrá todas las posiciones establecidas a 0 excepto aquella asociada a la palabra que se esté representado, la cual se fijará a 1. Este mecanismo puede servirnos cuando nuestro vocabulario no es muy extenso pero no es muy escalable ya que a medida que dicho vocabulario va creciendo nos encontraremos con vectores enormes llenos de ceros. Al generarse un vector por palabra *One Hot Encoding* mantiene el orden de la secuencia.

4.4.2. Bag of words

Esta aproximación es parecida a *One Hot Encoding*, también se crea un vector del tamaño del vocabulario con cada palabra asociada a una posición. Pero esta vez solo se genera uno por secuencia y no por cada palabra. Existen 2 tipos de representaciones *Bag of Words*: binaria y entera. Por una parte, en la representación binaria, cada palabra que aparece en la secuencia se anota en el vector con un 1 en la posición asociada a dicha palabra, siendo el resto ceros. Por otra parte, en la representación entera se cuenta la cantidad de veces que ha aparecido cada palabra en la secuencia y se escribe tal cantidad en la posición asociada a cada palabra, siendo el resto ceros.

4.4.3. Word Embeddings

Los *Word Embeddings* no solo representan el orden de las palabras al igual que los *One Hot Vectors* sino que también incluyen información semántica. Esto se debe al proceso de su creación. Para generar un *Word Embedding* se crea un pequeño modelo del lenguaje el cual aprende bien a predecir una palabra dadas las palabras adyacentes, bien a predecir las palabras adyacentes dada la palabra en cuestión. Una vez entrenado el modelo con todas las secuencias se elimina la última capa para que este devuelva un vector de valores decimales los cuales (si el entrenamiento ha sido fructífero) representan una palabra junto con su contexto.

4.4.4. Comparación

La elección de una u otra depende del tipo y de la cantidad de datos del dataset en cuestión al igual que la tarea a realizar. Aunque este trabajo se basa en la tarea de resumir textos también se ha explicado *Bag of Words* por ser ampliamente utilizado en tareas similares del campo del NLP.

Bag of Words únicamente representa la aparición de las palabras en una secuencia, no mantiene ninguna información sobre su orden. Por un lado, esto provoca que sea útil para tareas como clasificación de documentos ya que generalmente solo es necesario saber si determinadas palabras aparecen en él o no pero no el orden en el que estas aparecen. Por otro lado, esta misma característica deriva en que esta técnica sea poco ventajosa en problemas de comprensión de texto como es la traducción o el resumen.

En lo que al dataset respecta, si el tamaño de este es pequeño (y por ende el de nuestro vocabulario), conviene utilizar *One Hot Encoding* ya que es sencillo y suficiente para representar dichos datos. En este caso no sería adecuado el uso de *Word Embeddings* ya que con pocos datos de entrenamiento el modelo de lenguaje no podria capturar bien las relaciones entre palabras. En otro orden de ideas, si disponemos de un dataset de tamaño mediano o grande (a partir de unas decenas de miles de ejemplos), es conveniente utilizar *Word Embeddings* ya que con estos son capaces de representar la información semántica de las palabras si disponen de suficientes datos para entrenar.

4.5 Creación de modelos

En la literatura podemos encontrar una amplia variedad de topologías disponibles para afrontar los problemas de DL y el caso de resumir textos no es distinto. Como se ha comentado en el capítulo 2, existen arquitecturas como los transformers que ofrecen resultados asombrosos, pero para poder manejarlas primero debemos familiarizarnos con aproximaciones más sencillas. De esta manera entenderemos que problemas encontrábamos previamente y cómo la aparición de estas topologías supuso un gran avance en el campo.

Cabe destacar que el problema se ha abordado desde distintos niveles de abstracción. Para las arquitecturas más sencillas se han creado modelos manualmente mientras que para las más complejas se ha hecho uso de una librería de alto nivel que permite utilizar modelos ya entrenados. De esta manera conseguimos por un lado profundizar en las bases del campo y por otro exponemos un mayor número de arquitecturas para dar a conocer su existencia y que el lector pueda indagar en más profundidad en aquellas que le resulten interesantes.

Antes de proceder es preciso dar una definición general de ciertos términos además de explicar de qué manera estos están relacionados.

Un **modelo de ML (supervisado)** es una estructura de pesos, comúnmente llamados parámetros, los cuales se actualizan mediante una fase de entrenamiento. Esta fase consiste en introducir un valor en el modelo del cual sabemos su salida óptima asociada, observar cuan parecida es con la salida del modelo (el error) y posteriormente modificar los parámetros para minimizar el error en el hipotético caso de que se repita dicha entrada.

4.5.1. Modelos manuales

Para la creación de arquitecturas arbitrarias se han utilizado las librerías Tensorflow y su *wrapper* Keras. Tensorflow provee funciones para realizar y concatenar operaciones en el proceso de la creación de los modelos mientras que Keras crea un nivel de abstracción superior mediante el uso de capas. Adicionalmente ofrece clases para crear objetos de tipo *modelo* que aportan métodos para realizar entrenamiento y predicción de manera más estructurada.

Una capa es una estructura que agrupa una operación o conjunto de operaciones a realizar además de otros parámetros necesarios para la ejecución de estas, como la forma de los datos de entrada o la función de activación empleada. Antes de describir la topología de los modelos se exponen las capas de las cuales se componen. Esta lista no pretende abarcar la totalidad de capas disponibles sino dar una visión superficial de las más empleadas en la resolución de esta tarea. Las capas están listadas según el orden de aparición que suelen tener en los modelos.

- **Input:** La función de esta capa es crear un punto de entrada al modelo, en ella puede indicarse la dimensión de los datos de entrada aunque no es obligatorio.
- **Embedding:** Esta capa transforma las palabras de entrada en representaciones vectoriales que recogen las relaciones que hay entre ellas.

- **LSTM:** Esta capa implementa una memoria a corto y largo plazo (Long Short Term Memory), esto le permite al modelo capturar relaciones tanto entre palabras cercanas como distantes.
- **GRU:** Esta capa actúa de una forma muy similar a la LSTM pero ofrece una mayor eficiencia [3].
- **Bidirectional:** Es una capa envoltorio la cual indica que la capa que contiene recibe información tanto de las palabras anteriores como de las siguientes dentro de la secuencia.
- **Dense:** Esta capa consta de n neuronas donde n es un parámetro obligatorio en la construcción de esta. En los problemas de clasificación, n es número de clases posibles, en este caso en concreto, n es el tamaño de vocabulario. Todas la neuronas están conectadas a todas las salidas de la capa anterior. Al ser un problema de clasificación se utiliza *softmax* como función de activación.
- **TimeDistributed:** Esta capa también es una capa envoltorio la cual se utiliza para agrupar secuencias de salidas en un solo resultado como es el caso de nuestros modelos.

Cabe destacar que no se han podido probar todas las arquitecturas deseadas. Dada la gran dimensionalidad de los datos (secuencias del orden de 400 palabras), aunque los recursos utilizados a priori parecían bastar (tarjetas gráficas con 8GB de memoria RAM), en el momento de añadir más de 2 capas recurrentes el sistema necesitaba más recursos de los existentes.

Aunque se han probado distintas variaciones, por norma general todos los modelos manuales creados describen una arquitectura similar. Esta arquitectura se compone de dos módulos, el *encoder* y el *decoder*. Por un lado, el *encoder* consta de una capa de entrada (Input), una capa de representación de los datos (Embedding) y una o más capas recurrentes (LSTM). Por otro lado, el *decoder* consta de una capa de entrada (Input), una capa de representación de los datos (Embedding), una o más capas recurrentes (LSTM) y una capa de salida (Dense). Todas las capas recurrentes pueden ser bidireccionales (Bidirectional) y la capa de salida se envuelve en una capa distribuida en el tiempo (TimeDistributed) por la forma de los vectores de salida de la capa recurrente del *decoder*.

Son diversas las variaciones realizadas en las distintas pruebas ejecutadas, a continuación se listan estas detallando de qué manera pueden afectar a nuestro modelo en términos de tamaño y calidad. Como se ha comentado anteriormente, la elección de la mejor configuración dependerá del dataset y de la tarea a realizar.

Retomando lo explicado en la sección 4.4, se ha probado a trabajar tanto a nivel de carácter como a nivel de palabra. A nivel de carácter siempre se han agrupado en tri-gramas dado que si se trabajara con un único término el modelo perdería capacidad de generalización al ser el vocabulario del orden de decenas (letras del abecedario + signos de puntuación permitidos en el procesamiento). A nivel de palabras se ha elegido trabajar con mono-gramas (sin agrupaciones de términos) dado que el dataset no es lo suficientemente grande como para que palabras consecutivas se repitan un número significativo de veces.

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 500)]	0	
input_2 (InputLayer)	[(None, 79)]	0	
embedding (Embedding)	(None, 500, 200)	4000000	input_1[0][0]
embedding_1 (Embedding)	(None, 79, 200)	4000000	input_2[0][0]
lstm (LSTM)	[(None, 500, 200), (	320800	embedding[0][0]
lstm_1 (LSTM)	[(None, 79, 200), (N	320800	embedding_1[0][0] lstm[0][1] lstm[0][2]
time_distributed (TimeDistribut	(None, 79, 20000)	4020000	lstm_1[0][0]

Total params: 12,661,600
Trainable params: 12,661,600
Non-trainable params: 0

Figura 4.3: Ejemplo de la arquitectura común a los modelos.

En este proyecto se ha decidido utilizar *Word Embeddings* como mecanismo de representación ya que el dataset es suficientemente extenso como para capturar la información semántica del contexto de las palabras. Cuando se utiliza este tipo de representación debemos elegir la dimensión de los vectores que creará el modelo de lenguaje. No existe una manera trivial para seleccionar un valor adecuado, mediante prueba y error se debe encontrar un valor suficientemente alto para poder representar la totalidad de las palabras del vocabulario y a su vez suficientemente bajo para no agotar innecesariamente los recursos disponibles ya que se generará un vector de dicha dimensión por cada palabra de cada secuencia.

Complementariamente a lo comentado en el párrafo anterior es preciso exponer que los *Word Embeddings* se pueden aprender en tiempo de entrenamiento de nuestro modelo o pueden crearse asíncronamente mediante librerías como Gensim. En este último caso, disponemos de dos variantes: entrenar el *Word Embedding* con los datos de nuestro dataset o cargar uno ya entrenado y especializado en el idioma de nuestros datos. En nuestro caso en concreto hemos encontrado acertado una dimensionalidad de 300.

En otro orden de ideas, se puede experimentar con el número de capas recurrentes y el número de neuronas de cada capa para encontrar un compromiso entre la capacidad del modelo de capturar las dependencias entre palabras para generalizar mejor y el coste espacial de dicho modelo. Esta consideración es importante ya que podemos exceder los recursos del sistema con bastante facilidad, como se ha comentado anteriormente. En nuestro caso hemos utilizado entre una y dos capas LSTM con entre 100 y 500 neuronas por capa.

Otro factor a tener en cuenta es el tamaño de vocabulario. Este afecta directamente a la calidad del modelo. Por un lado, un vocabulario muy grande implica aumentar la complejidad del modelo sin mejorar resultados, o incluso empeorándolo. Por otro lado, uno demasiado pequeño perderá capacidad de generalización y habrá demasiadas palabras fuera del vocabulario.

4.5.2. Modelos preentrenados

En este apartado se explica como, mediante la librería transformers, se pueden cargar distintos modelos ya preentrenados para realizar inferencia directamente sin necesidad de entrenarlos. Esta aproximación permite probar las arquitecturas que ocupan el estado del arte sin necesidad de disponer de los recursos requeridos para entrenarlas. Ya que en algunas de ellas se ha empleado una cantidad de hardware órdenes de magnitud superior que el disponible por un usuario estándar.

La biblioteca transformers ofrece una interfaz con distintos niveles de abstracción. En el nivel más alto encontramos los pipelines, elementos que con tres simples líneas de código permiten realizar inferencia en la tarea indicada. A continuación se presenta una imagen donde se ejemplifica el uso más simple de una pipeline: en la primera línea se define una variable en la que se guarda un texto de ejemplo, en un caso real este sería el texto a resumir; en la segunda línea se instancia un objeto pipeline indicando la tarea que deseamos realizar, en este caso resumir; por último, en la tercera línea se hace uso del objeto declarado en la línea anterior para resumir el texto guardado en la variable *texto* indicando que se desea un resumen de longitud mínima 30 y longitud máxima 130, el resultado se muestra por pantalla gracias a la función *print*.

```
text = 'Texto a resumir'

summarizer = pipeline('summarization')

print(summarizer(text, max_length=130, min_length=30))
```

Figura 4.4: Ejemplo simple del uso de *pipelines* con la librería Transformers.

Internamente, una *pipeline*, si no se le especifica un modelo como parámetro, carga por defecto un modelo preentrenado para la tarea indicada. Esto es lo sucedido en el caso anterior, más concretamente la *pipeline* cargará el modelo T5 según la documentación. Pero esta no es la única manera de utilizar la librería, adicionalmente tenemos la posibilidad de cargar manualmente el modelo que deseemos de los disponibles por la biblioteca.

4.5.3. Modelos híbridos

Seguidamente se expone una aproximación intermedia entre montar un modelo manualmente y cargar uno ya entrenado para usarlo directamente en inferencia. Para realizar este proceso también se hará uso de la librería Transformers, esta permite cargar algunas capas de los modelos preentrenados para después incluirlas en nuestro modelo. Poder cargar parte de los modelos que consiguen los mejores resultados de la literatura permite crear nuevos modelos cercanos al estado del arte sin depender te tener una gran capacidad de computo.

El hecho de cargar únicamente parte de los modelos provoca que la salida de las capas cargadas no sea un resultado como tal sino que podemos interpretar

este proceso como una manera de realizar *feature extraction*. Esta técnica se utiliza para representar la información de forma más compacta y no redundante, está intrínsecamente relacionada con los mecanismos de reducción de la dimensionalidad. En este domino se encuentran algoritmos como Principal Component Analisis (PCA), Independent Component Analisis (ICA) o los Autoencoders.

Si prestamos un poco de atención nos damos cuenta de que este método es muy similar al comentado sobre los *Embeddings*, en ambos se introduce una secuencia de longitud arbitraria y se obtiene una secuencia de longitud fija que idealmente representa las características más importantes y distintivas de la secuencia inicial.

4.6 Entrenamiento

Para entrenar los modelos creados debemos tener en cuenta varios factores. Uno de los más importantes es la estricta coincidencia entre las dimensiones de los datos preprocesados y las del modelo. Parece algo obvio pero en la práctica es una de las causas más comunes de error en nuestro código. A continuación se comentan los aspectos que creemos que requieren una atención especial para ahorrar tiempo y esfuerzo a la hora de entrenar nuestro modelo.

Entrenar un modelo es una tarea muy exigente computacionalmente, por ello se busca agilizar este proceso mediante distintas técnicas. Un muy destacada y quizá la más ampliamente utilizada es el entrenamiento por lotes. Esta técnica consiste en "duplicar" nuestro modelo tantas veces como el tamaño de lote elegido (n en adelante), procesar n secuencias a la vez para obtener sus errores mediante la función de pérdida y condensarlos para realizar una única actualización de los pesos del modelo.

Esta optimización no tiene ninguna repercusión negativa en los modelos *feed-forward* pero al aplicarla en modelos con redes recurrentes surge un inconveniente: **todas las secuencias de un mismo lote deben tener la misma longitud.** Esto puede resultar un poco contraintuitivo ya que la característica principal de las redes recurrentes es poder procesar secuencias de longitud arbitraria pero aún se puede gozar de esta cualidad ya que la longitud de las secuencias de distintos lotes es independiente de la del resto.

Por esta razón, si queremos aprovechar la técnica del entrenamiento por lotes debemos transformar convenientemente nuestros datos. Existen diversas aproximaciones pero la más común y la utilizada en este proyecto consiste en extender o truncar todas las secuencias a la misma longitud. Usualmente las librerías de ML ofrecen funciones para realizar este proceso cómodamente. En nuestro caso se ha utilizado la función *pad_sequences* de Keras.

En otro orden de ideas, el preprocesado de datos a veces no es la última transformación que reciben estos antes de usarse en el entrenamiento. En nuestro caso el preprocesado resultaba en un conjunto de frases de entrada (noticias) y un conjunto de frases de salida (resúmenes) pero como se ha comentado en la sección 4.5.1, la arquitectura de nuestros modelos consta de dos entradas y una salida. Por lo tanto se necesita adaptar los datos para que sean compatibles con el modelo.

En nuestro caso se desea que el modelo prediga la palabra siguiente dado el conjunto de palabras anteriores por lo que la transformación realizada es la siguiente: se duplica el conjunto de secuencias de salida (sal-1 y sal-2 en adelante); del conjunto sal-1 se elimina el último término de cada secuencia y del conjunto sal-2 se elimina el primer término de cada secuencia. Por consiguiente ya disponemos de los tres conjuntos adaptados necesarios para entrenar nuestro modelo. La entrada del *encoder* recibe las noticias, la entrada del *decoder* recibe los resúmenes con los términos finales suprimidos y la salida hace uso de los resúmenes con los términos iniciales suprimidos.

Haciendo hincapié en la importancia de las dimensiones, podemos observar como la adaptación de los datos comentada en el párrafo anterior implica una modificación obligatoria en los hiperparámetros del modelo, en concreto en la dimensión de entrada del *decoder*. Aunque las secuencias hayan sido expandidas o truncadas a una longitud n para aprovechar el entrenamiento por lotes, en nuestro modelo deberemos utilizar $n - 1$. Asimismo, también debemos prestar especial atención cuando se utilicen capas bidireccionales dado que la salida de esta capa será el doble del valor indicado en la capa que envuelve.

Retomamos el tema del entrenamiento por lotes en redes recurrentes para explicar un aspecto a tener en cuenta cuando transformamos las secuencias a una longitud común a todas. En el caso de las secuencias largas no encontramos problema porque truncamos esta a la longitud deseada sin mayor complicación. Pero en el caso de las secuencias menores que la longitud indicada se necesita extender estas mediante algún mecanismo.

Lo mas común es reservar el valor 0 en el vocabulario para esta tarea. En consecuencia las secuencias menores que la longitud deseada n se rellenarán con $n - 1$ ceros para alcanzar el tamaño indicado. Esto por un lado soluciona el problema de la longitud pero genera otro, el modelo no entiende que ese valor es un comodín para rellenar los huecos. Por esta razón se debe de indicar esto explícitamente en la creación de los modelos. Para realizar dicha tarea la librerías ofrecen distintas opciones. En la capa Embedding de Keras podemos pasarle el parámetro *mask_zero=True*.

4.7 Inferencia

En la etapa de inferencia utilizamos datos nunca vistos por el modelo para evaluar cómo de bien ha aprendido a generalizar a partir de los datos de entrenamiento. Es por ello que el hecho de preprocesar los datos introducidos en la inferencia igual que los datos de entrenamiento es crucial.

En los modelos *feedforward* para producir un resultado completo solo se necesita realizar una predicción. Pero en el caso de los modelos secuencia-a-secuencia no sucede lo mismo. Para generar un resultado completo, por ejemplo un resumen, necesitamos de un número arbitrario de iteraciones del modelo. A este se le introduce un ya mencionado token de inicio y a partir de ese token y el contexto recibido del *encoder* se genera la frase de salida paso a paso.

Adicionalmente el modelo necesita un mecanismo para terminar las secuencias, en caso contrario este permanecería realizando predicciones indefinidamen-

te. Para solucionar este problema se implementan dos métodos. Por un lado se añade al final de toda secuencia un token de parada el cual el modelo aprende en entrenamiento y si lo produce en inferencia se termina la frase generada. Por otro lado, se puede establecer un límite manualmente en el código destinado a la inferencia para que, independientemente del modelo, las secuencias tengan una longitud máxima.

4.7.1. Algoritmo voraz

Existen distintos algoritmos para realizar inferencia, el más sencillo y ampliamente utilizado es la solución trivial. Es decir, en cada iteración se escoge la palabra que presenta mayor probabilidad de ser la siguiente en la secuencia. Este algoritmo es el más eficiente pero es muy propenso a la repetición.

4.7.2. Beam Search

Otro algoritmo mucho más potente y robusto es *Beam Search*. Este consiste en mantener un haz de secuencias paralelamente en la etapa de inferencia. Más concretamente, este mecanismo recibe dos parámetros. El primero indica el tamaño de haz, es decir, en cada iteración se mantiene un conjunto de posibles resultados y se expanden aquellos que ocupan los n mejores posiciones. El segundo indica, por cada secuencia, cuántas posibles secuencias se generan a partir de la primera.

4.8 Evaluación

Como se ha comentado anteriormente, los modelos que generan texto necesitan de varias iteraciones para obtener un resultado completo. Esto provoca que el *loss* y el *accuracy* no reflejen exactamente qué tan bien aprende nuestro modelo. Esto se deben a que esta métricas evalúan una única iteración de forma aislada y no la secuencia entera generada. Por ello en la literatura encontramos otro tipo de métricas especiales para suplir las carencias de las primeras. A continuación describimos dos de ellas.

4.8.1. ROUGE

Recall-Oriented Understudy for Gisting Evaluation (ROUGE) [16] sirve para evaluar automáticamente los resúmenes generados por un modelo comparándolos con los "idelaes" escritos por humanos. Esta métrica mide las unidades coincidentes como n-gramas y secuencias o pares de palabras entre las palabras de la predicción del modelo y el resumen real. ROUGE presenta 4 distintas medidas: ROUGE-N, ROUGE-L, ROUGE-W y ROUGE-S.

4.8.2. BLEU

Esta métrica, BiLingual Evaluation Understudy (BLEU) [13] utiliza distintos mecanismos de coincidencia de n-gramas entre las secuencias predichas por el modelo y las disponibles en el dataset. El hecho de utilizar n-gramas implica la elección de palabras o el orden de estas no importa, esta es una característica deseada ya que existen diferentes maneras de transmitir la misma información.

CAPÍTULO 5
Experimentación y resultados

Como se ha comentado en la sección 4.1, disponemos de dos tipos de resúmenes, unos más extensos (resúmenes como tal) y otros más concisos (títulos de las noticias). Esto provoca que dependiendo de qué tipos de resúmenes se elijan para entrenar nuestros modelos, la dificultad que encontremos al resolver la tarea varíe. No solo por que obviamente cuanto más corta sea la secuencia a predecir más fácil le resultará al modelo aprender, sino porque la cantidad de ejemplos disponibles de los resúmenes tipo título es mucho mayor. Claramente cuantos más ejemplos distintos se empleen para entrenar los modelos, más posibilidades tendrá de generalizar mejor otros datos nunca vistos. Se recalca la palabra posibilidad ya que más datos no siempre significa mejores resultados.

Partiendo de lo comentado en el párrafo anterior y puesto que un titular se puede considerar un resumen de longitud mínima, se ha abordado el problema desde ambas aproximaciones, tanto para predecir resúmenes como para predecir títulos. La metodología empleada para abordar simultáneamente ambas variaciones ha sido crear un modelo con distintos hiperparámetros por cada topología que se deseara probar. Es decir, las dos aproximaciones comparten una arquitectura común por cada prueba pero en cada instancia de los modelos se cambiaban los parámetros relativos a las longitudes de los datos.

Cabe destacar que el desarrollo de este trabajo ha sido un proceso iterativo en el que el alumno ha ido adquiriendo conocimientos a medida que se realizaban los experimentos, por ello algunos resultados podrían no ser rigurosamente exactos debido a errores metodológicos cometidos en las primeras etapas. Asimismo, es preciso mencionar que la intención principal ha sido **explorar** las distintas posibilidades y variantes que ofrecía cada etapa del proceso y no la de **explotar** estas para conseguir los mejores resultados.

5.1 Modelos manuales

A continuación se comentan las propiedades que comparten todos los modelos creados manualmente para después mencionar qué cambios presenta cada uno y como ha podido afectar este a los resultados.

Todos los modelos, ya sean para generar resúmenes completos o para generar únicamente el titular de la noticia, todos los modelos presentan una arquitectu-

ra *encoder-decoder*. Esto es, todos ellos primero condensan los datos de entrada de longitud arbitraria en un vector contexto de tamaño predefinido (*encoder*) y segundo, mediante el vector contexto junto con los términos generados anteriormente se predice el siguiente término de la secuencia de salida (*decoder*).

5.1.1. Generación de resúmenes

El primer modelo consta de, tanto para el *encoder* como para el *decoder*, una capa Input, una campa Embedding y una capa LSTM. El segundo difiere del primero porque posterior a la capa LSTM se añade una capa Dense envuelta en una TimeDistributed para la salida.

generación de resúmenes				generación de titulares			
entrenamiento		validación		entrenamiento		validación	
loss	acc	loss	acc	loss	acc	loss	acc
3.6096	0.2609	4.5627	0.1999	1.0899	0.6509	1.3697	0.6033

Tabla 5.1: Pérdida y precisión del modelo 1.

El segundo modelo copia la arquitectura del primero pero introduce una capa LSTM adicional en el *decoder*. Esto muestra que la adición de esta capa recurrente reduce el *overfitting*. Los parámetros de las capas se muestran en la siguiente imagen.

```
Layer (type)                 Output Shape              Param #   Connected to
==================================================================================================
input_1 (InputLayer)         [(None, 550)]             0
__________________________________________________________________________________________________
input_2 (InputLayer)         [(None, 79)]              0
__________________________________________________________________________________________________
embedding (Embedding)        (None, 550, 300)          3000000   input_1[0][0]
__________________________________________________________________________________________________
embedding_1 (Embedding)      (None, 79, 300)           3000000   input_2[0][0]
__________________________________________________________________________________________________
lstm (LSTM)                  [(None, 300), (None,      721200    embedding[0][0]
__________________________________________________________________________________________________
lstm_1 (LSTM)                [(None, 79, 300), (N      721200    embedding_1[0][0]
                                                                 lstm[0][1]
                                                                 lstm[0][2]
__________________________________________________________________________________________________
lstm_2 (LSTM)                [(None, 79, 300), (N      721200    embedding_1[0][0]
                                                                 lstm_1[0][1]
                                                                 lstm_1[0][2]
__________________________________________________________________________________________________
time_distributed (TimeDistribut (None, 79, 10000)      3010000   lstm_2[0][0]
==================================================================================================
Total params: 11,173,600
Trainable params: 11,173,600
Non-trainable params: 0
```

Figura 5.1: Ejemplo simple del uso de *pipelines* con la librería Transformers.

Adicionalmente se decidió hacer un procesado de los datos más agresivo sin cambiar los parámetros del modelo y esto derivó en una ligera mejora de los resultados.

A la vista de los resultados podemos afirmar que los modelos sufren de *overfitting* ya que la diferencia entre los valores de entrenamiento y validación es sig-

generación de resúmenes				generación de titulares			
entrenamiento		validación		entrenamiento		validación	
loss	acc	loss	acc	loss	acc	loss	acc
3.1465	0.2599	4.0392	0.2090	1.1244	0.6521	1.3659	0.6099

Tabla 5.2: Pérdida y precisión del modelo 2.

generación de resúmenes				generación de titulares			
entrenamiento		validación		entrenamiento		validación	
loss	acc	loss	acc	loss	acc	loss	acc
3.6717	0.2738	4.6487	0.2135	1.0455	0.6988	1.3556	0.6138

Tabla 5.3: Pérdida y precisión del modelo 3.

nificativa. Por un lado, los modelos destinados a generar resúmenes rondan el 20 % de precisión mientras que los destinados a generar tirulares rondan el 60 %. Por otra parte se aprecia que la inclusión de una capa LSTM adicional mejora ligeramente el modelo a la vez que reduce el *overffiting*.

5.2 Modelos preentrenados

En esta sección no nos extenderemos en demasía ya que estos modelos son modelos externos utilizados únicamente para realizar inferencia y sin ninguna modificación por el alumno. Por ello se ha decidido ejemplificar esta aproximación con un solo modelo y citar el resto de los modelos que ofrece la librería Transformers.

5.3 Modelos híbridos

Este enfoque es el menos explorado ya que se ha creado un modelo y no se ha conseguido realizar inferencia por errores que no se han podido solucionar por falta de tiempo. Por ello solo disponemos de las métricas *loss* y *acuracy*. loss: 2.6795 - accuracy: 0.4513 - val loss: 3.8351 - val accuracy: 0.3689

5.4 Comparación

Aunque no se han obtenido resultados significativos en la creación de modelos manuales o híbridos a continuación procedemos a comparar los modelos para ofrecer un flujo de trabajo completo. Se presenta cada tipo de modelo con una instancia; para el modelo 1 se ha elegido el mejor modelo modelo manual: incluida la capa LSTM adicional y el preprocesamiento más exhaustivo.

modelo	loss	accuracy	ROUGE	BLEU
1	1.3556	0.6138	12.24	0.036
2	3.8351	0.3689	—-	—-
3	0.5299	0.8312	40.32	0.068

Tabla 5.4: Comparación entre modelos con las métricas: pérdida, precisión, ROUGE y BLEU.

CAPÍTULO 6
Conclusiones

A pesar de que los modelos creados manualmente no han alcanzado niveles de eficacia significativos por ser aproximaciones simples en comparación con las del estado del arte, el hecho de haberse enfrentado a este problema ha provocado que el alumno adquiera los conocimientos necesarios para entender los conceptos relativos al campo, aprenda a crear modelos manualmente aunque sea a nivel básico.

De la realización de este trabajo podemos concluir que nos encontramos ante un problema no trivial que, aunque hemos vivido avances revolucionarios en el campo, aún encontramos un gran margen de mejora. Ya en la primera fase para abordar esta tarea encontramos una dificultad no contemplada. Nuestro dataset, comprendido por dos archivos de tipo CSV, estaba mal formateado. Los archivos de tipo CSV contienen un elemento por linea pero en nuestro caso encontramos elementos que se extendían por decenas de lineas y a veces no incluían las marcas necesarias para separar los atributos de dichos elementos.

Por otro lado, los ejemplos contenidos en el dataset distaban bastante de ser buenos. Entre ellos podríamos encontrar texto no válido para resumir como secuencias de código HTML o listas de sucesos con formato *hora - suceso*: *13:24 - Hamilton goes boxes*.

Una dificultad añadida es la frenética emergencia del campo. Al realizarse un número tan elevado de avances en distintos aspectos del NLP, resulta prácticamente imposible poder entender y explorar todos ellos. Esto, junto con la complejidad de algunos modelos provoca que las librerías tarden en incorporar estos de forma intuitiva y sencilla para los usuarios.

Aspectos importantes aprendidos por el alumno que consideramos necesario destacar son:

- Tanto los modelos como los dataset son cada vez más grandes, hasta tal punto de no ser abarcables en un sistema de usuario. Es por esto que encontramos muy útil la opción de cargar modelos (o partes de ellos) ya entrenados por corporaciones con tal capacidad. Gracias a esto evitamos el gran coste computacional necesario para crear tales modelos a la vez que podemos utilizar partes de estos para crear nuevos modelos que presenten resultados del estado del arte.

- A la hora de entrenar los modelos creados por el alumno se ha podido comprobar de primera mano la gran exigencia computacional que estos presentan. A medida que se aumenta la longitud de las secuencias, el tamaño de la representación de los datos (Embeddings), el número de capas y el número de neuronas que estas tienen; no solo aumenta el coste espacial sino también el temporal. Esto se debe a que las redes recurrentes son vagamente paralelizables. Una de las soluciones más prometedoras a este problema es el uso de Transformers. Arquitecturas que usan modelos de intención sin recurrencia.

Finalmente podemos manifestar satisfactoriamente que se han alcanzado los objetivos propuestos en la sección 1.2. Consideramos que se ha reunido la información necesaria para abordar la tarea de resumir textos así como otras del mismo campo (sección 4). Asimismo el alumno ha adquirido los conocimientos necesarios para realizar con éxito todas las fases de este proceso, además de comprender los fundamentos de esta disciplina, los cuales le permitirán seguir experimentando y empapándose de los avances casi diarios del campo.

6.1 Relación del trabajo desarrollado con los estudios cursados

Aunque este proyecto no se encuentra contenido directamente/exclusivamente en el glosario de estudio de ninguna asignatura impartida en el Grado en Ingeniería Informática impartido por la Universidad Politécnica de Valencia, sí se pueden destacar las asignaturas más relacionadas con el trabajo y aquellas que nos han servido de ayuda durante el desarrollo práctico de este.

Si bien se podría nombrar una retahíla de asignaturas que han contribuido indirectamente a la realización de este proyecto, se comentan únicamente aquellas con una relación directa. Asimismo no se relatan todos los contenidos de estas sino los más vinculados a este trabajo:

- **Sistemas Inteligentes (SIN):** En esta asignatura se presenta por primera vez el concepto de Machine Learning y se explica la unidad fundamental a partir de la cual surgen las redes neuronales el perceptrón.
- **Percepción (PER):** Esta materia explica cómo obtener y procesar la información obtenida desde distintos medios. Más concretamente indica la carga de texto desde diferentes tipos de archivo, el preprocesamiento que se le aplica a este y de qué opciones disponemos para representarlo.
- **Sistemas de Almacenamiento y Recuperación de información (SAR):** Además de la presentación de Python, el lenguaje de programación utilizado en este trabajo, esta asignatura enseña a mediar con grandes volúmenes de datos para un tratamiento eficiente de estos.
- **Aprendizaje Automático (APR):** Quizá esta sea la materia más relacionada con este proyecto. Explica en profundidad el campo del Machine Learning,

presenta las redes neuronales multicapa y el método de retropropagación que se utiliza para actualizar sus pesos.

6.2 Trabajos futuros

Se pretende que este trabajo sirva como punto de partida y/o guía para proyectos futuros que se enfrenten a la tarea de resumir textos o tareas similares como podría ser la traducción, la clasificación de textos o la creación de un modelo del lenguaje. Además se exponen a continuación una serie de propuestas para la ampliación y mejora del trabajo actual.

Se ha elegido el dataset de noticias indias de los periódicos *Times* y *Guardian* por su dificultad en el procesamiento, por su escasa cantidad de muestras y por la baja calidad de estas. Una posible aproximación puede ser utilizar un corpus mucho más extenso y de mayor calidad para comparar los resultados con los distintos tipos de modelos. Esto nos proporcionaría información muy valiosa acerca de si la mejora en la calidad de los datos se refleja por igual en todos los modelos y si, por lo contrario, algunos modelos no son capaces de aprender mejores parámetros aun con mejores datos.

En igual forma, se propone una ampliación de las arquitecturas manuales empleadas, tomando el proyecto escrito como base e implementando topologías más complejas y avanzadas como los modelos de atención. Esto permitiría un mayor control de los hiperparámetros y derivaría en un incremento del conocimiento así como el dominio de estas técnicas.

Dado que este ámbito no ha sido totalmente explorado, otra ampliación posible es la experimentación más exhaustiva en la aproximación que utiliza librerías de alto nivel para cargar algunas capas de modelos ya preentrenados. Se pueden utilizar en el *encoder* como *Embeddings* que ofrecen una representación de las secuencias mucho mejor que las obtenidas con los *Word Embeddings* clásicos.

Adicionalmente se propone la inclusión de redes convolucionales a la arquitectura desarrollada. Aunque no se hayan incluido en el capítulo 2 de este trabajo a causa del número creciente de modelos con mejores resultados, las convoluciones presentaron un avance significativo en el campo del NLP no solo en términos de eficacia sino de eficiencia [17]. Este hecho sirve de precedente como para experimentar con ellas e intentar construir un modelo que se aproveche de sus características.

Glosario

- **Cadena:** Secuencia de texto.
- **Conversión de tipos:** Proceso por el cual un objeto o elemento de un lenguaje de programación se transforma en otro de tipo distinto. Esto se debe a la necesidad de los métodos y funciones de recibir elementos de tipos concretos por su utilización interna. Este proceso es más común en los lenguajes fuertemente tipados.
- **Dataset:** Conjunto de datos, comúnmente con estructura de tabla, proveniente de una o varias fuentes y que contiene información relacionada con un único tema.
- **Deep Learning:** Conjunto de algoritmos de Machine Learning basados en múltiples capas de procesamiento no lineal que pretende modelar abstracciones de alto nivel mediante una estructura jerárquica en la cual las características de más alto nivel se derivan de las capas de nivel inferior.
- **Función de activación:** Si durante la creación de modelos se utilizaran únicamente operaciones lineales, el resultado colapsaría en una función. Para evitar este efecto no deseado se emplean las funciones de activación. Estas introducen no linealidad en la salida de cada neurona.
- **Hiperparámetros:** Comúnmente a la matriz de pesos que aprenden los algoritmos de Machine Learning se les conoce como parámetros del modelo. En contraposición, los parámetros que el modelo no aprende sino que son impuestos por el desarrollador se les llama hiperparámetros. Ejemplo: el número de capas recurrentes de un modelo de traducción de texto.
- **Inteligencia Artificial:** Conjunto de algoritmos y sistemas que pretenden imitar el comportamiento humano. Estos engloban tanto a los algoritmos que aprenden a partir de datos como a los programados por un experto para replicar cierta capacidad humana.
- **Inteligencia Artificial débil:** Modelos de Machine Learning expertos en resolver determinada tarea para la cual se le ha entrenado, incluso superando las capacidades humanas. Hasta la actualidad todos los modelos creados se clasifican como inteligencia artificial débil.
- **Inteligencia Artificial fuerte (o general):** Modelos de Machine Learning capaces de resolver diversas tareas de distintos campos no relacionados. Estos modelos hipotéticos serían capaces de resolver problemas de la misma manera en la que lo hacemos los humanos.

- **Lenguaje interpretado:** Lenguaje de programación en el cual el código escrito por el usuario no necesita ser compilado. En su lugar, un traductor o interprete lee una por una las líneas de código humano a la vez que las traduce a código máquina para que se ejecuten.
- **Machine Learning:** Conjunto de algoritmos capaces de aprender a realizar una tarea asociada normalmente a la inteligencia y capacidad humana. En oposición a los algoritmos clásicos, el desarrollador no indica cómo resolver el problema sino cómo utilizar los datos disponibles sobre esa tarea para aprender a resolverla.
- **N-grama:** Un n-grama es la agrupación de n términos, ya sean caracteres o palabras.
- **Natural Languaje Processing:** Subcampo de la lingüística y la inteligencia artificial que pretende dotar a las máquinas de la capacidad de interactuar mediante lenguaje natural para así poder, entre otras cosas, procesar y analizar grandes cantidades de datos en lenguaje humano.
- **One Hot Encoding:** Representación vectorial de palabras o términos en la cual cada palabra se representa mediante un vector del tamaño del vocabulario con todas las posiciones conteniendo ceros excepto la que identifica a esa palabra que será un uno.
- **Overfitting:** explain
- **PyCharm:** Entorno de Desarrollo Integrado (IDE por sus siglas en inglés) especializado en el desarrollo de aplicaciones en Python.
- **Stop Words:** Conjunto de palabras más comunes en un idioma que no aportan mucha información semántica a las frases. Dependiendo de la tarea se suelen eliminar para reducir el coste espacial y temporal.
- **Tokenizar (keras):** El significado que le otorga Keras al concepto de tokenizar es la acción de traducir secuencias de palabras o tokens a secuencias de números.
- **Tokenizar (nltk):** El significado que le otorga NLTK al concepto de tokenizar es el de convertir un texto en una secuencia de las palabras que este contiene.
- **Transfer Learning:** Conjunto de técnicas de Machine Learning que pretenden utilizar el conocimiento adquirido en la resolución de una tarea para aprovecharlo en la resolución de otra relacionada. Por ejemplo: las capas de una red neuronal profunda que aprenden a detectar contornos y ejes en un modelo de detección de coches también se podrían utilizar en un modelo de detección de camiones.
- **Transformers (arquitectura):** Los Transformers son una arquitectura en la cual se utilizan modelos de atención en capas convolucionales para solventar los problemas de dependencia a largo plazo entre palabras y de procesamiento (al poder dividir el problema y ejecutarlo en paralelo).

- **Transformers (librería):** Librería de Python desarrollada por la empresa HuggingFace que permite la utilización y mundificación de los modelos de NLP que conforman el estado del arte.
- **Web Scraping:** Técnica mediante la cual se recopilan grandes cantidades de datos alojados en la web y se almacenan en un archivo local o base de datos para su futuro procesamiento o análisis.
- **Word Embedding:** Representación vectorial de una palabra o término creada a partir de algoritmos que, con suficiente cantidad de datos, situarán en el espacio de representación conceptos similares juntos y conceptos distintos separados. De esta manera se consigue tener una representación con información semántica, al contrario que con la representación mediante One Hot Encoding.

Bibliografía

[1] Sepp Hochreiter The Vanishing Gradient Problem During Learning Recurrent Neural Nets And Problem Solutions. Institut fur Informatik, Technische Universitat Munchen.

[2] Sepp Hochreiter, Jurgen Schmidhuber LONG SHORT-TERM MEMORY

[3] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, Yoshua Bengio On the Properties of Neural Machine Translation: Encoder–Decoder Approaches

[4] Gail Weiss et al. On the Practical Computational Power of Finite Precision RNNs for Language Recognition

[5] Vaswani et al. Attention Is All You Need

[6] Jacob Devlin et al. BERT: Pre-training of Deep Bidirectional Transformers forLanguage Understanding

[7] Mike Lewis et al. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension

[8] Abigail See Get To The Point: Summarization with Pointer-Generator Networks

[9] Jingqing Zhang et al. PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization

[10] Yu Yan et al. ProphetNet: Predicting Future N-gram for Sequence-to-Sequence Pre-training

[11] Dongling Xia et al. ERNIE-GEN: An Enhanced Multi-Flow Pre-training and Fine-tuning Framework for Natural Language Generation

[12] Bird, Steven, Edward Loper and Ewan Klein. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

[13] Kishore Papineni BLEU: a Method for Automatic Evaluation of Machine Translation

[14] *News Summary*. Dataset and scripts for scraping the news articles from popular sources along with the summary of the article. Consultado en https://github.com/sunnysai12345/News_Summary.

[15] M. F. Porter An Algorithm for Suffic Stripping

[16] Chin-Yew Lin ROUGE: A Package for Automatic Evaluation of Summaries

[17] Jonas Gehring Convolutional Sequence to Sequence Learning