Departamento de Ingeniería Informática y Sistemas

TESIS DOCTORAL

Diseño e implementación de un Sistema Automático para la detección de opiniones turísticas en Redes Sociales

Carlos Alberto Martín Galán

Febrero de 2020

Directora: Rosa Mª Aguilar Chinea
Codirector: Jesús Miguel Torres Jorge

Dª. Rosa María Aguilar Chinea, Doctora en Informática y Catedrática de Ingeniería de Sistema y Automática de la Universidad de La Laguna, y asistida como Codirector por D. Jesús Miguel Torres Jorge, Doctor en Informática y Profesor del Departamento de Ingeniería Informática y Sistemas de la Universidad de La Laguna. CERTIFICA:

que D. Carlos Alberto Martín Galán, Ingeniero en Informática, ha realizado bajo mi dirección la presente Tesis, titulada "Diseño e implementación de un Sistema Automático para la detección de opiniones turísticas en Redes Sociales", para optar al grado de Doctor por la Universidad de La Laguna.

Con esta fecha, autorizo la presentación de la misma.

En San Cristóbal de La Laguna, a 17 de enero de 2020.

La Directora,

Rosa Mª Aguilar Chinea

A mis padres Carmen y Jesús,
que hicieron lo imposible para que yo pudiera formarme.
Sois unos padres maravillosos.

A mis hijos Santi y David, y a mi mujer Yurena,
porque me han dado todo su amor y apoyo
para poder terminar este trabajo.

Agradecimientos

Quería dar mi más sincero agradecimiento a mis directores Rosa María Aguilar Chinea y Jesús Miguel Torres Jorge, que me han acompañado en todo este proceso, no solo enseñando, sino también aconsejando y animando para que todo saliera lo mejor posible.

También quería dar las gracias a los profesores y amigos Juan Albino Méndez Pérez y Silvia Alayón Miranda que siempre se han ofrecido para ayudarme en lo que fuera posible de forma desinteresada, así como a todos los compañeros del Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, por facilitarme la labor en tantas ocasiones.

Gracias también a los compañeros de los Servicios Informáticos del Instituto de Astrofísica de Canarias, especialmente a Antonio Jesús Díaz Chinea que siempre me han dado cobertura para poder asistir a las actividades que este trabajo requería.

Por último quería agradecer a todos los profesores y personal de administración y servicio que hacen posible que funcione el programa de Doctorado de de Ingeniería Industrial, Informática y Medioambiental de la Universidad de La Laguna. Su implicación ayuda mucho a todos los doctorandos que iniciamos esta aventura.

Este trabajo ha contado con financiación por parte del proyecto *VITUIN: Vigilancia Turística Inteligente de Tenerife en Redes Sociales* a través de los fondos de investigación de la Fundación Cajacanarias.

Índice

Índice de tablas

Índice de figuras

Índice de acrónimos

Índice de figuras

xx

Símbolos y notación

a	escalar
$P(a)$	Probabilidad de a
$\bar{x} = (x_0, ..., x_n)$	vector x
a_i	*i*-ésimo elemento del vector o del conjunto
$\bar{x_i}$	vector codificado de la i-ésima muestra del conjunto
$sign(a)$	función signo de a
$\|\|a\|\|$	norma euclídea de a
a_j^l	salida de la *j-ésima* neurona de la capa l
$(...)$	vector o matriz
$\{...\}$	conjunto o lista de elementos
$tanh(x)$	función tangente hiperbólica en x
$\sigma(x)$	función sigmoide en x

Introducción

El turismo es uno de los sectores económicos más importantes en España, y aún más en las Islas Canarias, donde llegan millones de turistas cada año. El clima, su naturaleza, su cultura, su gastronomía y su historia son un patrimonio muy atractivo para nuestros visitantes, pero es igualmente importante no descuidar los servicios que ofrecemos: alojamiento, transportes, ocio o restauración.

Para poder mantener, o incluso mejorar la calidad de estos servicios, es necesario contar con la opinión de nuestros visitantes. Los métodos tradicionales de recopilación de opiniones, como son las encuestas de satisfacción, son excesivamente dirigidos y tienen un alto sesgo, perdiendo así parte de su utilidad.

Las nuevas tecnologías han revolucionado el mundo de las comunicaciones. La aparición de internet y de las redes sociales permiten a los usuarios expresar su opinión de forma libre y espontánea. Las grandes empresas de todos los sectores han aprovechado este hecho, y cada vez aparecen nuevos portales especializados donde se ofrecen productos a los consumidores en los que pueden expresar su opinión o experiencia con los mismos. Esta fuente de información es un activo muy valioso que

podemos aprovechar para recopilar los sentimientos que producen en los visitantes los servicios que ofrecemos en nuestro destino turístico.

El tratamiento de esta información, tan grande y sin una estructura definida, no es simple, por lo que debemos utilizar técnicas automatizadas para poder realizar un tratamiento adecuado. Las opiniones se expresan en lenguaje natural, muchas veces mediante mensajes cortos que pueden incluir símbolos o imágenes que añaden significado a los mismos.

El Aprendizaje Máquina, una de las disciplinas más en auge actualmente dentro de la Inteligencia Artificial, nos ofrece herramientas para poder clasificar este tipo de información de forma que pueda ser utilizada para detectar riesgos de degradación de los servicios o publicitar aquellos bien valorados que aún son poco conocidos.

El presente trabajo de investigación hace un recorrido por las técnicas más actuales del aprendizaje máquina. Comienza proponiendo diversas alternativas de codificación para los textos y realiza un análisis exhaustivo de los modelos de aprendizaje profundo más utilizados en la actualidad en problemas de procesamiento de lenguaje natural.

El objetivo final es analizar qué técnica o combinación de técnicas se ajusta mejor al dominio de los datos del caso de estudio, y poder realizar un estudio de polarización de las opiniones que nuestros turistas dejan en las redes sociales y portales especializados con la mayor precisión posible.

Los modelos más prometedores podrán ser utilizados en herramientas automáticas que permitan monitorizar la satisfacción de nuestros turistas o incluso servir como entrada a sistemas inteligentes que puedan mitigar problemas de imagen o realizar campañas de marketing dirigidas.

Capítulo 1

Motivación turística.

La industria del turismo representa en la actualidad el 10,4 % del Producto Interior Bruto (PIB) mundial según el *World Travel and Tourism Council* lo que lo convierte en uno de los sectores más productivos, y empleando a una de cada diez personas (WTTC, 2019). En España, en concreto, supone el 14,6 % del total de la economía.

En las Islas Canarias, los datos son aún más significativos, ya que la aportación del sector a su PIB es del 35 % y soporta casi el 40 % de los empleos de las islas, recibiendo en el año 2.018 15,56 millones de turistas (Statista, n.d.).

Aunque la industria del turismo es de vital importancia en el crecimiento y desarrollo de Canarias, tiene igualmente una influencia favorecedora en otros sectores como son el comercio, el transporte, la producción de alimentos y en general toda la industria.

La relevancia del sector en nuestra economía, así como la importancia de mantener su productividad en un mundo tan globalizado, cambiante y competitivo, motivan a aplicar nuevas tecnologías que ayuden a conocer

mejor el sector, y así corregir deficiencias o proponer mejoras.

1.1. La importancia de la información.

A la vista de la importancia económica del turismo en nuestras Islas, y del constante cambio al que se ve sometido el sector, la supervivencia del mercado de nuestros destinos y empresas dependerá en gran medida de su capacidad para ofrecer nuevas experiencias, productos y servicios, que se adapten a sus demandas de una forma actualizada. Por este motivo, es innegable la importancia de conocer cuáles son las variables socioeconómicas claves del sector para poder tomar las decisiones adecuadas. En concreto, es de vital importancia adaptarse a las siguientes condiciones:

- Requisitos y demandas de los clientes, así como las condiciones del sector en cada momento para poder competir nuevos destinos y mercados.
- Duración y distancia de los viajes, para adecuar los servicios a los mismos.
- A la situación financiera de los países de destino más habituales.
- A la creciente autonomía de las nuevas tecnologías de información y comunicación en los procesos de marketing, distribución y comunicaciones.
- A los límites de la capacidad ambiental, para poder crear un mercado sostenible.

Parte de la captación de la información para construir estos indicadores proviene de los propios usuarios. Las empresas tienen cada vez más presente la importancia de las opiniones de sus clientes, acerca de sus productos y servicios. El uso masivo de los dispositivos electrónicos para comunicarnos en nuestro día a día ha supuesto una explosión de comentarios que los usuarios publican en las redes sociales Redes Sociales (RRSS).

Esta nueva forma de conocer y de opinar por parte de los usuarios, espontánea y natural, no está condicionado en ninguna forma por la forma de la encuesta, lo que significa un mayor grado de objetividad. A través de los comentarios podemos descubrir cuáles son los factores que realmente importan para ellos:

- Detectando insatisfacciones entre lo ofrecido y lo obtenido.
- Localizando comportamientos no deseados (servicios inadecuados o infraestructuras en mal estado) que pongan en peligro el prestigio del destino.
- Localizando elementos de éxito pocos referenciados para potenciarlos en campañas de marketing directo.
- Localizando elementos de éxitos muy referenciados para poderlos mantener al nivel adecuado.

1.2. eWOM Boca a Boca Electrónico.

A diferencia de las conversaciones tradicionales que se tienen lugar en persona en un entorno determinado, las conversaciones digitales se realizan usando nuevos métodos y herramientas para involucrar a los contertulios, cuya característica de interacción social es el foco principal de

su dinámica (Sotiriadis y van Zyl, 2013). Es lo conocido como boca a boca electrónico Electronic Word of Mouth (eWOM), cuyas características y consecuencias son completamente diferentes (de Vries *et al.*, 2012) respecto a las conversaciones tradicionales. Estas características se centran en la facilidad con que se distribuye el mensaje a través de las RRSS, que son plataformas de comunicación en línea donde los usuarios crean su propio contenido permitiendo la escritura, publicación e intercambio de información (Kaplan y Haenlein, 2010). La incorporación del eWOM a las RRSS tiene las siguientes propiedades:

- Facilidad para la distribución o emisión del mensaje.
- Gran capacidad para la difusión entre los usuarios que pueden acceder a opiniones o sentimientos de otros.
- Gran diversidad de perfiles entre los usuarios, que tienen diferentes edades y pertenecen a diferentes colectivos, pero que comparten la información.
- Gran diversidad de medios de publicación: herramientas de mensajería básicas, portales de red social, o portales especializados al sector turístico.
- Persistencia en el tiempo, ya que las discusiones se cargan para referencia actual y futura.
- Información espontánea, sin pretensiones comerciales previas.

Estas características hacen que la monitorización de eWOM por parte de compañías turísticas sea particularmente relevante (Hennig-Thurau *et al.*, 2004).

De acuerdo con Hennig-Thurau *et al.* (2004), un eWOM es "cualquier declaración positiva o negativa hecha por clientes potenciales, actuales o anteriores sobre un producto o empresa, que se pondrá a disposición de una gran cantidad de personas e instituciones a través de Internet". Se compone de cinco elementos principales:

1. Declaración (positiva, negativa o neutra).
2. Comunicador (emisor), que incluye clientes potenciales, actuales o pasados.
3. Objeto (producto o servicio).
4. Receptor (multitud de personas e instituciones).
5. Entorno (internet, en concreto RRSS y portales especializados).

De acuerdo con (Peppers y Rogers, 2016), "una empresa que se vuelca en sus clientes es una empresa que utiliza la información para obtener ventaja competitiva y alcanzar crecimiento y la rentabilidad". Por este motivo es necesario trabajar en cuatro dimensiones:

- Identificación del clientes.
- Atracción.
- Retención.
- Desarrollo.

Esta cuarta dimensión es la que trata de descubrir y aprender de lo que opinen. La orientación al cliente permitirá desarrollar aquellos servicios turísticos que realmente necesitan nuestros visitantes, y esta información

la obtendremos del eWOM. Sin embargo, el eWOM también presenta algunas dificultades en su uso:

- La obtención de los datos. Como ya se ha comentado, la información que circula en internet (RRSS y portales especializados) es espontánea y natural. Sin embargo su explotación automatizada presenta la gran dificultad de la heterogeneidad de las fuentes, sus estructuras de información y la dificultad añadida del procesamiento del lenguaje natural. Las técnicas utilizadas para el análisis de contenido que se han usado hasta el momento son manuales (Zhang y Tsm, 1970) o semi-manuales (Pan *et al.*, 2007), por lo que el trabajo de su estudio implica gran tiempo y muchísimo esfuerzo. Con esas técnicas los investigadores deben extraer los contenidos desde el sitio web objetivo para almacenarlos en un archivo. Tras la extracción y el almacenamiento, hay que exportar el archivo donde se almacenan los contenidos a la herramienta de análisis. Esta manera de trabajar no es automática ni fácilmente reproducible y, además, la información y su análisis no se pueden obtener en tiempo real, pese a que se sabe que los contenidos de las distintas plataformas web se actualizan constantemente.
- La relación del comentario con el destino. Los problemas de categorización de los destinos turísticos están relacionados con los principales elementos que afectan a su imagen, para saber cómo gestionarlo y promocionarlo. Existe extensa literatura que demuestra la importancia de conocer los componentes que afectan a la imagen turística y su influencia hacia otros factores, por lo que puede ser usada como herramienta para trabajar la decisión y comportamiento de los visitantes (Marine-Roig, 2010; Park y Njite, 2010). La

imagen es un concepto complejo y multidimensional (Gallarza *et al.*, 2002), por lo que es importante hacer análisis de aquellos elementos que conforman la percepción de los turistas con respecto a un lugar determinado.

- El sentido de los mensajes extraídos. El análisis de sentimientos de textos en las redes sociales es un área de investigación cada vez más relevante por el impacto que tiene para las empresas. Sin embargo, el lenguaje humano es complejo y es difícil enseñar a un programa a diferenciar matices gramaticales, variaciones culturales, jergas o filtrar faltas de ortografías. Entender el contexto es un proceso muy difícil, más teniendo en cuenta que no se puede usar el tono al tratarse de un medio escrito. Actualmente para el análisis de sentimientos en texto existen corpus como WordNet Affect (Strapparava y Valitutti, 2004) y Affective Norms for English Words (ANEW) (Bradley y Lang, 1999), que facilitan usar una estrategia semántica para detectar emociones en un texto. Estos enfoques han sido combinados con algoritmos de clasificación, en (Sidorov *et al.*, 2013) donde se clasificaron más de 7000 publicaciones de Twitter en opiniones positivas, negativas y neutras con una precisión del 61 %. Otros trabajos combinan un clasificador semántico que consume una jerarquía de palabras enriquecida y un clasificador Naive Bayes (Miranda *et al.*, 2014).

Como se ha visto, debido a la diversidad y cantidad de información, hacer un seguimiento en tiempo real de la información que circula por internet es un reto que sólo se puede abordar de forma automatizada.

El valor de las opiniones de los clientes es pues un activo muy importante para cualquier empresa o sector, especialmente como se ha visto el

sector turístico. El análisis de los sentimientos que expresan los usuarios de las diferentes RRSS y de los portales especializados puede ser tratado utilizando técnicas electrónicas que permitan un procesamiento masivo. El siguiente capítulo mostrará una breve clasificación de estas técnicas y en qué grupo se encuadra el presente trabajo de investigación.

Capítulo 2

Procesamiento del Lenguaje Natural.

En el capítulo anterior se destacó la importancia que tiene la industria del turismo a nivel mundial, más incluso en el ámbito nacional y particularmente en las Islas Canarias. Este sector, como muchos otros, se ha visto sometido a importantes cambios con la aparición de nuevas tecnologías, y sobre todo con los nuevos medios de comunicación.

Es indudable la importancia que tiene para cualquier empresa la opinión de sus clientes. De forma tradicional siempre se ha tenido en cuenta ésta, y por este motivo se recogía por medio de encuestas o campañas de información. Sin embargo, hoy en día estas opiniones las podemos recolectar de fuentes inagotables de datos como son los portales especializados, las RRSS, y en general de internet.

La opinión de los usuarios, en este caso turistas, está expresada normalmente en el idioma de los propios usuarios (acompañado en algunas ocasiones de nuevos símbolos que resumen sus estados de ánimo). Como ya expresamos en el capítulo anterior esta información es natural y

objetiva, pero también presenta algunos inconvenientes: el volumen de información manejada es muy grande, y además presenta una ausencia total de estructura, lo que impide realizar un tratamiento tradicional de bases de datos.

El tratamiento de este tipo de información ha conllevado el auge de técnicas de inteligencia artificial, en concreto aquellas destinadas al Procesamiento del Lenguaje Natural (PLN) y al análisis de sentimientos.

2.1. Análisis de sentimientos.

El análisis de sentimientos es uno de los campos más activos en el área del procesamiento del lenguaje natural desde el año 2000 (Liu, 2012). Una de las definiciones más acertadas podría ser: ”el propósito del análisis de sentimientos es definir herramientas automáticas capaces de extraer información subjetiva para crear conocimiento estructurado que pueda ponerse en funcionamiento”.

Su principal objetivo es definir herramientas automáticas para obtener información de texto expresado en lenguaje natural, como opiniones y sentimientos que sirvan para posteriormente realizar acciones o tomar decisiones. Está claro que esta disciplina encaja perfectamente con este trabajo de investigación, que trata de identificar fortalezas o debilidades de un destino turístico a partir de las opiniones de los visitantes expresadas en un lenguaje natural.

Existe cierta confusión entre los investigadores entre los conceptos de análisis de sentimientos y minería de opiniones. La diferencia es bastante

sutil ya que uno contiene elementos de otro. Las definiciones indican que la opinión es más un punto de vista personal sobre algo, mientras que un sentimiento es más relativo a la sensación que se ha producido. Una misma persona podría decir "creo que el aire acondicionado del hotel estaba demasiado alto" indicando una opinión sobre estado de un servicio, de la misma manera que podría decir "estaba muy a gusto en la habitación gracias a que el aire acondicionado estaba fuerte". Otra persona podría compartir la opinión (que hemos expuesto en primer lugar), y sin embargo tener un sentimiento diferente respecto a la segunda (si fuera el caso que sintiera frío en la mayoría de las ocasiones). En cualquier caso están fuertemente relacionadas, ya que el sentimiento en esta ocasión venía ocasionado por el hecho del que estaba opinando.

En (Liu, 2012) se define una opinión como una quíntupla (e_i, a_{ij}, s_{ijkl}, h_k, t_l) que corresponde con:

- e_i es la entidad.
- a_{ij} es un aspecto de la entidad.
- s_{ijkl} es el sentimiento acerca del aspecto a_ij de la entidad e_i.
- h_k es el titular de la opinión.
- t_l es el instante de tiempo en el que el titular h_k expresó la opinión.

El aspecto es obviamente una de las dimensiones a tener en cuenta. Un visitante puede opinar en un mismo comentario sobre diferentes aspectos de un mismo servicio (un hotel podría tener magníficas instalaciones deportivas y un pésimo menú o limpieza de habitaciones). Su valoración

entonces puede estar condicionadas por diferentes opiniones sobre diferentes aspectos, y es importante distinguir entre ambos.

El sentimiento puede ser positivo, negativo o neutro, y además puede tener diferentes niveles de intensidad en los sentidos positivos y negativos (algunos de los portales de opinión turística que estamos utilizando en este trabajo de investigación usan estrellas o burbujas para enfatizar la fuerza de una opinión).

En concordancia con la definición presentada "el propósito del análisis de sentimientos es definir herramientas automáticas capaces de extraer información subjetiva para crear conocimiento estructurado que pueda ponerse en funcionamiento", la creación de tuplas estructuradas como las que se han presentado en función de los opiniones vertidas por los visitantes de un destino turístico puede considerarse como el paso previo para la creación de un entorno de trabajo para herramientas de procesamiento analítico. Éstas deben permitir o facilitar la toma de decisiones para minimizar las debilidades de los servicios e infraestructuras y potenciar las fortalezas de las mismas o aprovechar las oportunidades.

Los avances tecnológicos actuales permite el almacenamiento o recuperación de grandes cantidades de información, creando conocimiento de fuentes de de datos sin estructurar. Las RRSS son una oportunidad. Esta gran cantidad de comentarios, en forma de mensajes cortos de texto, que podrían ser procesados en tiempo real empujará sin duda investigaciones multidisciplinares incluyendo el procesamiento del lenguaje natural y el aprendizaje automático.

La tendencia general del análisis de sentimiento en redes sociales es aplicar técnicas heredadas del tradicional análisis de sentimientos de antes del 2000, pero considerando la evolución de las fuentes de los datos y sus formas (mensajes cortos, contenido con otros elementos que no son texto, metadatos en los mensajes de los que obtener más información como la localización o la edad). Está claro que las redes sociales han modificado también nuestro lenguaje. El uso de móviles y tabletas hacen que escribamos de otra forma diferente a como lo haríamos años atrás (no hay que irse al papel, una misma persona escribe diferente con teclado mecánico que en una pantalla de una tableta). Teniendo en cuenta estos aspectos, y que el lenguaje va a seguir evolucionado con el uso de las RRSS los sistemas que utilicen el análisis de sentimientos tendrán que adaptarse de forma nativa.

El análisis de sentimientos es un campo de investigación complejo. Estas son sus principales características (Federico Alberto Pozzi, 2016):

- Los sentimientos se categorizan. Cuando se analiza una frase el primer paso es distinguir si es una información objetiva o subjetiva. Si es objetiva no se requiere una fase posterior, pero por si el contrario es subjetiva hay que analizar si su polaridad (Wiebe *et al.*, 1999) que podrá ser positiva, negativa o neutra.
- Existen diferentes niveles en los que realizar el análisis. Cuando se investiga hay que decidir en qué capa se hace el análisis del texto. Lo más habitual es hacer a nivel de mensaje (o comentarios), a nivel de frase, o a nivel de entidad y aspecto (esto afectaría a las dimensiones de las tuplas que se han mencionado anteriormente en este documento).

- Las opiniones pueden ser expresadas en su forma normal o comparativa. Habitualmente las opiniones en forma normal suelen llamarse opiniones estándar y se pueden expresar de forma directa (ej: El hotel estaba sucio) o de forma indirecta (ej: cuando usaba cualquier instalación del hotel mi ropa se manchaba). Las opiniones expresadas en forma comparativa establecen relación entre dos o más entidades y puede afectar a ambas (ej: El hotel A tenía mejores instalaciones que el hotel B).
- Las opiniones pueden ser explícitas o implícitas. Las opiniones explícitas suelen ser claramente subjetivas (ej: la decoración del restaurante es horrible) , mientras que las implícitas suelen ser objetivas que expresan un deseo oculto, en ocasiones muy difíciles de encontrar (ej: el sábado está incluida la cena en hotel, no puedo esperar que llegue el momento) (es difícil determinar si el deseo viene dado por la calidad de la cena o no).

Aparte de estas características se deben tener en cuenta otros aspectos que hacen especialmente difícil el análisis de sentimientos.

Uno de ellos puede ser la semántica y el análisis del contexto. Para poder realizar una buena valoración se deben tener en cuenta informaciones relativas difíciles de detectar de forma automática (ej: "la decoración del hotel era terrorífica" cambia de polarización dependiendo si el comentario es emitido la noche de Halloween o no). La construcción de léxicos, corpus u ontologías ayudarán sin duda a la comprensión del contexto de un mensaje.

Otro aspecto que se deben tener en cuenta son las figuras retóricas del

lenguaje, como por ejemplo la ironía y el sarcasmo. La ironía trata de dar a entender lo contrario de lo que se dice, y el sarcasmo (que por otro lado hace uso de la ironía) tiene como objetivo burlarse u ofender a algo a a alguien. Ambas figuras son complicadas de analizar de forma automática y complican el análisis de las muestras.

Por último, otra consideración a tener en cuenta del análisis de sentimientos en RRSS o en portales colaborativos es la relación existente entre los emisores de los mensajes. Hasta el momento, lo habitual es asumir que las opiniones recogidas eran independientes. Sin embargo las RRSS y su contenido pueden estar relacionadas con el principio de homofilia (Lazarsfeld y Merton, 1964) que supone que usuarios conectados tienen tendencia a tener opiniones similares y reflejar la realidad de la misma manera.

Como se puede deducir de los párrafos anteriores, el PLN es un área de la inteligencia artificial muy extensa, con muchos aspectos a tener en cuenta a la hora de realizar una aplicación práctica. En este trabajo de investigación nos hemos centrado en la aplicación de técnicas de aprendizaje automático para realizar un análisis de la polaridad de las opiniones de turistas. En la siguiente sección se realiza una breve clasificación de las técnicas actuales de aprendizaje automático para la clasificación de textos expresados en lenguaje natural, como es el caso de estudio.

2.2. Aprendizaje Supervisado

El término *Inteligencia Artificial* (Inteligencia Artificial (IA)) aparece en las ciencias de computación en los años 50 para hacer referencia a la ca-

pacidad de las máquinas, y en concreto de los programas de cómputo a realizar actividades que requieran inteligencia, cosa que hasta el momento solo se pensaba posible en los humanos. Durante su aparición, algunas tareas se consideraron dentro del área de la IA, pero el avance tecnológico y la consiguiente capacidad de las máquinas para hacer cálculo las han convertido en tareas no inteligentes. En la actualidad, son muchas las aplicaciones que se encuentran dentro de este campo. Entre muchas otras:

- Reconocimiento de voz.
- Visión artificial.
- PLN.

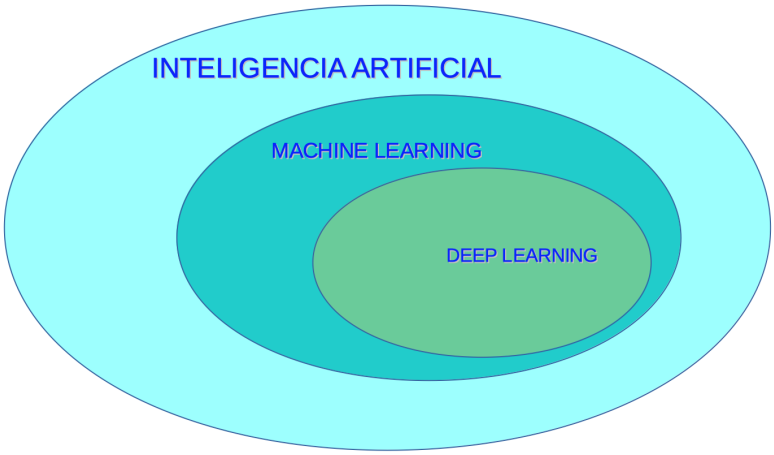


Figura 2.1. Jerarquía: Inteligencia Artificial, Machine Learning y Deep Learning (Cajamarca, 2019).

El aprendizaje automático (Machine Learning (ML)) es una rama de la IA. Incluye todas aquellas técnicas que permiten aprender a los ordenadores. Se utilizan en problemas de compleja naturaleza, que no pueden ser resueltos únicamente mediante métodos numéricos. La figura 2.1 representa la Jerarquía existente entre la IA y el ML.

En la programación clásica, un conjunto de reglas (especificadas como un algoritmo), con un conjunto de datos es capaz de producir respuestas. El ML cambia el paradigma, de forma que a partir de un conjunto de datos, y las respuestas ya conocidas, la máquina sea capaz de producir las reglas (Cajamarca, 2019). La figura 2.2 representa este concepto.

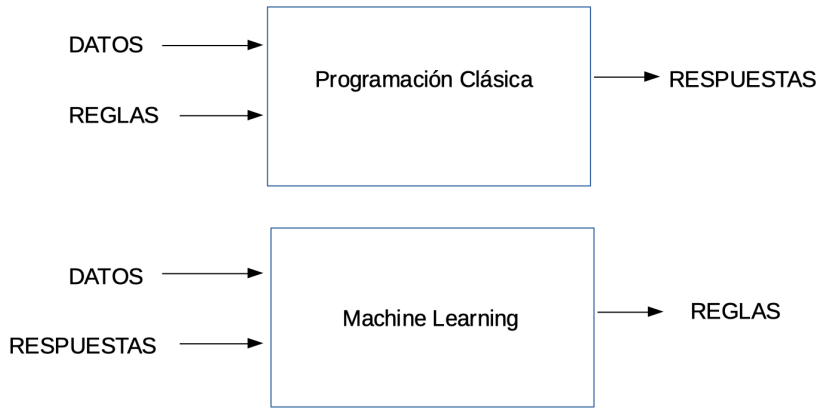


Figura 2.2. Paradigma Machine Learning vs Programación Clásica (Cajamarca, 2019).

La idea que subyace es que el sistema de ML es entrenado, en vez de programado. Mediante los datos de partida que han generado las respues-

tas el sistema aprende los patrones, y aprende una forma de representar las reglas que los generan. A posteriori, una vez las reglas ya han sido construidas, el sistema será capaz de generar respuestas ante nuevos datos de la misma naturaleza con los que fue entrenado.

Las aplicaciones expuestas anteriormente, son en concreto, aplicaciones de ML. Uno de los usos de estas técnicas en los últimos tiempos es el que ocupa este trabajo de investigación, el análisis de sentimientos. La discusión del estado del arte en lo que refiere a las técnicas del análisis de sentimientos desde la perspectiva del ML distinguen entre:

- Aprendizaje supervisado: El sistema es capaz de aprender o generar reglas a partir de ejemplos de los cuales conocemos la respuesta o salida. A los ejemplos cuya salida conocemos se les suele dar el nombre de *datos etiquetados*. Cuando la salida obtenida es una categoría se les denomina *modelos de clasificación*. Por el contrario, cuando la salida es un valor continuo se les denomina *modelos de regresión*. Como se podrá comprobar en capítulos posteriores, uno de los inconvenientes a la hora de aplicar estas técnicas radica en encontrar suficientes ejemplos etiquetados.

- Aprendizaje no supervisado: En este caso, el sistema es capaz de aprender reglas capaces de agrupar muestras según su parecido. Entre sus aplicaciones se encuentra el agrupamiento, conocido como *clustering* o la simplificación de la estructura (*reducción de la dimensionalidad*). A diferencia del anterior, el aprendizaje no supervisado no requiere de datos previamente etiquetados.

- Aprendizaje semisupervisado: Se basa en la combinación de los dos

anteriores. En concreto, su principal utilidad aparece cuando el número de ejemplos etiquetados previamente es reducido, y utilizamos ejemplos no etiquetados para completar el entrenamiento del sistema.

Nuestro objetivo será encontrar una función que prediga si un texto que expresa un sentimiento o una opinión sobre una infraestructura o servicio emite una valoración positiva, negativa o neutra. El procesamiento del lenguaje natural y el análisis de sentimientos son problemas complejos en el que intervienen muchas dimensiones diferentes. El dominio de datos del problema que se trata son los comentarios de los visitantes, y por lo tanto las funciones de predicción no son simples. Por este motivo, en el trabajo de investigación que se propone intentaremos centrarnos en el aprendizaje supervisado, utilizando un conjunto de opiniones de visitantes que ya han sido previamente clasificadas como positivas o negativas.

Es importante resaltar que para el aprendizaje automático tenga éxito es absolutamente necesario usar un conjunto de datos de entrenamiento grande. Afortunadamente las RRSS y los portales colaborativos ofrecen gran cantidad y gran variedad de los mismos. Sin embargo, el análisis de información en RRSS tiene algunos retos de complejidad con los que hay que tratar:

- Textos cortos con alto contenido semántico, que pueden contener otro tipos de información (como hashtags en algunas redes sociales o incluso imágenes).
- Contenido ruidoso. Lleno de expresiones coloquiales, abreviaturas, emoticonos y estructuras gramaticales propias de ese mundo.

- Gran dinamismo. La tarea de clasificación de opiniones es muy volátil, ya que las mismas pueden cambiar en poco espacio de tiempo.
- Información explícita e implícita. Puede necesitar de la correlación con otras variables para poder hacerla efectiva.
- Gran diversidad de idiomas.
- Las relaciones. Las redes sociales y los portales se basan en la conexión entre diferentes fuentes de opinión. Es igualmente un reto importante reforzar la semántica (o simplemente despejar las dudas sobre algunas interpretaciones) analizando otras con las que se relaciona.

Existen otros aspectos que pueden ser tratados de forma lateral a la clasificación de textos donde los turistas expresan sus emociones. Algunos de estos aspectos que podrían tratarse para añadir significado, y que no aparecen normalmente en textos convencionales son:

- Identificación de palabras o símbolos que detecten subjetividad, polaridad o ironía.
- Identificación de contenido para-lingüístico, como emoticonos o iniciales que ayude en la clasificación de la opinión.
- Uso de estilos de escrituras, como uso de negritas, cursivas o entrecomillados.
- Uso de letras mayúsculas. En muchos foros ya estándar para expresar alta voz.

Los métodos tradicionales pueden combinarse con técnicas de detección de estos aspectos para mejorar la precisión de sus predicciones. En los párrafos posteriores se exponen de forma muy breve, a forma de introducción, algunas de las técnicas más utilizadas actualmente en la clasificación de textos, y que podrían se utilizadas en el caso de estudio. Aquellas que han sido utilizadas en este trabajo de investigación se comentarán con más detalle en capítulos posteriores.

- **Support Vector Machine (SVM)**. Las máquinas de vectores de soporte SVM son un conjunto de algoritmos de aprendizaje automático que se incluyen dentro de la rama de aprendizaje supervisado. Aparecen por primera vez en el año 1963 de la mano de Vladimir Vapnik.

  La aplicación principal de la familia de algoritmos SVM es clasificar un conjunto de datos en grupos. Para ello tratan de construir un *hiperplano* que mantenga a la misma distancia (*margen*) aquellas muestras del conjunto que se encuentran más cerca de él. A éstas se le conoce como **vectores soporte** y son las que dan nombre a este algoritmo.

  En el caso de estudio, los datos a clasificar son las opiniones de los turistas. Como es lógico, la representación vectorial numérica de estas opiniones tendrán muchas dimensiones, por lo que no es posible representar de una manera clara y sencilla las muestras, el hiperplano y el margen. En el índice de figuras, podemos encontrar la figura 4.1 donde se puede ver una representación simplificada.

  En el capítulo 4, en concreto en la sección 4.1 se detalla el funcionamiento de la técnica SVM y que fue utilizada en este trabajo de

investigación dando excelentes resultados.

- **KNN**. La técnica de los *K vecinos más cercanos*, igual que la técnica anterior se encuentra dentro de la rama de aprendizaje automático supervisado.

  Su fundamento básico es que cada nueva muestra a analizar se incluirá en el grupo correspondiente según sea el grupo de sus vecinos más cercanos. El número de vecinos a tener en cuenta para su clasificación se parametriza con *k*. El algoritmo calcula la distancia de la muestra que queremos clasificar con el resto de elementos, y se queda con los *k* vecinos que estén más cerca de él. Si el número *k* es impar, bastará con llevar a cabo un sistema de votación simple para etiquetar la nueva muestra.

  A diferencia de otras técnicas, KNN no genera un modelo a través de un entrenamiento, sino que la clasificación se realiza en el momento de clasificar la nueva muestra.

  La aplicación de esta técnica depende en gran medida del valor que demos al parámetro *k*, y también de la representación numérica que demos a las muestras y del cálculo de la distancia entre dos muestras. En nuestro caso de estudio por tanto, al tratarse de opiniones de turistas, el número de características o dimensiones para representarlo será grande. La figura 2.3 representa gráficamente la incidencia del parámetro *k* en el algoritmo.

  No existe ninguna regla para determinar el valor de *k* en cada caso, ya que depende totalmente de las muestras etiquetadas que tengamos. Como parece lógico, elegir valores de *k* elevados reduce la probabilidad de etiquetar mal por el hecho de tener valores de rui-

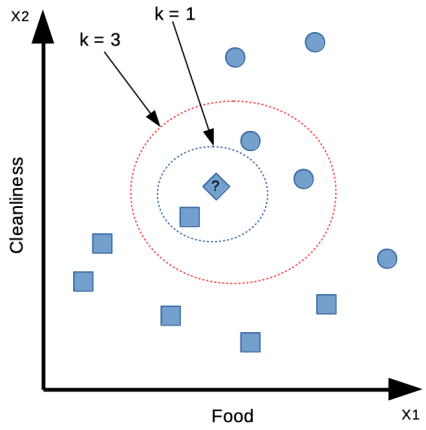


Figura 2.3. Influencia del parámetro K en KNN.

do (en este caso serían comentarios mal etiquetados). Como en todos las técnicas de aprendizaje supervisado, la calidad del etiquetado de las muestras es determinante en la precisión de la misma. Conjuntos de datos ruidosos hacen fracasar el uso de esta técnica. El valor $k = 1$ da lugar al algoritmo conocido como *Nearest Neighbor Algorithm*.

Existen muchas variantes del algoritmo KNN. Algunas de las más conocidas son:

- KNN con distancia media. Que consiste en etiquetar la nueva muestra pero calculando la media de la distancia con los elementos elegidos de cada grupo (en vez de realizar un sistema de votación).
- KNN ponderado. En esta ocasión no todos los vecinos contri-

buye de igual manera al etiquetado de la nueva muestra. Se le asignará más peso a aquellos vecinos más cercanos.

En este trabajo de investigación no se aborda en detalle la aplicación de la técnica KNN al caso de estudio, optando por la aplicación de técnicas de aprendizaje profundo.

- **Deep Learning (DL)**.

  El aprendizaje profundo DL se basa en el uso de redes neuronales y forma parte de las disciplinas de aprendizaje automático supervisado. Usan modelos computacionales compuestos de múltiples capas de procesamiento (la profundidad de las capas le da nombre a la disciplina). El uso de estas técnicas ha supuesto un gran avance en algunos campos de procesamiento, como pueden ser reconocimientos de imágenes, del habla, así como el procesamiento del lenguaje natural.

  Una red neuronal es un conjunto de unidades simples de procesamiento (neuronas) que se conectan de diversas maneras y que tienen diferentes funciones de activación. Cada una de estas unidades de procesamiento combina mediante una función de ponderación las entradas que recibe con un peso para cada una de ellas, y mediante una función de activación emite una salida de otro dominio. De forma básica, y de la misma manera que se explicó en párrafos anteriores el aprendizaje supervisado en las redes neuronales consiste en la modificación de los pesos de las neuronas en función del error cometido al introducir como entrada muestras de las que se conocen previamente sus valores reales. La figura 2.4 muestra la forma que tiene una neurona básica perteneciente a una red neuronal.

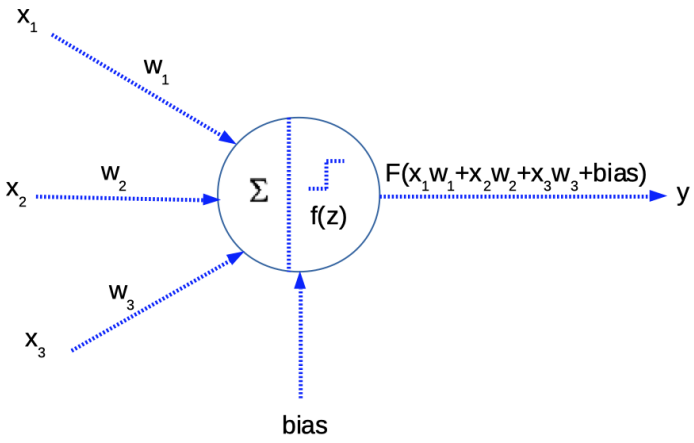


Figura 2.4. Representación gráfica de una neurona básica.

Las redes neuronales son excelentes para problemas de aprendizaje automático con dominios complejos. Los sistemas que se han explicado tienen un costo alto de cómputo, y las redes neuronales se pueden ajustar con técnicas similares menos costosas.

En este trabajo de investigación se aborda en capítulos posteriores el uso de técnicas DL para la clasificación de opiniones de los turistas. En el capítulo 4 en concreto se realizan comparativas de resultados entre los modelos más utilizados en la actualidad, como son las redes neuronales convolucionales y las redes recurrentes.

En este capítulo se han introducido el procesamiento del lenguaje natural y el problema del análisis de sentimiento. El objeto principal de este trabajo de investigación es el estudio de las diferentes técnicas para ana-

lizar el sentimiento expresado por los turistas mediante comentarios en redes sociales y portales especializado. La aplicación de técnicas de ML es uno de los campos de estudio con mucha actividad en la actualidad, debido a su gran utilidad para aprovechar la información masiva que circula en internet.

Como se ha visto, el aprendizaje supervisado ofrece técnicas diversas para poder abordar nuestro caso de estudio. En capítulos siguientes se estudiarán en profundidad diferentes técnicas y variantes de DL comparando los resultados en nuestro problema de aplicación. Ya se ha significado en varias ocasiones la importancia que tienen los datos de entrenamiento en los resultados obtenidos. El siguiente capítulo analiza diferentes aproximaciones para la obtención y codificación de los mismos.

Capítulo

3

Preparación de los datos

Todas las técnicas de aprendizaje automático requieren de un conjunto de datos de los cuales poder extraer conocimiento. Las nuevas tecnologías, el uso de internet, la proliferación de las redes sociales y su penetración en los hábitos de comunicación de la sociedad actual, así como la masiva creación de portales especializados en todos las disciplinas han abierto un universo de información muy útil para el desarrollo de las técnicas de aprendizaje automático.

Esta información, en mucho de los casos pública, es la fuente de la que deben aprender los algoritmos, sin embargo posee algunas características problemáticas que deben tenerse en cuenta:

- Es información no estructurada. Los datos no se encuentran organizados en bases de datos de fácil acceso, en el que podamos realizar consultas precisas y extraer aquellos campos útiles para nuestros objetivos.
- Las fuentes de información, aunque sean públicas, están pensadas para un propósito, y normalmente su exposición está preparada pa-

ra formar parte de un texto que debe ser leído por un usuario como parte de un documento o conjunto de datos que deben ser interpretados (ya sea en una página web, o en una aplicación para dispositivos móviles). A modo de ejemplo podemos pensar en los comentarios de unos turistas. Éstos no tendrían sentido si algún otro componente del documento no hace referencia al hotel o servicio al que está asociado.

- Bajo la prerrogativa de que la información es poder, y finalmente posibilidad de negocio, las compañías que hay detrás de las redes sociales y los portales especializados no están dispuestas a ofrecer este valor gratuitamente para que puedan ser extraídas de forma masiva, ya que esto daría ventaja a sus competidores.

- La velocidad de cambio en cualquier producto tecnológico es muy alta. Cuando has preparado tu entorno para poder acceder a una información existe una probabilidad alta de que la fuente de información haya experimentado algún cambio tecnológico que te haga realizar cambios (que pueden ser grandes) en tus sistemas de extracción.

En las siguientes secciones de este capítulo se explican en detalle las técnicas utilizadas en este trabajo para realizar una extracción y preparación de los datos que se utilizarán posteriormente en los diferentes modelos de aprendizaje máquina.

3.1. Extracción de los datos.

Antes de profundizar en las técnicas y herramientas utilizadas para la extracción de los datos, es necesario definir el dominio en el que se desea aplicar aprendizaje máquina y fijar el objetivo de las fuentes de información de las cuáles realizar la extracción.

Nuestro caso de estudio es el análisis de los sentimientos, y en concreto la clasificación de las opiniones de los turistas que visitan destinos turísticos de las Islas Canarias. El objetivo planteado es entrenar un modelo que pueda predecir si el texto escrito por un visitante expresa un sentimiento positivo o negativo, de forma que se puedan tomar acciones de fidelización o correctivas en base a los mismos.

Este tipo de problemas se clasifica entre los denominados de *aprendizaje supervisado*, que requieren un conjunto de datos que hayan sido previamente clasificados para realizar el entrenamiento del modelo y la validación del mismo.

Para poder satisfacer este requerimientos realizamos una búsqueda de las fuentes de información del dominio mencionado y centramos la misma en dos portales especializados, en los que turistas de habla inglesa comentaban su experiencia en hoteles y realizaban una valoración de los mismos. En concreto se analizaron las siguientes fuentes:

- www.booking.com (en su portal para Reino Unido www.booking.co.uk).
    - El comentario del usuario tiene una valoración numérica global de los servicios entre 0 y 10.

    - El autor del comentario tiene la posibilidad de separar el comentario en dos partes: una sección de texto para expresar aspectos positivos, y otra sección de texto para expresar aspectos negativos.

- www.tripadvisor.com (en su portal para Reino Unido www.tripadvisor.co.uk).
    - El comentario del usuario se evalúa usando burbujas o estrellas, asignado un valor entre 1 y 5.
    - Únicamente existe un campo de texto para expresar una opinión

Para extraer la información se desarrollaron en Python diferentes programas utilizando el framework Scrapy (Scrapinghub, n.d.), y tomando como base los desarrollados por MonkeyLearn (MonkeyLearn Inc., n.d.) que ofrece diferentes soluciones de aprendizaje máquina.

Los proyectos de extracción basados en el framework Scrapy tienen la siguiente estructura de Directorios: Carpeta principal del proyecto:

- fichero de configuración scrapy.cfg que contiene información básica sobre el proyecto que debe ejecutarse cuando se ejecute el comando scrapy (encargado de realizar la extracción) en esta carpeta.
- carpeta con el nombre del proyecto
    - fichero items.py que contendrá las clases que definen qué campos (items en Scrapy) y sus tipos que hay que extraer en los diferentes spiders que programemos.

- fichero middlewares.py donde se podrán definir funcionalidades personalizadas para procesar las respuestas que se envían a los spiders.
- fichero pipelines.py donde se pueden realizar acciones de los items extraídos, como tratamientos o almacenamiento en bases de datos.
- fichero settings.py donde podremos configurar diferentes parámetros de ejecución de los scripts de extracción, como número procesos concurrentes, retardos, o comportamientos de los accesos mediante http.
- Directorio spiders con tus scripts de extracción.
  - ficheros .py con mis spiders de extracción.

Cuando se ejecuta el comando *scrapy crawl nombre_spider* el framework Scrapy ejecuta el método *Request* a las direcciones web que hayamos definido en el atributo *start_urls* de nuestro spider y una vez vaya recibiendo la respuesta de cada una las pasa como argumentos al método *parse* de nuestro spider para realizar el procesamiento. La forma en la que Scrapy asocia el spider que debe ejecutar es mediante el atributo *name* que debe definirse como parte de la clase donde programamos nuestro spider.

Scrapy nos ofrece sin tener que programar utilidades para poder exportar los datos extraídos a un formato de fichero estándar para poder ser procesados posteriormente (CSV, XML, o JSON).

En concreto, en el apéndice de códigos A.1 podemos encontrar el script

items.py que muestra los campos que fueron extraídos para cada unos de los portales en estudio.

Los apéndices de código A.2 y A.3 muestran los códigos de los spiders que se programaron *tripadvisor_spider.py* y *booking_spider.py* respectivamente.

Para explicar el funcionamiento básico de los spider podemos fijarnos en el apéndice de código A.2 que contiene el script *tripadvisor_spider.py*.

Para ejecutar este script, desde la carpeta que contiene el proyecto Scrapy se ejecuta el comando:

scrapy crawl tripadvisor -o itemsTripadvisor.csv -s CLOSESPIDER_ITEMCOUNT=10000

Como se comentó anteriormente, Scrapy en este momento busca el spider dentro del proyecto cuya clase tiene un atributo con el nombre *tripadvisor*, que se ha especificado como parámetro, y que en nuestro ejemplo particular corresponde con la clase *TripadvisorSpider*. A continuación ejecuta un *Request* sobre todas las direcciones web que se encuentran en la lista *start_urls* de la propia clase, entregando la página obtenida en tal petición al método *parse*. En nuestro spider, hemos desarrollado tres métodos diferentes para estructurar la extracción:

- *parse*. Encargado de recorrer las páginas de la búsqueda realizada.
- *parse_hotel*. Encargado de recorrer cada uno de los comentarios del hotel.
- *parse_review*. Encargado de realizar la extracción de campos de cada

uno de los comentarios de los turistas

Esta estructura de código sirve de base para poder el desarrollo spiders de extracción de datos. Si se observa en detalle los métodos, se puede apreciar que el *Request* de la página devuelve un al que hemos llamado *response*. Este objeto contiene la respuesta en HTML de la página solicitada. La manera de extraer la información es mediante los métodos *css* y *xpath* del objeto *response*. CSS (Cascading Style Sheets, n.d.), hojas de estilo en casacada, es un lenguaje para definir la forma en la que se presentan las páginas en HTML, o en general cualquier interfaz de usuario. Mediante el método *css* del objeto *response* podemos buscar en la página devuelta campos de información y extraerlos para un posterior procesamiento. En nuestros códigos nos hemos decantado por la utilización de XPATH (XPATH Expressions, n.d.), que es un lenguaje para especificar rutas que permiten seleccionar uno o varios elementos de la página devuelta.

La búsqueda de las rutas *xpath* para acceder a los elementos es una de las labores más complicadas a la hora de realizar la extracción de información. Durante la realización de este proyecto se hizo uso de dos herramientas en la búsqueda de estos elementos:

- *Selector Gadget* (Cantino y Maxwell). Es una extensión para el navegador que permite inspeccionar los diferentes elementos de la página. Haciendo click en los diferentes elementos podemos obtener.
- Dentro del framework *Scrapy*, existe una herramienta, *shell* que es un intérprete de comandos Python para procesar mediante comandos individuales la respuesta (*response*) que se obtendría al utilizar el framework.

Utilizando de forma conjunta estas dos herramientas para buscar las rutas a los elementos, y para comprobar que utilizando esa ruta *xpath* se obtienen los elementos que se pretendían podremos completar nuestros spiders y obtener la información deseada.

Es importante reseñar que el desarrollo de los spiders es costoso en tiempo y esfuerzo, debido a que los portales especializados no estructuran la información de manera clara: ya sea por ocultación de la misma para herramientas de extracción automática, o bien porque los gestores de contenido usados para su implementación no lo hacen por defecto.

Además de la dificultad ya mencionada, la tecnología de los portales requiere de continuas actualizaciones, lo que implica que los scripts realizados para la extracción de la información sirven solo durante un corto espacio de tiempo. Los que se incluyen en los apéndices de código A.1, A.2 y A.3 se utilizaron en este proyecto para la generación de los conjuntos de datos de entrenamiento y test que se utilizan posteriormente en los experimentos. Lamentablemente, en el momento de escribir este documento la estructura de los portales cambió y ya no cumplían con su cometido.

Mediante la ejecución de los spiders se obtuvieron un total de 53.172 comentarios de turistas del portal de Tripadvisor y 37.616 de Booking.

3.2. Preparación de los datos.

En la sección anterior se explicaron las técnicas y herramientas utilizadas para extraer la información de los portales especializados Tripadvisor

y Booking. Como resultado de la recolección de datos se obtuvieron dos ficeros CSV (comma separated values) con la siguiente información (tal y como puede observarse en el fichero *items.py* del apéndice de códigos A.1):

- *Tripadvisor*.
  - Destino. Información texto sobre el destino turístico al que refiere el comentario.
  - Fecha del comentario. Fecha en la que el turista emitió su opinión.
  - Localización del autor del comentario. Información de texto sobre el origen del autor del comentario.
  - Valoración. Texto que representa la valoración del turista. En la página web el autor valora el destino con estrellas (1 a 5 estrellas). Esta valoración se traduce en un texto de la forma: ”3 of 5 bubbles”, significando esta valoración que el usuario ha valorado con 3 estrellas el destino.
  - Título. Texto que resume su comentario.
  - Contenido. Texto amplio en el que el turista describe su experiencia en el destino turístico.
- *Booking*.
  - Fecha del comentario. Fecha en la que el turista emitió su opinión.

- Localización del autor del comentario. Información de texto sobre el origen del autor del comentario.
- Valoración. Valor numérico con un decimal, entre 1 y 10, que expresa la valoración global del turista en el destino.
- Título. Texto que resume su comentario.
- Contenido Positivo. Texto amplio en el que el turista describe aquellos aspectos positivos de su experiencia en el destino turístico.
- Contenido Negativo. Texto amplio en el que el turista describe aquellos aspectos negativos de su experiencia en el destino turístico.

Como puede observarse, los datos obtenidos de diferentes portales tienen estructura y tipos diferentes, por lo que hay que hacer un trabajo de adaptación de los mismos a un conjunto de datos de entrenamiento y test útiles para el objetivo de clasificar opiniones de turistas de forma binaria (positivo o negativo). Como se aplicará en un modelo de aprendizaje supervisado, que trata de predecir si un comentario de texto de un turista es positivo o negativo, se requiere que ambos conjuntos de datos correspondan con la entrada y salida del modelo.

Por este motivo, los ficheros obtenidos de la extracción se convirtieron a un único fichero de datos con dos campos: Comentario del turista, sentido del comentario (si es positivo o negativo). Para ello se desarrollaron dos scripts que realizaban las siguientes transformaciones:

- En el caso de *Tripadvisor* se concatenó el título del comentario y el contenido para formar el campo correspondiente al comentario del turista. Por otro lado, si la valoración emitida era de 4 o 5 estrellas se etiquetaba el comentario como positivo ("Good" en el script), y si por el contrario tenía una valoración de 1 o 2 estrellas se etiquetaba como negativo ("Bad" en el script). Se descartaron aquellos comentarios cuyo sentido era intermedio.

- En el caso de *Booking* se comprobaba el campo Valoración. Si era mayor que 5 se concatenaba el título del comentario con el campo contenido positivo para formar el comentario del turista, y se etiquetaba el comentario como positivo. Si la valoración era menor de 5 se concatenaba el título del comentario con el campo contenido negativo para formar el comentario del turista, y se etiquetaba el comentario como negativo.

El script *Preparar_tripadvisor_csv.py* del apéndice de códigos A.4 y el script *Preparar_booking_csv.py* del apéndice de códigos A.5 muestran el detalle de cómo se realizaron las transformaciones mencionadas.

El resultado final de la preparación de datos son tres conjuntos diferentes de datos (*data sets*): uno de entrenamiento con el mayor número de muestras (tratando que este lo más balanceado posible, es decir, aproximadamente la mitad de las muestras etiquetadas como positivas y la otra mitad como negativas), y dos conjuntos de test, del que conozcamos la respuesta, para validar el modelo (hemos separados los datos de validación en dos conjuntos, uno para para probar el modelo tras su entrenamiento, y otro para predicciones individuales y estadísticas).

En las secciones posteriores se hace un recorrido por las técnicas de codificación más utilizadas en problemas de aprendizaje profundo, explicando sus principales características haciendo uso de un modelo básico de clasificación común basado en una red neuronal básica perceptrón multicapa (modelos más complejos son introducidos en posteriores capítulos).

3.3. Bag of Words (BoW).

Las técnicas de aprendizaje de máquinas utilizadas para problemas de procesamiento de lenguaje natural requieren de la representación de los datos en formato numérico. Como se ha explicado en la sección 3.2, los datos a utilizar en los modelos y experimentos de este trabajo son los comentarios extraídos de nuestros visitantes expresados en forma de textos y su valoración mediante una etiqueta.

Bag of Words (BoW) es una técnica ampliamente usada para la representación de textos mediante vectores numéricos, de forma que puedan ser utilizados en modelos matemáticos.

Para poder implementar una representación basada en BoW se necesita tener un vocabulario de palabras que defina el conjunto de las mismas que se va a reconocer y una métrica sobre la frecuencia de apariciones de esas palabras en los textos utilizados.

El nombre de la representación nos da una idea visual de su principal característica: el orden de aparición de las palabras no es relevante (se in-

troducen en una bolsa y finalmente lo importante es cuántas palabras de cada tipo hay en el texto).

Los pasos necesarios para poder usar esta representación vectorial de textos serían:

- Generar una lista de todas las palabras que son usadas en la totalidad de los comentarios.
- Limpiar la lista generada suprimiendo signos de puntuación y caracteres no alfabéticos, convirtiendo además todas las palabras a minúsculas para que no sean interpretadas como distintas algunas palabras iguales con algún carácter capitalizado. Esta lista depurada será lo que se conocerá como **vocabulario**.
- A cada palabra del vocabulario se le asignará un índice. El número total de palabras del vocabulario será la dimensión final de todos los vectores que representarán a cada uno de los comentarios tratados.
- La representación final de cada comentario será un vector de dimensión fija (igual al número de palabras del vocabulario). El vector tendrá en su componente *i-ésima* un número n, que corresponderá con el número de apariciones en el comentario de la palabra del vocabulario correspondiente a ese índice.

La librería Keras Keras API (n.d.) ofrece una clase de objetos denominada Tokenizer que facilita esta labor. Con el fin de ilustrar cómo funciona la técnica se extraen del conjunto de entrenamiento los siguientes textos tabla 3.1.

Tabla 3.1. Ejemplo comentarios BoW

Comentario
Good location with nice rooms. Staff very friendly and helpful. Clean rooms and reasonably priced.
Great seawater pool. Friendly staff and food ok. Cheap and cheerful, you get what you pay for.
Clean comfortable hotel. Breakfast was very good and most of the staff were excellent.
The hotel and pools very clean. Room basic but you get what you pay for well worth the money.
It is in a nice location not far from the beach. Rooms not clean, not a four star hotel needs to be closed and renovated.
Very poor hotel and service. Very rude service, dirty and old equipment,daily changed the password to the Internet. Cold water in the pool.
Never again! Everything was terrible. Food was bad, rooms was dirty and not like on photos. Every night there was loud noise. For internet, which didnt work at rooms, we needed to pay and it was really really slow.
Hotel in poor state of repair. Flies all over food in restaurant. Condensation and mildew over walls in restaurant. Pool dirty. Attitude of reception staff very unhelpful. Cockroaches everywhere. Worst hotel ever.

Los índices generados por la clase Tokenizer con los textos de la tabla 3.1 se muestran en la tabla 3.2 . Como se puede observar, el índice 0 lo reserva la librería y no corresponde con ninguna palabra del vocabulario (se utilizará posteriormente como índice de relleno para igualar los comentarios codificados en longitud en otras técnicas), lo que implicará que

todos los vectores de codificación correspondientes con los comentarios comenzarán con cero.

Tabla 3.2. Índices y vocabulario BoW

Índice	Palabra	Índice	Palabra	Índice	Palabra	Índice	Palabra
1	and	28	service	55	far	82	there
2	the	29	internet	56	from	83	loud
3	very	30	really	57	beach	84	noise
4	hotel	31	over	58	four	85	which
5	was	32	restaurant	59	star	86	didnt
6	rooms	33	with	60	needs	87	work
7	in	34	helpful	61	be	88	at
8	staff	35	reasonably	62	closed	89	we
9	clean	36	priced	63	renovated	90	needed
10	you	37	great	64	rude	91	slow
11	not	38	seawater	65	old	82	state
12	pool	39	ok	66	equipment	93	repair
13	food	40	cheak	67	daily	94	flies
14	pay	41	cheerful	68	changed	95	all
15	for	42	comfortable	69	password	96	condensation
16	of	43	breakfast	70	cold	97	mildew
17	to	44	most	71	water	98	walls
18	dirty	45	were	72	never	99	atitutde
19	good	46	excellent	73	again	100	reception
20	location	47	pools	74	everything	101	unhelpful
21	nice	48	room	75	terrible	102	cockroaches
22	friendly	49	basic	76	bad	103	everywhere
23	get	50	but	77	like	104	worst
24	what	51	well	78	on	105	ever
25	it	52	worth	79	photos		
26	a	53	money	80	every		
27	poor	54	is	81	night		

Como muestra de cómo se realiza la codificación tomamos el primer comentario: *"Good location with nice rooms. Staff very friendly and helpful. Clean rooms and reasonably priced."*

[0. 2. 0. 1. 0. 0. 2. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

Como se puede observar, el índice 1, que corresponde a la palabra *and* (ver tabla 3.2) tiene una frecuencia de aparición de 2 así como el índice 3 que corresponde a la palabra *very* tiene una frecuencia de 1.

La técnica de representación BoW presenta como principal ventaja su simplicidad. Se trata de una técnica intuitiva en la que con una buena elección del vocabulario, enfatizar en aquellas palabras del dominio (turismo en nuestro ejemplo) que aportan significado. Sin embargo, y como se puede deducir del ejemplo mostrado presenta algunas debilidades difíciles de superar:

- Con un conjunto de datos de tan solo 8 comentarios, y tras realizar algunas transformaciones de limpieza, nos ha quedado como resultado un vocabulario de 105 palabras. Esto conllevará que la codificación de cada comentario resultará en un vector de 105 dimensiones. Una elección automatizada del vocabulario como la que hemos realizado en una muestra mucho mayor (como por ejemplo un conjunto de entrenamiento para una red neuronal) podría resultar en un vocabulario gigantesco que aumentaría significativamente la complejidad computacional. Existen algunas técnicas automatizadas para reducir el tamaño del vocabulario, buscando palabras sinónimas, o palabras derivadas de una misma raíz (stemming).

- El contexto es muy importante en el significado de una palabra. La

técnica BoW no representa en ninguna forma el contexto de una palabra ni el orden de aparición de sus vecinas. Técnicas similares, en las que se representan la frecuencia de aparición de grupos de n-palabras tratan de resolver este problema (bi-gram para grupos de dos palabras, tri-gram para grupos de tres palabras, o n-gram en general).

Para evitar tener un vocabulario excesivamente grande, es necesario aplicar algunas transformaciones. Una manera de reducirlo es suprimir de los comentarios de entrenamiento lo que se conoce como "stop words" (en español es más usado el término de "palabras vacías"). Stop words son aquellas palabras cuya aportación al significado de la frase es escasa. Suelen ser artículos, preposiciones, pronombres, etc. Sería ideal disponer de un vocabulario específico para el dominio del problema en estudio, pero a falta del mismo podemos, mediante herramientas ya desarrolladas, generar nuestro propio vocabulario, y eliminar mediante una función algunas de estas palabras vacías. La librería NLTK (NLTK Project, n.d.) nos provee de una lista de las mismas en diferentes idiomas.

En la tabla 3.3 se observa que tras realizar las transformaciones comentadas en el párrafo anterior, las quince primeras palabras más usadas adquieren un mayor significado en el dominio de estudio, el turismo.

Experimentos con BoW.

Para poder comparar las diferentes técnicas de codificación utilizaremos una red neuronal simple del tipo *perceptrón multicapa* (Widrow y Lehr, 1990) con el fin de poder realizar una clasificación binaria de los comen-

Tabla 3.3. Comparativa 15 palabras más usadas. Sin procesar vocabulario vs Procesándolo

Indice	Sin procesar	Procesando
1	the	hotel
2	and	room
3	a	good
4	to	staff
5	was	food
6	we	would
7	in	pool
8	of	great
8	hotel	holiday
10	for	clean
11	is	one
12	i	rooms
13	it	day
14	were	stay
15	this	stayed

tarios de los turistas.

En la fig. 3.1 se muestra el esquema de la red elegida. Se trata de una red neuronal perceptrón multicapa, con una capa de entrada que tendrá tantas neuronas como la longitud de la entrada que elijamos en los experimentos, una capa oculta formada por cincuenta neuronas con función de activación *relu*, y una capa de salida formada por una única neurona y función de activación *sigmoid* ya que vamos a realizar una clasificación binaria de los comentarios. Se ha elegido *relu* como función de activación en la capa oculta por eficiencia computacional y por evitar un mayor des-

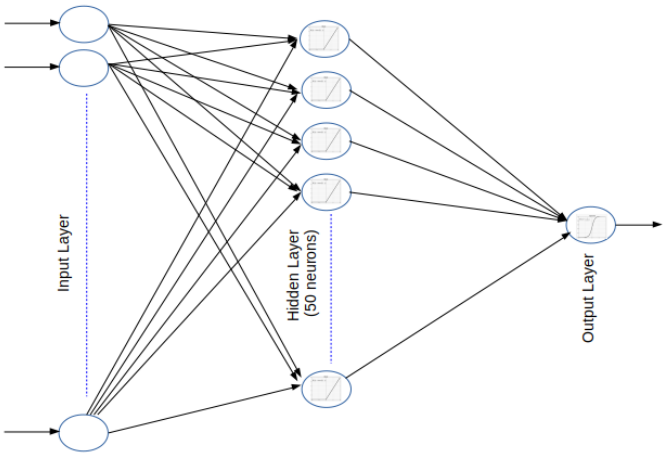


Figura 3.1. Red Perceptrón Multicapa usada en experimentos.

vanecimiento del gradiente (Glorot *et al.*, 2011) (Nwankpa *et al.*, 2018).

La capa de entrada a la red, capa visible, tiene tantas neuronas como dimensiones tiene el vector que representa a una entrada del conjunto de datos de entrenamiento, de manera que la componente x_i es la única entrada de la neurona *i-ésima* de esta capa. Las neuronas de la capa de entrada no son perceptrones, simplemente dejan pasar a la salida de la misma el valor que se presenta en su entrada. La librería Keras (Keras API, n.d.) utiliza como estructura básica de datos un *tensor*, que es un *array* de múltiples dimensiones (implementados en *Numpy* (Numpy Developers, n.d.)). Las propiedades que definen un tensor son las siguientes:

- Su dimensión (*ndim*). En el caso de tratarse un único número (un escalar), se dice que el tensor es un *scalar* o tensor 0D, y tendrá di-

mensión 0. Si se trata de un vector simple (*array* de números), de le dice vector o tensor 1D y tendrá dimensión 1. Si se trata de un vector de vectores, se le conoce como *matrix* o tensor 2D y tendrá dimensión 2, y así sucesivamente.

- Su forma (*shape*) corresponde con el número de elementos que tendrá en cada una de las dimensiones. Se representa con una *tupla* (con *ndim* componentes), así que si el tensor era un escalar, como *ndim* es 0, la tupla es (). Si se trata de un vector de 100 componentes, su *shape* será de (100), y si se trata de una matriz de 20x50, su *shape* será (20,50).

- El tipo de los datos (*dtype*). Es el tipo de los datos que contiene el tensor, que por lo general es un booleano, entero, flotante o complejos (pareja de flotantes) en diferentes precisiones.

Los experimentos que se realizan en este trabajo utilizarán un conjunto de 9640 comentarios de turistas etiquetados como "Good" o "Bad" en su fase de entrenamiento, seleccionados de la extracción de datos comentada en la sección 3.2. Como preparación previa a la construcción del modelo se eliminaron las *stopwords* del vocabulario, y haciendo uso de la clase *Tokenizer* se realizó una codificación de los comentarios utilizando la técnica BoW para diferentes tamaños de vector (100, 200, 400, 800 y 1600). Para ilustrar la codificación véase el siguiente ejemplo seleccionado de la muestra:

Comentario original: *"Good 3star hotel. Just returned from hotel Las Vegas and it is a very good 3 star hotel. After reading previous reviews we asked for a top floor room to get better views. We were given room 1301 .We got the lift to the top*

floor and then had to walk up another floor with our cases to th 13th. On entering the room we found there were three steps leading down to a rather cramped bedroom and bathroom that appeared a little dated, however after opening the patio doors we stepped onto an amazing roof terrace with fantastic 360 degree views of the sea and mountains."

Comentario normalizando a minúsculas y eliminando "stopwords": *"good 3star hotel. returned hotel las vegas good 3 star hotel. reading previous reviews asked top floor room get better views. given room 1301 .we got lift top floor walk another floor cases th 13th. entering room found three steps leading rather cramped bedroom bathroom appeared little dated, however opening patio doors stepped onto amazing roof terrace fantastic 360 degree views sea mountains."*

Vector:

[0. 3. 3. 2. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 3. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

El código Python utilizado para generar este ejemplo puede consultarse en el apéndice de códigos A.6.

La librería Keras Keras API (n.d.) ofrece herramientas para modelar de una manera sencilla redes neuronales. La clase *Sequential* permite ir agregando capas a nuestra red mediante el método *add*, en el que se deberá especificar el tipo de capa (densa en nuestro modelo para la capa oculta), el número de neuronas que tiene, y el número de entradas (siendo únicamente necesario especificarlo en la primera capa oculta, ya que con el

Tabla 3.4. Ejemplo codificación BoW

Componente	Palabra	Apariciones
1	hotel	3
2	room	3
3	good	2
19	get	1
36	star	1
41	3	1
50	got	1
57	sea	1
59	floor	3
62	walk	1
66	reviews	1
74	little	1
79	found	1

resto de capas la librería es capaz de deducirlo). El proceso de compilación del modelo convierte esta estructura de capas en estructuras matriciales eficientes. Para realizar la compilación se debe definir la función de optimización a utilizar en el entrenamiento (*optimizer*), así como la función de pérdida (*loss*) utilizara para realizar la evaluación de la diferencia entre el valor estimado por el modelo y el realmente etiquetado.

La figura 3.2 muestra la representación que hace Keras del modelo. En concreto, el representado en la misma es la implementación de la figura 3.1. Como se puede observar, el tensor de entrenamiento que entra en la red tiene como *shape* (9640,100) lo que significa que entrarán a la misma para entrenar 9640 muestras que han sido codificadas en vectores de

100 componentes. Esto implica que el número de neuronas de la capa de entrada será 100. La capa oculta como ya se explicó consta de 50 neuronas, que se conectan a la capa de salida, que tiene una única neurona. Es importante mencionar (ya que incide en la cantidad de cálculos que den hacerse para entrenar el modelo) el número de parámetros que se deben entrenar. En el ejemplo mostrado es 5.101 parámetros. Esta cantidad es el resultado de hacer el siguiente cálculo: cada una de las 100 neuronas de la capa de entrada se conectan a las 50 neuronas de la capa oculta (al tratarse de una capa de tipo densa), lo que implica que habrá $50x100 = 5000$ pesos que entrenar, más las 50 entradas de tipo *bias*, lo que da un total 5050 parámetros en la capa oculta. Cada neurona de la capa oculta se conecta a la neurona de la capa de salida (al ser una capa densa), lo que implica que habrá otros 50 pesos adicionales a entrenar, a los que nuevamente hay que añadir la entrada *bias* de la neurona de la capa de salida.

```
Tensor x_train.ndim =  2  x_train.shape =  (9640, 100)  x_train.dtype =  float64
Tensor x_test.ndim =  2  x_test.shape =  (80, 100)  x_test.dtype =  float64
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_21 (Dense)             (None, 50)                5050
_________________________________________________________________
dense_22 (Dense)             (None, 1)                 51
=================================================================
Total params: 5,101
Trainable params: 5,101
Non-trainable params: 0
```

Figura 3.2. Modelo MLP ejemplo en Keras.

La librería Keras permite implementar cuatro diferentes variantes de codificación basadas en la técnica BoW. Esto se realiza mediante el parámetro *mode* del método *texts_to_matrix* de la clase *Tokenizer*.

- Modo *binary*. Si la plabra del vocabulario correspondiente al índice *i* aparece en el texto, se pondrá un 1 en la componente *i-ésima* del vector, y un 0 en caso contrario.
- Modo count. En la componente *i-ésima* del vector se representará un contador entero con el número de veces que aparece la palabra del vocabulario correspondiente al índice *i*. Este es el método que se ha explicado en los ejemplos codificados.
- Modo *freq*. En la componente *i-ésima* del vector se representará el resultado de dividir el número de veces que aparece la palabra del vocabulario correspondiente al índice *i* entre el número de palabras del texto a codificar. Es igual al modo *count* dividiendo por el número de palabras del texto a codificar. Se utiliza este modo para hacer relativo el peso de las palabras en comentarios más largo.
- Modo *tfidf*. Conocido como *tf-idf*, se calcula como la frecuencia (número de veces que aparece la palabra en el texto a codificar dividido por el número de palabras del texto), multiplicado por el logaritmo del producto del número comentarios usados en el entrenamiento y el número de veces que aparece la palabra en ellos (Term Frequency-Inverse Document Frequency, n.d.). Se trata de una medida estadística para evaluar cómo de importante es una palabra en el comentario, de forma que su importancia crezca proporcionalmente al numero de veces que aparece en el comentario, pero compensado si aparece muchas veces en el total de las muestras de entrenamiento.

Para poder realizar un estudio completo de la incidencia de la codificación BoW en el modelo planteado se realizaron todas las posibles combi-

naciones de los siguientes parámetros:

- Realizando preprocesamiento previo o no de los comentarios antes de construir la bolsa de palabras (eliminando las *stop-words*).
- Modificando el tamaño de vector codificado (eligiendo siempre para la codificación en el *Tokenizer* las palabras más usadas) entre 100, 200, 400, 800, 1600 y 20880 (número total de palabras del vocabulario).
- Utilizando los diferentes modos presentados: *binary, count, freq, tf-idf.*

La tabla 3.5 muestra los resultados obtenidos. La columna encabezada como *% accuracy*, representa la precisión o exactitud del modelo, que se define como el porcentaje de aciertos en la predicción sobre el total de predicciones realizadas.

El código utilizado para realizar las comparativas mostradas en la tabla 3.5 se muestra en el apéndice de códigos A.7. Como se puede ver en el cuerpo principal del mismo se realizan iteraciones sobre todas las modalidades de codificación *BoW* y sobre las dimensiones de la capa de entrada (tamaño del vocabulario).

En la figura 3.3, que realiza la comparación entre los modos de BoW y los tamaños de los vectores de entrada realizando un procesamiento previo de los comentarios, se puede observar cómo aumentar el número de palabras a considerar en el vocabulario (seleccionando las más usadas) aumenta, pero que a partir de las 800 palabras, esta aportación no tiene demasiada relevancia (casi hay superposición de las tres líneas superio-

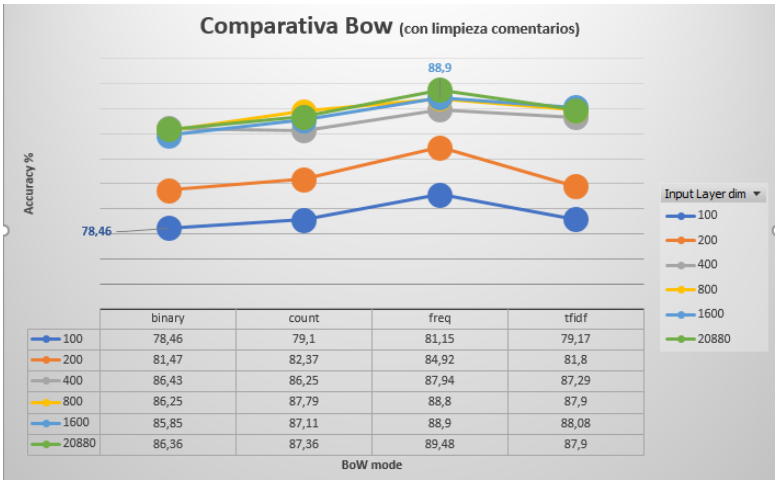
res). Del mismo se aprecia, que para este dominio concreto el modo *freq* es el que tiene mejores resultados para todos los tamaños de entrada, alcanzando un 88,9 % de exactitud para la longitud de 1600 palabras.

En la figura 3.4, que realiza la misma comparativa pero esta vez sin realizar ningún tratamiento sobre los comentarios, se puede observar que para el problema concreto en estudio el tamaño de entrada que mejor ha funcionado es 800, obteniendo una exactitud de 91,1 %. Nuevamente el modo *freq* ha sido el más certero para todos los tamaños de entrada.

Comparando los resultados de ambas figuras 3.3 y 3.4 se puede concluir que no existen diferencias significativas, por lo que para este experimento concreto no merece la pena realizar el preprocesamiento de comentarios, obteniendo incluso mejor exactitud sin limpiar comentarios y con una entrada de datos menor.

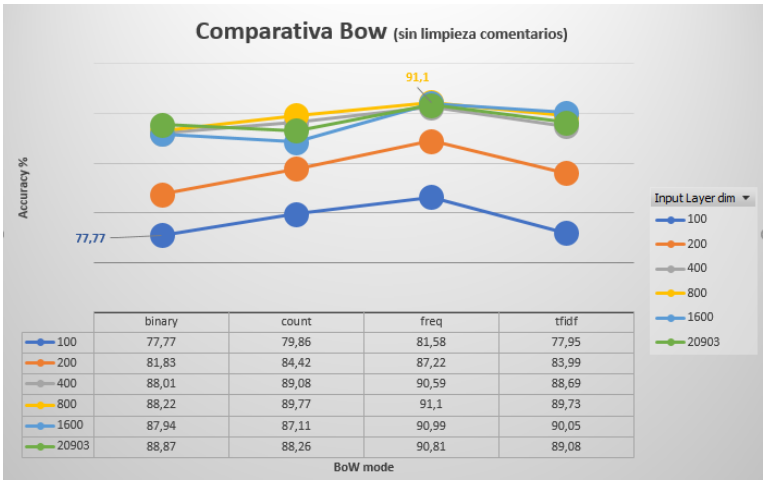
En relación al experimento global con el modelo utilizado, y en este dominio podemos concluir lo siguiente: El modo y el tamaño de la entrada tienen una relevancia notable en el modelo. La diferencia en puntos entre el modelo más exacto y el menos exacto utilizando esta codificación es de 13,33 % que debe tenerse en cuenta.

La tabla 3.5 muestra los resultados obtenidos.



	binary	count	freq	tfidf
100	78,46	79,1	81,15	79,17
200	81,47	82,37	84,92	81,8
400	86,43	86,25	87,94	87,29
800	86,25	87,79	88,8	87,9
1600	85,85	87,11	88,9	88,08
20880	86,36	87,36	89,48	87,9

Figura 3.3. Comparativa BoW. Limpiando comentarios



	binary	count	freq	tfidf
100	77,77	79,86	81,58	77,95
200	81,83	84,42	87,22	83,99
400	88,01	89,08	90,59	88,69
800	88,22	89,77	91,1	89,73
1600	87,94	87,11	90,99	90,05
20903	88,87	88,26	90,81	89,08

Figura 3.4. Comparativa BoW. Sin Limpiar comentarios

Tabla 3.5. Experimentos BoW

Experiment	Input Layer dim	Bow Codif	Clean review	Train params	% acc indiv test
Bow-1	20903	freq	no	1045251	90,81
Bow-2	1600	freq	no	80101	90,99
Bow-3	800	freq	no	40101	91,1
Bow-4	400	freq	no	20101	90,59
Bow-5	200	freq	no	10101	87,22
Bow-6	100	freq	no	5101	81,58
Bow-7	20903	tfidf	no	1045251	89,08
Bow-8	1600	tfidf	no	80101	90,05
Bow-9	800	tfidf	no	40101	89,73
Bow-10	400	tfidf	no	20101	88,69
Bow-11	200	tfidf	no	10101	83,99
Bow-12	100	tfidf	no	5101	77,95
Bow-13	20903	count	no	1045251	88,26
Bow-14	1600	count	no	80101	87,11
Bow-15	800	count	no	40101	89,77
Bow-16	400	count	no	20101	89,08
Bow-17	200	count	no	10101	84,42
Bow-18	100	count	no	5101	79,86
Bow-19	20903	binary	no	1045251	88,87
Bow-20	1600	binary	no	80101	87,94
Bow-21	800	binary	no	40101	88,22
Bow-22	400	binary	no	20101	88,01
Bow-23	200	binary	no	10101	81,83
Bow-24	100	binary	no	5101	77,77
Bow-25	20880	freq	yes	1044101	89,48
Bow-26	1600	freq	yes	80101	88,9
Bow-27	800	freq	yes	40101	88,8
Bow-28	400	freq	yes	20101	87,94
Bow-29	200	freq	yes	10101	84,92
Bow-30	100	freq	yes	5101	81,15
Bow-31	20880	tfidf	yes	1044101	87,9
Bow-32	1600	tfidf	yes	80101	88,08
Bow-33	800	tfidf	yes	40101	87,9
Bow-34	400	tfidf	yes	20101	87,29
Bow-35	200	tfidf	yes	10101	81,8
Bow-36	100	tfidf	yes	5101	79,17
Bow-37	20880	count	yes	1044101	87,36
Bow-38	1600	count	yes	80101	87,11
Bow-39	800	count	yes	40101	87,79
Bow-40	400	count	yes	20101	86,25
Bow-41	200	count	yes	10101	82,37
Bow-42	100	count	yes	5101	79,1
Bow-43	20880	binary	yes	1044101	86,36
Bow-44	1600	binary	yes	80101	85,85
Bow-45	800	binary	yes	40101	86,25
Bow-46	400	binary	yes	20101	86,43
Bow-47	200	binary	yes	10101	81,47
Bow-48	100	binary	yes	5101	78,46

3.4. Word Embedding.

La técnica de codificación *Word Embedding* trata de resolver algunos de los problemas de la técnica BoW. Como se explicará en secciones posteriores consiste en representar de forma vectorial las palabras que conforman un comentario. Antes de profundizar más en la técnica es necesario introducir el concepto de *secuencia*, ya que previo a la aplicación de las técnicas de *embedding* es necesario transformar los textos en ellas.

Conversión de textos a secuencias de enteros

Utilizando las mismas herramientas que se presentaron en la sección 3.3 donde se explicaba la técnica BoW, se construirá un vocabulario con el conjunto de todas las palabras que se reconozcan en los comentarios del conjunto de entrenamiento. A cada palabra del vocabulario se le asignará un número entero (1 para la más frecuente, 2 a la siguiente y así de forma sucesiva). Para codificar un texto en secuencias se deberá dividirlo en palabras, construyendo un vector en el que cada componente corresponde con cada palabra del texto, en el mismo orden en el que aparecen en el texto, y sin importar si se repite. Posteriormente, cada palabra del vector se sustituirá por el número entero por el que se identificó al construir el vocabulario. Como se puede observar, a diferencia de la técnica BoW, en una secuencia sí está representado el orden en el que aparecen las palabras dentro de cada comentario. Los pasos necesarios para construir las secuencias son los siguientes:

- Generar una lista de todas las palabras que son usadas en la totali-

dad de los comentarios.

- Limpiar la lista generada suprimiendo signos de puntuación y caracteres no alfabéticos, convirtiendo además todas las palabras a minúsculas para que no sean interpretadas como distintas algunas palabras iguales con algún carácter capitalizado. Esta lista depurada será lo que se conocerá como *vocabulario*.
- A cada palabra del vocabulario se le asignará un *índice*.
- La representación final de cada comentario será un vector con tantas dimensiones como palabras del vocabulario tenga ese comentario. El vector tendrá en su componente *i-ésima* un número *n*, que corresponderá con el entero unívoco (*índice*) que se asignó a esa palabra en el vocabulario.

Nuevamente, mediante la clase de objetos *Tokenizer* de la librerías Keras (Keras API, n.d.) podemos desarrollar un programa que automatice los pasos descritos anteriormente. El siguiente ejemplo muestra la codificación de textos en secuencias con uno de los comentarios de muestra que se presentaron en la tabla 3.1. El diccionario generado con estos comentarios se mostró anteriormente en la tabla 3.2.

Comentario: Good location with nice rooms. Staff very friendly and helpful. Clean rooms and reasonably priced.

Vector codificado: [19, 20, 33, 21, 6, 8, 3, 22, 1, 34, 9, 6, 1, 35, 36]

Correspondencia de palabras con índices:

[19 - good, 20 - location, 33 - with, 21 - nice, 6 - rooms, 8 - staff, 3 - very, 22

- friendly, 1 - and, 34 - helpful, 9 - clean, 6 - rooms, 1 - and, 35 - reasonably, 36 - priced]

El código utilizado para generar la secuencia mostrada en este ejemplo es el correspondiente al apéndice de códigos A.8.

Como se puede observar la técnica también presenta algunas dificultades que hay que solucionar:

- Es necesario realizar un tratamiento de los datos al construir el vocabulario, de forma que la misma palabra escrita con mayúsculas o minúsculas no sean consideradas diferentes. Afortunadamente el método *fit_on_texts* de la clase *tokenizer* ya realiza estas labores (aparte de eliminar símbolos).
- El vocabulario se construyó usando las palabras del conjunto de datos de entrenamiento. Sin embargo, cuando se vaya a utilizar la técnica de codificación con un nuevo comentario, podría darse el caso de que usa una palabra inexistente en el vocabulario. Si este fuera el caso, el método *texts_to_sequences* de la clase *tokenizer* ignoraría esta palabra a la hora de codificar.
- El objeto de codificar los textos es que puedan ser usados en modelos basados en redes neuronales. Las redes esperan vectores numéricos de un número fijo de componentes en su capa de entrada, mientras que la técnica tal y como se ha mostrado codifica en vectores de tantas componentes enteras como palabras diferentes del vocabulario contiene.

Para resolver esta última dificultad existen varias soluciones:

- Limitar el número de componentes a la longitud en palabras del comentario más corto del conjunto de entrenamientos. Esta posibilidad no parece la más idónea, ya que un comentario corto limitaría la información en el resto del conjunto.

- Codificar los textos en vectores que tengan tantos componentes como palabras tenga el comentario más largo del conjunto de datos de entrenamiento. Para completar la codificación de los comentarios cortos, rellenaríamos las componentes del vector codificado con *ceros* hasta alcanzar el número de componentes que ha fijado el comentario más largo. A esta técnica se le conoce como *padding*, y puede hacerse añadiendo los ceros al principio o al final del vector. El uso del *padding* es el motivo por el cual el primer índice válido del vocabulario era *1*, ya que reservaba el entero *0* para poder ser usado como número de relleno y que no se confundiera con una palabra.

- Fijar el tamaño de los vectores codificados, a un número asumible por el modelo, de forma que rellenemos mediante *padding* los comentarios más cortos que este tamaño, y limitemos los comentarios largos a éste (con el riesgo que la parte eliminada sea la que tuviera más valor semántico).

Definición de Word Embedding.

Word Embedding es una técnica de codificación para representar palabras por vectores de valores reales de un espacio predefinido, dando lugar a una representación distribuida en vez de una distribución dispersa como lo hacía BoW.

La principal característica de las técnicas basadas en *Word Embedding* es que es una representación que se aprende, de forma que palabras con un significados similares tengan una representación vectorial similar. Uno de los beneficios de usar una representación distribuida como *Word Embedding* es que se reduce de forma considerable el tamaño de los vectores en la capa de entrada, y por consiguiente el número de parámetros para entrenar. La capa de entrada tendrá la *secuencia* de texto codificada como se ha visto en la sección anterior (limitada como máximo al tamaño del comentario más largo) en el que cada componente se codificará con un vector de dimensión fija.

Existen diferentes maneras de implentar la técnica de *Word Embedding*: aprendiendo la codificación (junto al propio modelo de red neuronal o aparte), o bien utilizar una codificación que ha sido entrenada previamente.

Usando embedding para codificar textos de entrada a una red neuronal.

Para utilizar las técnicas de *embedding* en una red de clasificación como la que se presentó en la figura 3.1 es necesario introducir alguna capa más antes de la capa oculta. Para entender la arquitectura final de la red, es importante introducir lo que se conoce como la *matriz de embedding*.

Supongamos que tenemos un *corpus* simple con los siguientes comentarios: *['hotel was nice', 'good food', 'hotel nice pool']*. Si generamos un tokenizador partiendo de las mismas podríamos obtener un vocabulario como el mostrado en la tabla 3.6 en el que a cada palabra se le asigna un índice

(empezando por 1, que se asigna a la palabra más usada).

Tabla 3.6. Vocabulario

índice	palabra
1	hotel
2	nice
3	was
4	good
5	food
6	pool

Si construimos las secuencias correspondientes a cada uno de los comentarios, tal y como se explicó en la sección 3.4 *conversión de textos a secuencias de enteros* quedarían codificados de la siguiente manera:

"hotel was nice"[1,3,2] "good food"[4,5] "hotel nice pool"[1,2,6]

Sin embargo, como ya se introdujo anteriormente, esta codificación no es adecuada para modelos de redes neuronales, ya que todos los comentarios no tienen la misma longitud. Supongamos que utilzamos una técnica de *padding* rellenando con el número *0* por el final las secuencias, hasta alcanzar la longitud de la secuencia más larga. De esta forma las codificaciones cambiarían:

"hotel was nice"[1,3,2] "good food"[4,5,0] "hotel nice pool"[1,2,6]

Una *matriz de embedding* sirve para codificar cada palabra del vocabulario, en un vector de números reales de dimensión fija. De forma, que en la fila *i-ésima* de la matriz estará la codificación correspondiente a la palabra *i-ésima* del vocabulario. Es importante para el cálculo de paráme-

tros a entrenar, tener en cuenta que la *matriz de embedding* tendrá *(tamaño del vocabulario + 1)* filas y tantas columnas como dimensiones del espacio se hayan fijado para realizar la codificación. El motivo de añadir un *1* al número de filas de la matriz, es para poder representar en la fila *0* la codificación correspondiente para el *padding* de las secuencias. Supongamos a modo de ejemplo la siguiente *matriz de embedding* fijada para un espacio vectorial de dimensión *5* (que definirá el número de columnas de la matriz):

$$\begin{bmatrix} 0{,}0 & 0{,}0 & 0{,}0 & 0{,}0 & 0{,}0 \\ 1{,}1 & 1{,}2 & 1{,}3 & 1{,}4 & 1{,}5 \\ 2{,}1 & 2{,}2 & 2{,}3 & 2{,}4 & 2{,}5 \\ 3{,}1 & 3{,}2 & 3{,}3 & 3{,}4 & 3{,}5 \\ 4{,}1 & 4{,}2 & 4{,}3 & 4{,}4 & 4{,}5 \\ 5{,}1 & 5{,}2 & 5{,}3 & 5{,}4 & 5{,}5 \\ 6{,}1 & 6{,}2 & 6{,}3 & 6{,}4 & 6{,}5 \end{bmatrix}$$

El vector de codificación para el *padding* será el *[0.0,0.0,0.0,0.0,0.0]* mientras que el vector de codificación para la palabra *nice* que tiene el índice *2* será *[2.1,2.2,2.3,2.4,2.5]*. La librería Keras permite crear modelos con capas de *embedding*, en el que la entrada a la capa es la *secuencia* de cada uno de los comentarios y la salida de la capa es la matriz resultante de multiplicar la secuencia por la matriz de embedding. Como la salida de la capa de *embedding* es una matriz, no se puede introducir directamente al modelo de la red neuronal que se presentó en la figura 3.1, por lo que será también necesario introducir una capa de aplanado (*flatten*) cuya única misión es transforma la matriz en un vector unidimensional tomando las filas desde la primera a la última. La figura 3.5 representan las capas que deben

añadirse para poder utilizarse en una red neuronal como la utilizada en los experimentos.

El programa *Embedding_Flatten_example.py* del apéndice de códigos A.9 sirve de muestra para la codificación del corpus utilizado en el ejemplo.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_6 (Embedding)      (None, 3, 5)              35
_________________________________________________________________
flatten_6 (Flatten)          (None, 15)                0
=================================================================
Total params: 35
Trainable params: 0
Non-trainable params: 35
_________________________________________________________________
```

Figura 3.5. Modelo ejemplo de embedding.

Como se puede observar la salida de la capa de *Embedding* es un tensor de (3,5), ya que son secuencias de 3 palabras, cada una de ellas codificadas como un vector de 5 número flotantes. Posteriormente, la capa de aplanado simplemente será ya un tensor (15). Es importante ver, que el número de parámetros de una capa de *embedding* coincide exactamente con *(tamaño del vocabulario + 1) x (dimensión del espacio de embedding)* que justamente es el número de elementos de la *matriz de embedding* como ya se comentó antes. La capa de *flatten* no tiene parámetros puesto que su única misión es convertir la matriz en un vector. Como puede comprobarse en la figura 3.5 los parámetros de la capa de *embedding* aparecen como no entrenables, puesto que pertenecen a un ejemplo en el que la matriz de *embedding* fue suministrada. En el caso (como se verá más adelante) de añadir una capa de *embedding* que se entrena junto con el modelo los parámetros serían

entrenables. La figura 3.6 representa la salida de cada una de las capas para el comentario *hotel was nice* del ejemplo planteado.

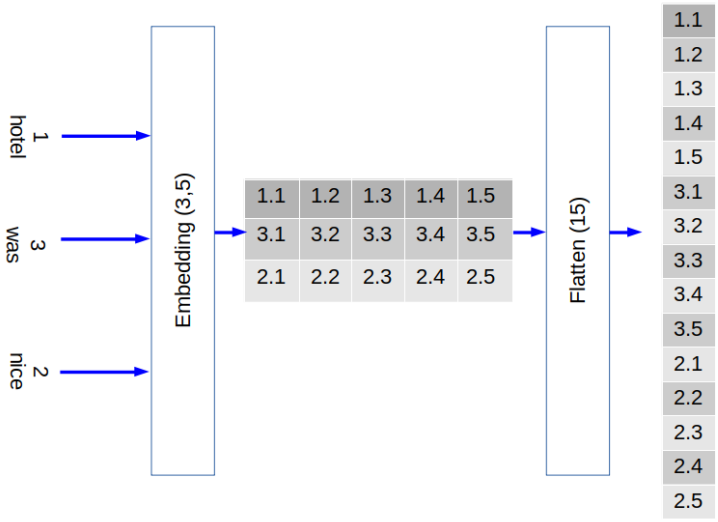


Figura 3.6. Ejemplo de embedding.

En las siguientes secciones se mostrarán diferentes implementaciones de *embedding*. Algunas de ellas aprendiendo la codificación mediante un modelo independiente, o bien entrenando la capa junto con el resto de capas de la red neuronal. También se compararán los resultados con *embeddings* ya entrenado con *corpus* externos.

3.5. Word2Vec W2V.

W2V es un modelo para aprender representaciones vectoriales de palabras (a estas representaciones se les llama "Word embedding". Normalmente se utiliza como un paso previo en modelos de redes neuronales más complejos de procesamiento de lenguaje natural, como medio para codificar las palabras en vectores de números de una dimensión fija.

La idea que hay detrás de el algoritmo W2V es entrenar una red neuronal con una capa oculta para que haga una cierta tarea, pero en vez de quedarnos con la red para que haga esa tarea, lo que queremos en realidad es quedarnos con los pesos entrenados de la red, porque esos serán justamente los "word vectors".

3.5.1. Modalidad skip-gram.

La manera de entrenar esta red de capa oculta es la siguiente: De cada palabra de cada documento del corpus (que en su iteración llamaremos palabra de entrada), elegimos sus palabras vecinas. Se considerará una palabra vecina a aquella que esté a una distancia en palabras adyacentes que hayamos definido (esta distancia es un parámetro del algoritmo que se conoce como tamaño de la ventana "window size"). La red deberá aprender decirnos cuál es la probabilidad de que esas palabras vuelvan a ser vecinas.

Supongamos el siguiente ejemplo. En cada iteración se ha puesto en negrita la palabra de entrada a la red, y subrayadas las consideradas vecinas

(que estarán a una distancia menor o igual que el tamaño de la ventana "window size"). A continuación, se muestra los pares que se generan para el entrenamiento considerando un windows size de dos.

- "**the** water in pool was cold" [the, water], [the, in]
- "the **water** in pool was cold" [water, the], [water, in], [water, pool]
- "the water **in** pool was cold" [in, the], [in, water], [in, pool], [in, was]
- "the water in **pool** was cold" [pool, water], [pool, in], [pool, was], [pool, cold]
- "the water in pool **was** cold" [was, in], [was, pool], [was, cold]
- "the water in pool was **cold**" [cold, pool], [pool, was]

Como es de esperar, la probabilidad de que la palabra "water" sea vecina de "pool" será más alta que muchas otras en el vocabulario.

Para poder realizar esto, primero es necesario poder representar los textos del corpus como vectores numéricos. Para ello, el algoritmo utiliza una variante de codificación denominada "one-hot". La codificación "one-hot" es una de las variantes de BoW en las que a cada palabra del vocabulario (el vocabulario está formado por todas las palabras del corpus) se le asigna un índice *i* diferente. La codificación de una palabra mediante esta codificación será un vector con tantas dimensiones como diferentes palabras haya en el vocabulario, de forma que tendrá un "1" en el índice correspondiente a esa palabra, y un "0" en el resto de componentes (la implementación de esta codificación es sencilla haciendo uso del modo "binary" en el método *texts_to_matrix* de la clase *Tokenizer de Keras*).

La salida de la red neuronal será un vector de números reales, con tantas componentes como palabras del vocabulario haya, de forma que por cada neurona de salida saldrá la probabilidad de que la palabra asociada a ese componente en el vocabulario sea vecina de la que se ha presentado en la entrada.

La capa oculta estará formada por un número N de neuronas lineales (no tienen una función de activación linear, sino que son la suma de las entradas por sus pesos), mientras que la capa de salida es de tipo *softmax* (Mahmood, 2018) con tantas neuronas como palabras tiene el vocabulario.

Supongamos que tenemos un corpus simple con los siguientes comentarios: ['hotel was nice', 'good food', 'hotel nice pool']. En la tabla 3.7 se muestra el vocabulario generado al partir del corpus y la codificación *one-hot* de cada una de sus palabras.

Tabla 3.7. Codificación one-hot

Indice	Palabra	one-hot
1	hotel	(1,0,0,0,0,0)
2	nice	(0,1,0,0,0,0)
3	was	(0,0,1,0,0,0)
4	good	(0,0,0,1,0,0)
5	food	(0,0,0,0,1,0)
6	pool	(0,0,0,0,0,1)

Una red como la mostrada en la figura 3.7 se entrena con todos los pares que se han generado (como se explicó en el ejemplo con el texto "the water in pool was cold"), de forma que la entrada es un vector codificado con la técnica "one-hot" para la palabra de entrada y durante el entrenamiento

la salida también es la codificación "one-hot" de la otra palabra del par.

Una vez entrenada la red, lo que se obtiene a la salida cuando se presenta en la entrada la codificación "one-hot" de una nueva palabra, es un vector con el mismo número de componentes (el tamaño del vocabulario), pero representando una distribución de la probabilidad de que la palabra correspondiente a ese índice en el vocabulario sea vecina de la de entrada, o más precisamente, en la neurona de salida *i* de la capa *softmax* tendremos la probabilidad de que si cogemos una palabra vecina de la palabra de entrada, esta coincida con la palabra *i-ésima* del vocabulario.

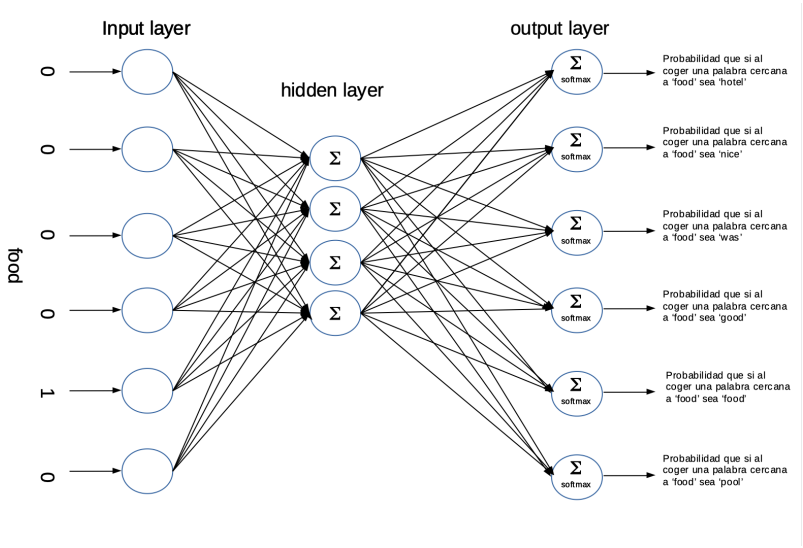


Figura 3.7. Red Neuronal W2V Skip-gram.

El número de neuronas *N* de la capa oculta representa el número de ca-

racterísticas a aprender. Cada una de las neuronas de la capa de entrada se conectará con un peso determinado a cada una de las neuronas de la capa oculta, así que, en realidad, lo que se está entrenado es una matriz de V filas (donde V es el número de palabras del vocabulario) por N columnas (donde N es el número de características a aprender, o lo que es lo mismo, la dimensión final que tendrán nuestro vectores de codificación). Esta matriz es precisamente la *matriz de embedding* que debemos aprender (McCormick, 2016).

De la capa oculta saldrán vectores de N *dimensiones*, que servirán en entrada a la capa de salida. La capa de salida es una capa *Softmax* que tiene tantas neuronas de salida como palabras hay en el vocabulario. Como se ha dicho anteriormente, la salida de la neurona i será la probabilidad de que si coges al azar una palabra vecina de la palabra codificada a la entrada, esa sea la correspondiente en el vocabulario al *índice i*.

Una representación esquemática de la red se presenta en la figura 3.8 (Weng, 2015).

La matriz de pesos representada como W es justamente la *matriz de embedding* que se inicializará con pesos aleatorios y que después de ser entrenada servirá para poder usar con otros modelos.

Como se puede entender VxN parámetros a entrenar es una gran cantidad de parámetros y mucha cantidad de cálculos (V en nuestro ejemplo de vocabulario es mayor que 20.000 y N por ejemplo 300 que es lo que usa el embedding de Google W2V). Además se necesitaría un corpus muy grande para obtener buenos resultados. Por este motivo, en un segundo trabajo, los autores de *Word2Vec* (Mikolov *et al.*, 2013) introducen dos me-

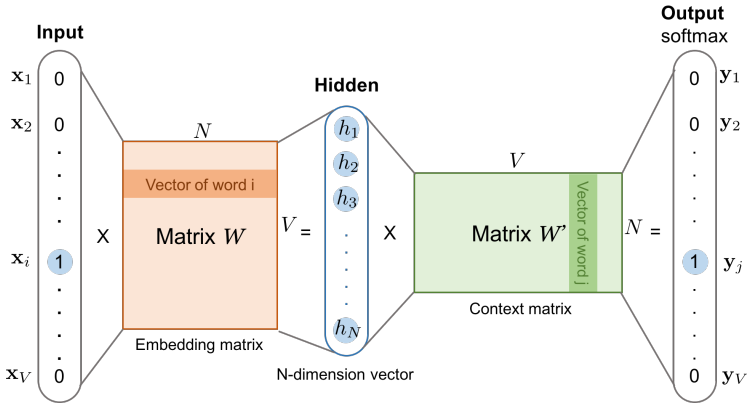


Figura 3.8. Representación esquemática W2V Skip-gram (Weng, 2015).

joras:

1. Hacer un sub-muestreo, de forma que no se elijan todas las muestras como se indicó antes.

2. Usando una técnica de optimización que llamaron "muestreo negativo" de forma que una muestra solo actualice un pequeño porcentaje de los pesos de la matriz.

Hacer un sub-muestreo consiste en disminuir de forma significativa el número de muestras que se utilizan en el entrenamiento. De forma intuitiva, muchas palabras que no aportan excesivo valor en significado (como pueden ser artículos o preposiciones) aparecerán dentro de los pares de entrenamiento. Por un lado, si una de estas palabras aparece m veces, por un lado generará $m * window\_size$ muestras, y por otro lado, también

aparecerá en las muestras de palabras que estén a una distancia menor o igual que *windows_size*. La técnica utilizada por el equipo de Tomas Mikolov para reducir el número de muestras, es eliminar algunas palabras en función de la frecuencia en la que aparecía en el corpus.

Sea una palabra w_i del corpus. Se también $z(w_i)$ al cociente de apariciones de la palabra w_i en el corpus dividida por el número total de palabras del corpus. Para practicar sub-muestreo se define un parámetro que se denomina *"sample"* y que por defecto es 0,001, de forma que la probabilidad de considerar una palabra como válida para generar las muestras $P(w_i)$ será:

$$P(w_i) = \left(\sqrt{\frac{z(w_i)}{sample}} + 1 \right) \times \frac{sample}{z(w_i)}$$

Si se representa gráficamente $P(w_i)$ como función de $z(w_i)$ se puede observar que cuando $z(w_i)$ aumenta disminuye progresivamente $P(w_i)$.

En una red neuronal convencional, cada muestra del conjunto de entrenamiento ajusta todos los pesos de la red. La técnica del muestreo negativo, por otro lado, trata de disminuir el número de pesos a modificar con las muestras de entrenamiento. La manera en la que lo hace es la siguiente:

Para entrenar la red y ajustar los pesos, se utilizan muestras con pares de palabras *(input word, output word)*, en la capa de entrada de la red neuronal se presenta la codificación *one-hot* de *input word*, mientras que en la capa de salida se presenta la codificación *one-hot* de *output word*. Recordemos que la codificación *one-hot* de una palabra corresponde con un vector de V dimensiones, donde V es el número de palabras del vocabulario, de forma que todas las componentes del vector serán 0, excepto la que co-

rresponde en índice con la palabra *output word*. La técnica lo que hace es únicamente seleccionar para ser ajustados los pesos correspondientes a la neurona de salida que representa el *output word* (que se le dice palabra positiva), y los pesos correspondientes a un pequeño grupo de neuronas que tienen un cero como salida (que se les dice palabras negativas, y por este motivo la técnica se conoce como *negative sampling*). El estudio original dice que es suficiente eligiendo los pesos de 5 a 20 neuronas correspondientes a palabras negativas para corpus pequeños, pudiéndose reducir a 2 para corpus grandes. Las palabras con mayor frecuencia de aparición en el corpus serán elegidas con mayor probabilidad como candidatas a ser palabras negativas, y por lo tanto los pesos de sus neuronas de salida correspondientes los que se ajusten con mayor probabilidad en los entrenamientos.

3.5.2. Modalidad Continuous Bag of Words (CBOW).

CBOW es otra de las modalidades del algoritmo W2V. A diferencia del anterior (skip-gram), la entrada del modelo son las palabras del contexto, y lo que trata es de predecir cuál será la palabra que es acompañada por ese contexto.

Los pasos, para construir y entrenar un modelo basado en CBOW son los siguientes:

- Construir el vocabulario del corpus (conjunto total de comentarios con los que entrenar en modelo).
- Generar las muestras de entrenamiento. En este caso, a diferencia

del anterior, el modelo trata de predecir la palabra actual en función de las palabras de contexto, por lo que para construir la muestra de entrenamiento, las entradas serán las palabras que están en su contexto (a una distancia menor o igual que *window size*) y la etiqueta será la palabra objetivo. Se generarán por tanto, pares de la forma [contexto, palabra objetivo].

Supongamos el mismo ejemplo que el detallado para la técnica *skip-gram*. En cada iteración se ha puesto en negrita la palabra objetivo, y subrayadas las consideradas vecinas (que estarán a una distancia menor o igual que el tamaño de la ventana *"window size"*). A continuación, se muestra los pares que se generan para el entrenamiento considerando un *windows size* de dos. Debe tenerse en cuenta que para las entradas que tienen menos vecinos por encontrarse cerca de los extremos debe usarse una palabra de *padding* o relleno (las redes neuronales esperan entradas de tamaño fijo):

- "**the** water in pool was cold" [[water,in, pad,pad], the]
- "the **water** in pool was cold" [[the, in, pool, pad], water]
- "the water **in** pool was cold" [[the, water, pool, was], in]
- "the water in **pool** was cold" [[water, in, was, cold], pool]
- "the water in pool **was** cold" [[in, pool, cold, pad], was]
- "the water in pool was **cold**" [[pool, was, pad, pad], cold]

- Construir el modelo para obtener la *matriz de embeding*, también mediante el uso de una red neuronal. La primera diferencia se encuen-

tra en la capa de entrada, ya que en la misma deberá presentarse la codificación del contexto de la palabra central. Esto supone que si el contexto tiene un número de palabras igual a *2 x window size*, el tamaño de la capa entrada si se utiliza la codificación one-hot será *2 x window size x vocabulary size* (siendo *vocabulary size* el número de palabras diferentes que se extrajeron del corpus).

De la misma forma como ocurría en la modalidad de *skip-gram*, el modelo debe entrenar una *matriz de embedding* de dimensión *VxN* (siendo *V* el tamaño del vocabulario, o lo que es lo mismo, la dimensión de la codificación de una palabra mediante *one-hot*, y siendo *N* el número de neuronas de la capa oculta, o lo que es lo mismo, la dimensión que tendrán los vectores que produzca el *embedding*). La figura 3.9 representa un esquema de la red neuronal, sus entradas, sus salidas y los parámetros a entrenar.

- El último paso sería entrenar el modelo con las muestras [contexto, palabra objetivo] generadas a partir del corpus.

3.5.3. Exerimentos con Embedding.

La librería Python Gensim (Radim Rehurek, n.d.) ofrece las herramientas necesarias para poder crear un embedding independiente de la red neuronal usando el algoritmo W2V, así como la posibilidad de cargar otros *embeddings* pre-entrenados.

En esta sección se compara la eficacia de diferentes *embeddings* con el mismo modelo que usamos con la técnica BoW 3.1 con el fin de poder ex-

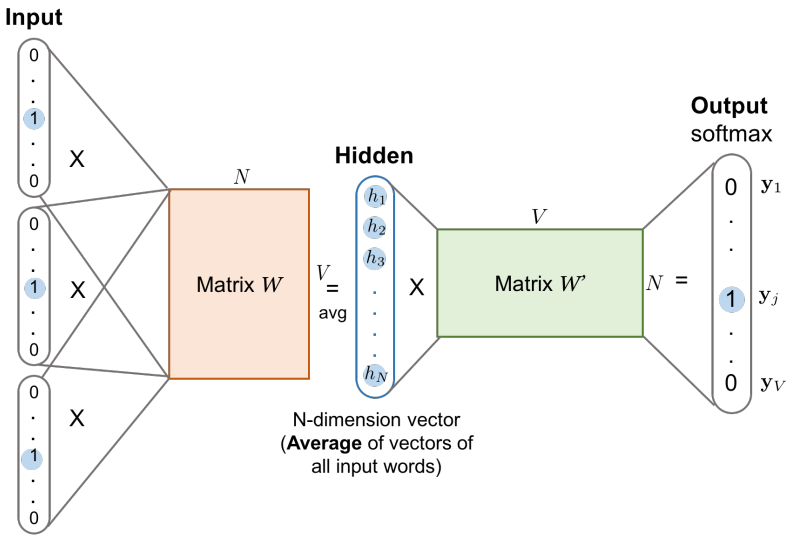


Figura 3.9. Representación esquemática W2V CBOW (Weng, 2015).

traer algunas conclusiones que nos ayuden a tomar decisiones en la elección de la técnica de codificación en futuros experimentos en el dominio que estamos estudiando. En concreto se utilizarán los siguientes *embeddings*:

1. Utilizando la librería Gensim utilizamos el método acW2V para generar una *matriz de embedding* usando como muestras el mismo fichero de entrenamiento que se utilizó en la técnica BoW. El método de la librería permite especificar diferentes parámetros para tener alternativas distintas. Algunas de ellas son:

    - *size*. Para determinar la dimensión de los vectores que genera.

Por defecto es 100, pero para los experimentos de este trabajo se utilizó 300.

- *windows*. Para determinar el tamaño de la ventana a la hora de usar el algoritmo W2V. El defecto es 5, y así se dejó en los experimentos.
- *min_count*. Este parámetro es útil para *corpus* grandes. Permite decir el número de ocurrencias mínimo que debe tener una palabra para que sea considerada en el algoritmo W2V. El defecto es 5, pero en los experimentos se utilizó 1 para que se consideraran todas las palabras (al no tratarse de un *corpus* grande).
- *workers*. Permite especificar el número de threads de cpu a utilizar para generar el embedding. Debemos tener en cuenta que si el *corpus* es muy grande puede consumir muchos recursos computacionales. El defecto es 3.
- *sg*. Permite elegir entre los algoritmos *CBOW* que es el defecto y *skip-gram*, para lo que debe ponerse a 1. En nuestros experimentos se usó el defecto.

2. Utilizando el *W2V embedding de Google*. Google ofrece un *embedding* pre-entrenado en un fichero que ocupa 3.5 GB (Google word2vec Project, n.d.). Produce vectores de dimensión 300.
3. Utilizando el *Stanford's GloVe Embedding*. Este *embedding* no utiliza el algoritmo W2V, sino el conocido como *Global Vectors for Word Representation*, y fue desarrollado por investigadores de la universidad de Stanford (Pennington *et al.*, 2014). El *embedding* de GloVe ocupa

3.29 GB y permite usar por separado *embeddings* de diferentes dimensiones (50, 100, 200 y 300). Fue entrenado usando los datos de la wikipedia. En nuestros experimentos, por compatibilidad con el resto usamos la dimensión 300.

4. Utilizando una capa de *embedding*. La librería Keras ofrece herramientas para poder añadir a nuestro modelo esta capa. Puede utilizarse de diferentes maneras: entrenándose en conjunto con el resto del modelo, o bien entrenarse por separado, aprender la codificación y salvarla a un fichero para poder ser usada en diferentes modelos. En la figura 3.5 se mostró a modo de ejemplo cómo conectar esa capa a nuestro modelo. La capa *Embedding* de Keras admite diferentes parámetros:

    - *input_dim* es el tamaño del vocabulario que se va a utilizar en el entrenamiento. Es importante reseñar en este sentido (como se hizo en la descripción la *matriz de embedding*) que el tamaño del vocabulario es el número de palabras reconocidas en el vocabulario más uno, ya que el índice reservado para el *padding* en las secuencias debe tener su correspondiente entrada en el *embedding*.

    - *output_dim* es la dimensión del vector. En nuestros experimentos, y por por compatibilidad con el resto de *embeddings* usaremos 300.

    - *input_length* que es el tamaño de las secuencias que utilizaremos en la capa de entrada previa a la capa de *embedding*.

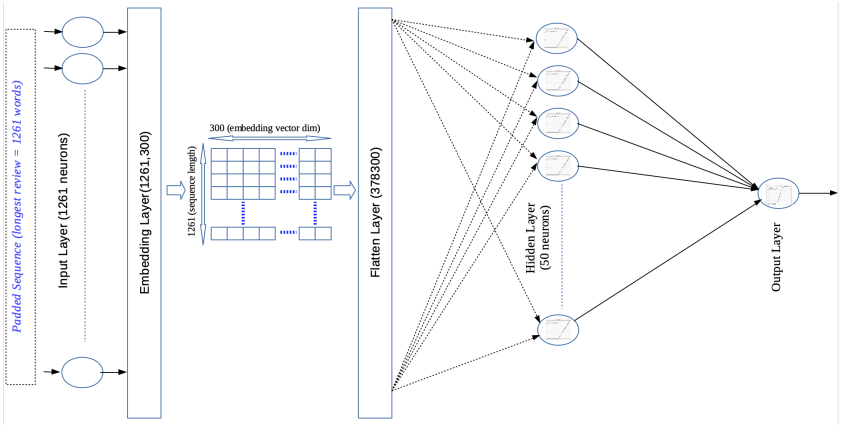


Figura 3.10. Modelo con capa embedding usado en los experimentos.

En la figura 3.10 representamos el modelo que se ha utilizado para realizar los experimentos con las diferentes codificaciones basadas en *embedding*. En concreto, se ha representado el modelo que se ha utilizado, cuando se ha elegido como tamaño de la capa de entrada la longitud del comentario más largo de la muestra de entrenamiento (1.261 palabras para este experimento). Las características completas de este modelo son las siguientes:

- La capa de entrada está formada por 1.261 neuronas. A éstas llegan las secuencias correspondientes de los comentarios. Estas secuencias han sido formadas utilizando un *Tokenizer* que creaba un vocabulario único a partir de los comentarios de entrenamiento, asignado un índice (a partir del número 1, y en orden por frecuencia de uso en el total del conjunto). Las secuencias de enteros generadas se igualaron en longitud utilizando la técnica de *padding*, que consistía

en completar con ceros hasta el tamaño de entrada. En nuestro experimentos se eligió realizar el *padding* por el final de la secuencia. También para la fase de test, en el caso de llegar un comentario más largo que la longitud establecida como máximo se decidió cortar por el final.

- La capa de *embedding* es la encargada de convertir cada una de las palabras del comentario (que ya están codificadas como enteros en la secuencia) en un vector de números reales. En nuestros experimentos hemos fijado el tamaño de este vector a 300, para poder comparar los resultados con codificaciones pre-entrenadas que utilizaban esta longitud. Como se explicó en la sección 3.4, existen dos formas de construir la capa de *embedding*. La primera de ellas es suministrarle una *matriz de embedding* ya entrenada (esto fue lo que se hizo para construir los modelos en los que se utilizan las codificaciones pre-entrenadas de Word2Vec con nuestras nuestras, Google Word2Vec y GloVe). La segunda, consiste en añadir la capa para que sea entrenada al mismo tiempo que el resto de la red. Debemos tener en cuenta que este entrenamiento consiste en crear la matriz de embedding, por lo que el número de parámetros a entrenar en esta capa y para este caso sería: $(vocabulary\_size + 1) * vector\_dim$. En nuestros experimentos *vocabylary_size* es 20.902 y como ya se ha comentado *vector_dim* 300, por lo que el número de parámetros a entrenar correspondientes a la *matriz de embedding* es de 6.270.900.

- La capa de *flatten* o aplanado tiene como única misión convertir el *tensor* de dos dimensiones que sale de la capa de *embedding* en un tensor de una única dimensión, que sirva de entrada al modelo per-

ceptrón multicapa que teníamos. En nuestro caso particular, si aplanamos un comentario convertido en una secuencia de 1261 enteros, cada uno codificado en 300 valores reales tendremos un *tensor 1D* de 378.300 reales.

- La capa *densa* de neuronas perceptrón la hemos diseñado con 50 entradas, equivalente al modelo que utilizamos para realizar la comparativa de codificación *BoW*. Los parámetros a entrenar en esta capa son: $(output\_flatten\_layer * neurons) + neurons\_bias$. En nuestro caso sería $(378{,}300 * 50 + 50)$ 18.915.101.

- La capa de salida en nuestro modelo consta de una única neurona. En este caso, cada una de las 50 neuronas de la capa *densa* se conectará con un peso a la neurona de salida, a la que tendríamos que añadir una entrada de *bias*, por lo que el número total de parámetros a entrenar para esta capa serían 51.

- A modo de ejemplo, un modelo como el planteado en la figura 3.10 con secuencias de entrada de 1.261, una capa de *embedding* que genera vectores de 300 dimensiones, para un vocabulario de 20.902 palabras diferentes, una capa *densa* de 50 neuronas, y una capa de salida con una única neurona tendrá los siguientes parámetros a entrenar: $6{,}270{,}900 + 378{,}300 * 50 + 50 + 51 = 25{,}186{,}001$

Los códigos utilizados para realizar estos experimentos se muestran en el apéndice de códigos: A.10 *Word2Vec_MLPerceptron.py* en el que se entrena el embedding aparte de la red neuronal, y A.11 *Embedding_MLPerceptron.py* en el que se entrena la capa de embedding junto con la red.

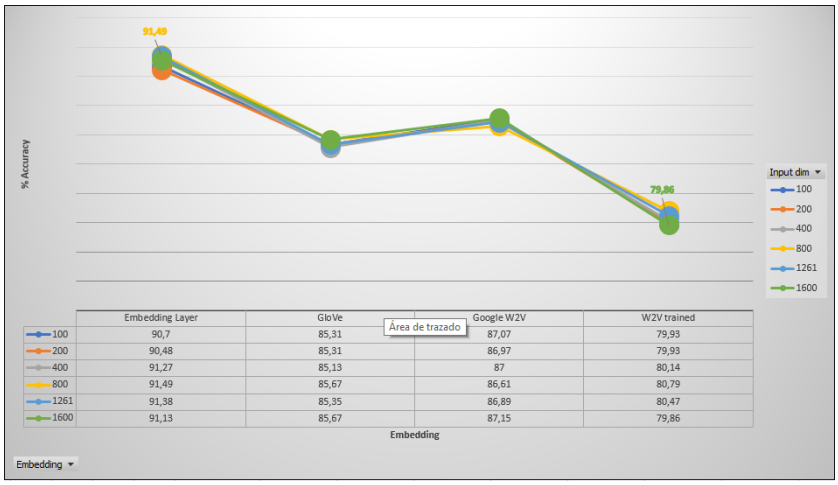
En la tabla 3.8 se muestran los resultados de los experimentos realizados con cada uno de los modos de codificación *embedding* que se han expuesto.

Tabla 3.8. Experimentos Embedding

Embedding	Input Layer dim	Train par.	No Train par.	% accuracy
Google W2V	100	1500101	6270900	87,07
Google W2V	200	3000101	6270900	86,97
Google W2V	400	6000101	6270900	87
Google W2V	800	12000101	6270900	86,61
Google W2V	1600	24000101	6270900	87,15
Google W2V	1261	18915101	6270900	86,89
GloVe	100	1500101	6270900	85,31
GloVe	200	3000101	6270900	85,31
GloVe	400	6000101	6270900	85,13
GloVe	800	12000101	6270900	85,67
GloVe	1600	24000101	6270900	85,67
GloVe	1261	18915101	6270900	85,35
W2V trained	100	1500101	6270900	79,93
W2V trained	200	3000101	6270900	79,93
W2V trained	400	6000101	6270900	80,14
W2V trained	800	12000101	6270900	80,79
W2V trained	1600	24000101	6270900	79,86
W2V trained	1261	18915101	6270900	80,47
Embedding Layer	100	7771001	0	90,7
Embedding Layer	200	9271001	0	90,48
Embedding Layer	400	12271001	0	91,27
Embedding Layer	800	18271001	0	91,49
Embedding Layer	1600	30271001	0	91,13
Embedding Layer	1261	25186001	0	91,38

En la gráfica 3.11 podemos comprobar cómo los mejores resultados en precisión se producen cuando utilizamos un modelo de codificación con una capa de embedding que se entrena con nuestro modelo, eligiendo como tamaño de la secuencia 800 palabras. Como conclusión de este experimento podríamos decir además que el *embedding* pre-entrenado de

Google funcionó mejor que el de GloVe, y que ambos, al entrenarse con una cantidad de muestras muy superior generaron un mejor embedding que cuando lo entrenamos por separado con nuestras muestras. Vuelve a ser significativa en este caso la diferencia en precisión para el mejor de los modelos y el peor, con una diferencia de 11,63 puntos.



Embedding Layer	GloVe	Google W2V	W2V trained
100	90,7	85,31	87,07	79,93
200	90,48	85,31	86,97	79,93
400	91,27	85,13	87	80,14
800	91,49	85,67	86,61	80,79
1261	91,38	85,35	86,89	80,47
1600	91,13	85,67	87,15	79,86

Figura 3.11. Comparativa precisión de embeddings.

En este capítulo se han expuesto las actividades necesarias para preparar los datos antes de que puedan ser usados por los diferentes algoritmos de aprendizaje máquina (*Machine Learning*). Las fuentes de información son heterogéneas y los datos no están organizados estructuralmente lo que complica especialmente la extracción de los mismos.

Para poder preparar un experimento es necesario centrar el dominio de la información y los objetivos de los cuales vamos a extraer los datos.

En nuestro caso particular, éstos fueron obtenidos de dos portales especializados, y se prepararon los programas necesarios para poder hacerlo. Como ya se explicó, las fuentes cambian habitualmente su tecnología y estructura, por lo que un estudio de las mismas junto con una nueva programación de las herramientas se hace necesario si se quiere hacer una nueva extracción.

Los diferentes algoritmos de *Machine Learning* requieren una representación numérica de la información. Por este motivo es necesario realizar una codificación de los datos extraídos. A lo largo de este capítulo se presentaron aquellas técnicas de codificación más habituales en esta disciplina, y se ha hecho un estudio del impacto en precisión al utilizar una u otra. Como resultado de estos experimentos se pueden extraer las siguientes conclusiones en lo referente a la influencia de la codificación con un modelo básico de red neuronal:

- En la aplicación de la técnica BoW el modo de codificación más prometedor fue *freq*, de forma que el peso de cada palabra se hace relativo a la longitud del comentario.
- No se observó mejora realizado un pre-procesamiento de las entradas, quitando palabras poco relevantes.
- En lo que refiere al tamaño de la entrada, el método BoW utiliza como tamaño el número de palabras del vocabulario. La manera de poder limitar el tamaño del vocabulario es no tener en cuenta sino las palabras más usados del mismo. Cuantas más palabras eran consideradas en el vocabulario mejores resultados se obtuvieron, pero cuando el número de palabras consideradas pasaba de 800 no se

observaba mejora, sino un ligero empeoramiento. Obviamente este parámetro dependerá el conjunto de datos de entrenamiento.

- En la técnica de codificación BoW la elección del tamaño del vocabulario y del modo de construcción tuvo una incidencia importante en los experimentos, resultado una diferencia porcentual entre el peor y el mejor modelo de 13,33 % obteniendo el mejor resultado un 91,1 % de precisión en un modelo en el que se tuvieron que entrenar 40.101 parámetros.

- En los experimentos que utilizaron la técnica de *embedding* se utilizaron secuencias de diferentes tamaños. En casi todos los experimentos los mejores resultados se obtuvieron para un tamaño de secuencias de 800 palabras. A diferencia de la técnica BoW este parámetro está relacionado con la longitud de los comentarios de los turistas. Con esta longitud estaban consideradas todas las palabras de la gran mayoría de los comentarios.

- En nuestro caso en estudio, entrenar de forma separada un *embedding* mediante el algoritmo W2V obtuvo unos resultados muy pobres. La exactitud mejoró con un *embedding* pre-entrenado de GloVe, y aún más con el de Google W2V, sin embargo el mejor de los resultados se obtuvo al entrenar el *embedding* junto con el resto del modelo obteniendo una precisión del 91,49 %.

- De la misma manera que ocurrió con BoW también es relevante la elección del método de *embedding*, ya que la diferencia porcentual de exactitud con el peor experimento fue de un 11,63 %.

- El mejor resultado en precisión, para el dominio en estudio se obtuvo utilizando la técnica de codificación de *embedding* con un 91,49 % de precisión. La diferencia en exactitud con el mejor experimento de BoW fue de 0,38 % a costa de entrenar 18.230.900 parámetros más.
- En los experimentos utilizando la técnica de *embedding* se utilizó un tamaño de vectores de codificación de 300 componentes, de manera que fueran comparables con algunas de las matrices pre-entrenadas que pudimos obtener como Google W2V. En un experimento posterior, eligiendo el modelo de *embedding* que entrenaba junto con el resto de la red, y para un tamaño de secuencia de entrada de 400 y vectores de codificación de 25 componentes obtuvimos un resultado de precisión del 91,85 %, lo cual mejoraba ligeramente la precisión y además se obtenía un reducción significativa de los parámetros a entrenar, siendo en esta ocasión 1.022.676, frente a los 18.271.001.

Una vez presentadas las técnicas de preparación de datos y codificación, se presentan en los siguientes capítulos diferentes algoritmos y modelos *Machine Learning*, que utilizarán como entrada estos datos codificados con las técnicas que resultaron más prometedoras en precisión.

Capítulo 4

Análisis de Sentimientos

Una de las aplicaciones del *procesamiento del lenguaje natural (PLN)* es el *análisis de sentimientos*. En este trabajo de investigación ponemos en estudio la clasificación de las opiniones de los turistas sobre los servicios que disfrutan, clasificándolos de forma binaria como *buenos* o *malos*, con el fin de poder construir indicadores sobre satisfacción o priorización de mejoras.

Las técnicas de *machine learning*, y concretamente las de *deep learning* requieren de datos en formato numérico para poder ser aplicadas. En el anterior capítulo 3 se presentó como realizar la extracción de los datos de portales especializados, así como la preparación de los mismos. También se hizo un estudio sobre el impacto que tenían las diferentes técnicas codificación que convertían comentarios expresados en lenguaje natural en vectores numéricos, a la hora de ser utilizados por modelos de aprendizaje.

En las diferentes secciones de este capítulo se hace un recorrido por los algoritmos y modelos de aprendizaje profundo más utilizados para la clasificación de opiniones. Se hará una descripción detallada de cada técnica

y una comparativa de los resultados aplicado al problema en estudio.

4.1. Support Vector Machine (SVM).

Las máquinas de vectores de soporte, o SVM son un conjunto de algoritmos de aprendizaje máquina supervisado ampliamente usado y con grandes resultados. El primer algoritmo SVM se le atribuye a Vladimir Vapnik en el año 1963.

Una de las aplicaciones principales de los algoritmos SVM es la clasificación. Realizar una clasificación es separar en diferentes grupos un conjunto de datos (Cortes y Vapnik, 1995). La forma de hacerlo es construir un *hiperplano* que mantenga la misma distancia (*margen*) a las muestras del conjunto de datos más cercanas a él, denominadas **vectores soporte** que dan nombre a este conjunto de algoritmos.

Supongamos que queremos clasificar en dos grupos diferentes un conjunto de datos, y que estos datos los podemos representar por un valor real (un escalar). En el caso de nuestro problema en estudio (clasificación de opiniones escritas de los turistas), equivaldría a clasificar las opiniones de los usuarios en "buenas" o "malas", pudiendo usar un único escalar para representar el comentario escrito. Si encontrásemos un valor numérico que separa perfectamente los comentarios etiquetados como "buenos", de los etiquetados como "malos", diremos que estos datos son *linealmente separables*.

Parece obvio que un escalar no es la mejor representación para el sen-

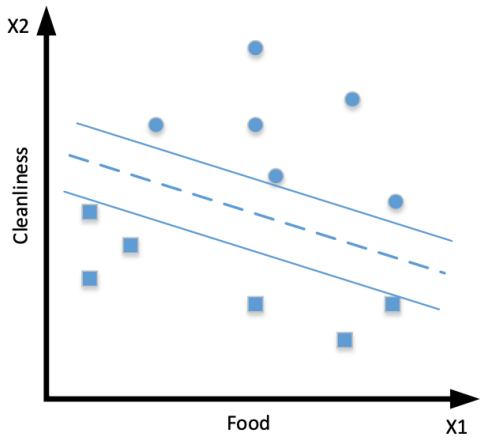


Figura 4.1. Hiperplano para datos de entrenamiento 2D.

timiento expresado en un texto escrito, ya que este tipo de datos tienen muchos aspectos o características a representar. Por este motivo, lo normal sería que si representásemos el conjunto de datos en una recta, los valores estuvieran mezclados y no sería posible separarlos. De hecho, ésta la situación más probable con problemas reales. Los datos de problemas reales suelen necesitar cientos o miles de dimensiones para poder ser representados (pese a que en la bibliografía los ejemplos o demostraciones se hagan con dos o tres dimensiones para poder ser representados gráficamente). En la figura 4.1 se representa un conjunto de datos de entrenamiento a modo de ejemplo correspondiente a textos con opiniones de turistas sobre hoteles codificados en dos dimensiones (una representativa de la comida y otra representativa de la limpieza). Los círculos representan aquellas opiniones que fueron valoradas por los usuarios como

buenas, y los cuadrados las que fueron valoradas como *malas*.

Cada vez que se añade una dimensión en la representación de nuestros datos, la figura usada para separar los mismos es diferente: un punto para una dimensión, una línea para dos dimensiones, un plano para tres dimensiones, o un *hiperplano* para más de tres dimensiones. En general, un *hiperplano* es una figura que divide el espacio multidimensional en dos mitades.

En lo que refiere a su representación mediante fórmula, en general, un hiperplano lo podemos representar como $w \cdot x + b = 0$ donde x es un vector de cualquier dimensión, y w es un vector de pesos de la misma dimensión que x, siendo $w \cdot x$ el producto escalar del vector por sus pesos.

Como se comentó con anterioridad, un conjunto de datos de dos dimensiones linealmente separables se puede separar mediante una recta. La ecuación de la recta más habitual responde a la forma $y = a \cdot x + b$ (siendo a la pendiente, y b la intersección con el eje de ordenadas). Como se puede observar, se puede representar esta misma ecuación de otra forma: $(a, -1) \cdot (x, y) + b = 0$ correspondiendo con la fórmula general de hiperplano de dos dimensiones. En alguna bibliografía y por simplificar la formulación, se suele añadir una dimensión constante (con un 1 como primera dimensión) al vector x, y pasar la b a la primera dimensión del vector de pesos:

$$\begin{aligned} \bar{x} &= (x_0, x_1, ..., x_n) = (1, x_1, ..., x_n) \\ \bar{w} &= (w_0, w_1, ..., w_n) = (b, w_1, ..., w_n) \end{aligned} \tag{4.1}$$

En cuyo caso el hiperplano quedaría definido como $\bar{w} \cdot \bar{x} = 0$ siendo $\bar{x}$ y $\bar{w}$ los definidos en (4.1). (Kowalczyk, 2017).

Encontrar el hiperplano que separa los datos en dos conjuntos (lo que se conoce como clasificación binaria) equivale a encontrar el vector de pesos $\bar{w}$. En el caso de datos linealmente separables, realizar la clasificación binaria de un conjunto de muestras consiste en aplicar una función hipótesis signo:

$$h(\bar{x}_i) = \begin{cases} +1 & \text{si } \bar{w} \cdot \bar{x}_i + b \geqslant 0 \\ -1 & \text{si } \bar{w} \cdot \bar{x}_i + b < 0 \end{cases} \tag{4.2}$$

Que usando la notación expresada en la ecuación (4.1) se denotaría como $h(\bar{x}_i) = sign(\bar{w} \cdot \bar{x}_i)$ donde $\bar{x}_i$ es cada una de las muestras del conjunto de datos de entrenamiento codificada.

El algoritmo de aprendizaje Perceptron Learning Algorithm (PLA) (Newell, 1969) (que es el que se usa como base en las redes neuronales perceptrón multicapa) fue desarrollado por Fran Rosenblatt en 1957, y es el precursor del SVM. Se basa en que un conjunto de datos de entrenamiento D, y un conjunto de funciones hipótesis, encontrar los pesos $\bar{w} = (w_0, w_1, ...w_n)$ de forma que $h(x_i) = y_i$ para cada x_i.

De forma simple, el algoritmo PLA (usando como función hipótesis el signo) consiste en encontrar el $\bar{w} = (w_0, w_1, ...w_n)$ que cumple que $sign(\bar{w} \cdot \bar{x}_i) = y_i$ para cada muestra del conjunto de entrenamiento $\bar{x}_i$ y su respectiva etiqueta y_i. De forma algorítmica:

1. Dar valores a los pesos aleatorios $\bar{w}$.

2. Recorrer todas las muestras $\bar{x}i$ y comprobar que $sign\,(\bar{w} \cdot \bar{x}_i) = y_i$

3. Si no se cumple para todas las muestras lo comprobado en el punto anterior, elegir otros pesos $\bar{w}$y volver al paso 2.

Como es lógico, la forma en la que se actualizan los pesos es lo que define la solución del algoritmo y la convergencia del mismo. El algoritmo PLA tiene muchas ventajas, como su simplicidad, pero su mayor inconveniente es que busca un hiperplano que separe los datos de entrenamiento sin tener en cuenta si es el óptimo o no. El algoritmo SVM trata de buscar el hiperplano óptimo, que se puede definir como aquél que deja a mayor distancia, el punto del conjunto de entrenamiento más cercano al hiperplano (en la literatura SVM se define como aquél que maximiza el **margen funcional** de todas las muestras de entrenamiento).

Si $\bar{w} \cdot \bar{x} = 0$ es la ecuación del hiperplano, entonces la función $f = \bar{w} \cdot \bar{x}$ dará cero para cualquier punto que esté en el hiperplano, y un número distinto de cero para aquellos que no lo estén. De hecho, cuanto más alejado esté el punto del hiperplano, dará un valor mayor de f en valor absoluto. Se cumplirá además que todos los que estén en la misma parte que ha quedado dividida por el hiperplano tendrán el mismo signo.

Con el fin de evitar problemas de escalado, o problemas con el signo de la función hipótesis se utiliza como medida lo que se conoce como **margen geométrico**. El **margen geométrico M** es una medida de la distancia a la que está el punto más cercano al hiperplano del conjunto de muestra.

Dadas las siguientes consideraciones:

- Sea m el número de muestras de entrenamiento, e i el iterador que

va desde 1 hasta m.

- El cálculo se basa en el mínimo, porque se desea saber cuál es la muestra más cercana al hiperplano.
- Se divide por la norma del vector de pesos $|||\bar{w}|||$ porque puede haber infinitos vectores que definan el mismo hiperplano (por analogía a una recta en plano, que es un hiperplano 2D, existen infinitos vectores con el mismo ángulo y diferente módulo que la definirían mediante la fórmula punto vector).
- Se multiplica por la etiqueta y correspondiente a esa muestra de entrenamiento para comprobar la predicción de la hipótesis (en el caso de ser diferentes, el signo será negativo, mientras que en el caso de coincidir predicción y etiqueta el producto será positivo).

El **margen geométrico** del conjunto de datos se formula de la siguiente manera:

$$M = \min_{i=1,\dots m} y_i \left(\frac{\bar{w}}{\|\bar{w}\|} \cdot \bar{x}_i + \frac{b}{\|\bar{w}\|} \right) \tag{4.3}$$

Así pues, el problema que se quiere resolver es encontrar aquél hiperplano (es decir los valores del vector de pesos w y b) que maximizan el margen geométrico. Por lo tanto se trata de un *problema de optimización* para el cual existen diferentes aproximaciones que producen diferentes variantes del algoritmo.

El problema de optimización planteado para la familia de algoritmos *SVM* es:

Dado un conjunto de m muestras de entrenamiento linealmente sepa-

rables, codificadas en p dimensiones y definido como:

$$D = \{(\bar{x}_i, y_i) \,|\bar{x}_i \in \mathbb{R}^p, y_i \in \{-1, 1\}\}_{i=1}^{m} \tag{4.4}$$

y un hiperplano definido por un vector de pesos $\bar{w}$ y un bias b. Encontrar:

$$\begin{aligned} &\max_{\bar{w},b} M \\ &\text{restringido a} \qquad f_i \geq F, i = 1, ..., m \end{aligned} \tag{4.5}$$

Siendo f_i el *margen funcional* correspondiente a cada muestra y F el margen funcional mayor de los posibles (el que hace mayor la distancia entre el hiperplano y la muestra más cercana).

La restricción planteada en este formulación, no tiene en cuenta la realidad existente los conjuntos de entrenamiento seleccionados de datos reales. Rara vez existen conjuntos de datos limpios en su totalidad, de forma que un pequeño porcentaje de los mismos son ruido o errores de etiquetado. Estos errores conllevarían la elección errónea del hiperplano de separación óptima, ya que la formulación expresada en el problema de optimización (4.5) no contempla que puedan haber muestras incorrectas. Esta forma de elección del *margen geométrico* es la que se conoce en la bibliografía como *margen duro* o *hard margin*.

Sobre el problema anteriormente expuesto, diferentes evoluciones de la familia de algoritmos SVM fueron añadiendo mejoras que permitían suavizar la restricción para algunos puntos. De esta manera, se solucionaba el problema de que algunas de las muestras del conjunto de entrenamiento pudieran ser erróneas (ruido o errores de etiquetado) introduciendo nue-

vos parámetros en el problema de optimización. Los parámetros para el suavizado de esta restricción dependen de la exactitud del conjunto de entrenamiento, y tienen que ser ajustados en función del experimento. El margen elegido en este caso, es conocido como *margen blando* o *soft margin*.

Cuando el conjunto de datos no es linealmente separables en una dimensión concreta es necesario realizar transformaciones en los mismos si queremos utilizar técnicas de *SMVs*. Estas transformaciones consiste en añadir nuevas dimensiones a las muestras, de forma que se consiga encontrar un hiperplano más general que permita separarlas. La transformación a aplicar depende del conjunto de datos, y encontrar la que mejor se ajusta es un problema complejo que en muchas ocasiones requieren de adquirir la experiencia necesaria mediante prueba y error.

Un **kernel** es una función de transformación que permite convertir conjuntos de datos no separables linealmente en conjuntos de datos linealmente separables, con la valiosa cualidad para algoritmos de optimización expresados en términos de productos escalares de no tener que mapear todas las muestras (lo que conllevaría un alto coste computacional). Recordar que los algoritmos SVM buscan la maximización del margen geométrico cuyo cálculo se basa en productos escalares como se formuló en (4.3). La *función kernel* permitiría realizar el cálculo del producto escalar de dos elementos que han sido transformados en una dimensión mayor sin necesidad de realizar la transformación de los mismos.

Existe una amplia lista de *kernel* que se pueden utilizar (Souza, n.d.), cuya elección depende del conjunto de datos, y cuyo ajuste requiere un intenso trabajo. Las funciones *kernel* más populares son:

- kernel Lineal.
- kernel Polinomial.
- kernel Gausiano o RBF.
- kernel String.

kernel lineal es ampliamente usado en clasificación de textos, debido a que este tipo de kernel funcionan bien con datos con muchas características (como son los textos), por su rapidez y por tener pocos parámetros para optimizar.

En lo que refiere a la implementación, la librería Scikit-learn (SciKitProject, n.d.) es una librería libre para Python que provee diferentes algoritmos de aprendizaje máquina, entre ellos SVM.

Como ya se comentó en párrafos anteriores el *margen blando (soft margin)* se introduce como una modificación al algoritmo original del SVM para permitir al clasificador cometer algunos errores (y de esta manera poder sobrellevar el problema de que haya ruido o fallos en el etiquetado de las muestras). El parámetro *C* del algoritmo se introduce en el problema de optimización para ponderar la importancia de la permisibilidad ante la existencia de muestras ruidosas (mal etiquetadas). La parametrización de *C (C >= 0)* debe realizarse de la siguiente manera:

- Valores de *C* muy altos intervienen en el problema de optimización para que se busque un hiperplano que no permita muestras mal etiquetadas, haciendo que se comporte como si se buscara un hiperplano con *margen duro*. Esto implicaría que ante la existencia de

muestras ruidosas no se lograran conjuntos de datos separables.

- Valores de C cercanos a cero debilitan las restricciones y permiten numerosas muestras mal etiquetadas, pudiendo llegar a hiperplanos que no consigan clasificar muestras.

El ajuste de C es muy dependiente del experimento (concretamente de la calidad de sus muestras). Debido a que C es escalable (puede ir desde cero hasta infinito positivo) se suele usar una variante del algoritmo *SVM* conocido como *nu-SVC* ($\nu - SVC$) en el que se usa el parámetro *nu* que varía entre $\nu \in (0, 1]$ para regular la permisibilidad de muestras de entrenamiento mal etiquetadas (un valor de uno para el parámetro *nu* equivale a definir un *margen duro*).

El parámetro *nu* define el tanto por uno máximo de errores de margen permitidos y el tanto por uno mínimo de vectores de soporte en relación con el número total de muestras de entrenamiento. Es decir, si $\nu = 0{,}05$, permitirá que el 5 % de las muestras sean consideradas como mal etiquetadas y que al menos el 5 % de las muestras de entrenamiento son vectores de soporte (muestras más cercanas al hiperplano).

Los algoritmos de clasificación SVM requieren que el conjunto de datos de entrenamiento esté codificado mediante vectores numéricos de dimensión finita. Por este motivo, y con el fin de comparar los resultados, se ha utilizado la técnica de codificación BoW explicada en la sección 3.3. Como se comprobó en los experimentos representados en las figuras 3.3 y 3.4 no existían diferencias significativas al realizar el preprocesamiento de comentarios, obteniendo incluso mejor exactitud sin limpiar comentarios. Los resultados más prometedores en dichos experimentos se obtu-

vieron al utilizar BoW con el modo *freq*, y considerando las 800 palabras más utilizadas del vocabulario. Es conveniente reseñar que no se utiliza el método de codificación que obtuvo mejores resultados en la comparativa 3.11 (embedding entrenado junto con la red neuronal con secuencias de tamaño 800), porque no se puede entrenar separada del modelo de red neuronal para realizar una codificación previa de las muestras (y el resto de técnicas de embedding presentaron resultados menos prometedores que BoW).

Así pues, los experimentos que se realizan en este trabajo utilizarán un conjunto de entrenamiento 9640 comentarios de turistas etiquetados como "Good" o "Bad", seleccionados de la extracción de datos comentada en la sección 3.2 y codificados mediante BoW con el modo *freq*, y limitando la dimensión de los vectores a 800 componentes.

Como conjunto de datos de test utilizamos una muestra independiente a la de entrenamiento de 2.785 comentarios de los cuales 1.408 estaban etiquetados como "Good" y 1.377 estaban etiquetados como "Bad".

El código utilizado para poder realizar los experimentos fue realizado en Python, utilizando la librería Scikit-learn (SciKitProject, n.d.) y puede consultarse en el apéndice de códigos bajo la referencia de A.12.

Como se puede observar, el experimento se ha realizado utilizando la variante *nu-SVC* de la familia de algoritmos SVM, probando cuatro *funciones de kernel* diferentes, y haciendo variar el parámetro *nu* en el intervalo $\nu \in (0, 1]$.

La tabla 4.1 muestra para cada tipo de *función kernel* probada, cuáles

Tabla 4.1. Resultados nu-SVC (máximos % aciertos)

kernel Function	nu-value	% Accuracy
Linear	0,42 / 0,46	92,57
Poly	0,8	77,74
RBF	0,42	91,92
Sigmoid (gamma = 0,5)	0,39 / 0,41 / 0,46 / 0,47	92,57

son los valores del parámetro *nu* que obtuvieron mejores porcentajes de acierto.

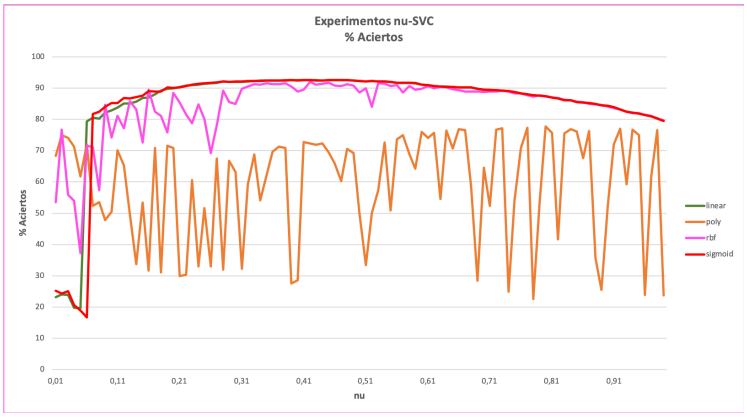


Figura 4.2. Resultados experimentos nu-SVC.

La gráfica 4.2 muestra la variación en el porcentaje de aciertos para cada valor del parámetro $\nu \in (0, 1]$ de las diferentes *funciones kernel* que se probaron.

De los resultados de la tabla 4.1 y la gráfica 4.2 se pueden obtener las siguientes conclusiones:

- Las funciones kernel *lineal* y *sigmoide* son las que obtiene el mejor porcentaje de aciertos en predicciones, con un 92,57 % para dos valores de *nu* (0,42 y 0,46) en el caso del kernel *lineal* y para cuatros valores de *nu* (0,39, 0,41, 0,46 y 0,47) en el caso del kernel *sigmoide*.

- La función kernel *lineal* y la función kernel *sigmoide* son también las más estables en su variación del parámetro *nu*. Ambas, casi superponen la curva de aciertos para todos los valores probados de *nu*. Como ya se comentó anteriormente, la función kernel *lineal* funciona bien en dominios con muchas características (como los textos) y además es rápida, por lo que ante los mismos porcentajes de acierto la convierten en una mejor candidata.

- La función kernel *RBF* también funciona de forma adecuada y llega alcanzar una precisión del 91,92 %.

- La función kernel *polinomial* no obtuvo resultados tan buenos como la *lineal* o la *RBF* obteniendo su mejor porcentaje de aciertos 77,74 % para un valor de *nu* de 0,8.

- Para poder utilizar la función kernel *sigmoide* hubo que añadir un nuevo parámetro *gamma*, debido a que su valor por defecto (como se dejó para las otras funciones kernel) no permitía separar los datos de entrenamiento con un hiperplano. El parámetro *gamma* permite suavizar la restricción en la optimización del margen, de forma que valores de gamma elevados penalizan la influencia de los vectores soporte que definen el hiperplano óptimo, y por el contrario valores pequeños de gamma no suavizan las restricciones y pueden conducir a no aprender el límite de decisión.

- Para el conjunto de datos de entrenamiento utilizado, codificado como ya se ha explicado anteriomente se observa que valores de *nu* entre 0,39 y 0,47 tiene buenos resultados con dos funciones kernel diferentes.

4.2. Convolutional Neural Networks (CNN).

Las redes neuronales convolucionales CNN tienen su origen en el campo de la visión por computador, y concretamente en el reconocimiento de imágenes, buscando imitar el funcionamiento de la corteza visual. Sus fundamentos se inician con el Neocognitron de Kunihiko Fukushima en 1980 (Fukushima, 1980) y es mejorado posteriormente en 1998 (LeCun *et al.*, 1998). En el año 2012 se consigue por parte de Dan Ciresan (Cire csan *et al.*, 2011) excelentes resultados de las mismas utilizando *GPU* (unidades de procesamiento gráfico).

Las CNN son un tipo de red neuronal cuyo principal uso y donde mejores resultados ha obtenido es en el reconocimiento de imágenes. La idea subyacente, es formar una estructura jerárquica de capas de neuronas, donde las primeras capas se encargan de aprender patrones que puedan reconocer elementos simples (como bordes), para en capas más profundas poder reconocer elementos y estructuras más complejas (lo cual favorece el reconocimiento de diferentes elementos dentro de una imagen en diferentes posiciones de la misma).

4.2.1. Operaciones de convolución y pooling.

En las CNN existen grupos de neuronas organizados en capas que están especializadas en dos operaciones: convolución y pooling. (Torres, 2018)

La *capa convolucional* es la encargada de realizar la operación de convolución. En una capa densa todas las neuronas de la capa anterior se conectan a cada una de las neuronas con unos pesos. Por el contrario, en una capa convolucional lo que se pretende es que cada neurona aprenda patrones de una pequeña porción de la información que viene de la capa anterior. Cada neurona de la capa convolucional a_j^l (*j-ésima* neurona de la capa l) comparte los mismos pesos y *bias*, de forma que realiza una operación de filtro convolución con una porción de la salida de la capa anterior (Martín *et al.*, 2018).

$$a_j^{lk} = \sigma^{lk}\left(\left(W^{lk} * a^{l-1}\right)_j + b^{lk}\right) \tag{4.6}$$

La ecuación (4.6) muestra la operación de convolución done $\left(W^{lk} * a^{l-1}\right)_j$ es el j-ésimo resultado de la convolución definido por el filtro W^{lk} de pesos (compartido por las neuronas de la capa para ese filtro) con la salida de las neuronas de la capa anterior a^{l-1}. En este caso concreto σ^{lk} es la función de activación *ReLU* de las neuronas de la capa convolucional, y $k \in [0..K]$ hace referencia al *k-esimo* filtro. En los ejemplos se ha utilizado esta función activación *ReLU* para evitar el problema de desvanecimiento del gradiente (Glorot *et al.*, 2011).

Para poder explicar el funcionamiento de la capa se utilizará un ejemplo

simplificado con imágenes y posteriormente se desarrollará el modelo con los textos utilizados en los experimentos.

Supongamos que tenemos una imagen en blanco y negro (si fuera color habría que representar cada una de las dimensiones pertenecientes a los canales *RGB*) de 10x10 píxeles como entrada. Esto supondría que nuestra capa de entrada tendría 100 neuronas. Supongamos además que queremos hacer una operación de convolución utilizando una ventana de 3x3 píxeles que se desplazará pixel a pixel en horizontal y en vertical (tanto el tamaño de de la ventana como el desplazamiento *stride* es configurable al problema). Cada neurona de la capa convolucional se conectará entonces con 9 neuronas de la capa de entrada, mediante unos pesos y una entrada de *bias*. Teniendo en cuenta el desplazamiento de la ventana, y su tamaño, el número de neuronas necesario en la capa convolucional sería de 64 (una ventana de 3x3 solamente podrá desplazarse 7 veces más en la horizontal por lo que se necesitarán por fila la actual más 7 desplazamientos, es decir, 8 neuronas, y también podrá desplazarse únicamente 7 veces en vertical por lo que de forma intuitiva se necesitaría una matriz de 8 x 8 = 64 neuronas). Es conveniente comentar que existen diferentes implementaciones de convolución, en la que se puede añadir un borde con ceros *padding* con el fin de permitir más movimientos de la *ventana* y variar la dimensión de la salida. La figura 4.3 muestra de forma visual la distribución de neuronas por capas en un ejemplo para imágenes de 10 x 10 píxeles. El valor de la salida de cada neurona de la capa convolucional corresponde a una operación conocida como *filtro* entre el valor de las entradas, los pesos y el *bias*. Una de las características principales de la capa convolucional es que utiliza el mismo filtro (es decir los mismos pesos y bias) para todas las neuronas de la capa, reduciendo así significativamente el número de

parámetros a entrenar. El *filtro* que viene definido por el conjunto de pesos y bias será lo que permita detectar una característica concreta de la imagen, y por este motivo es por el que se suelen utilizar varios filtros en una capa convolucional. La figura 4.3 muestra una imagen representativa de convolución de dos dimensiones de una imagen, usando un tamaño de *kernel o ventana* de 3 y con un solo filtro (es habitual aplicar diferentes filtros para aprender diferentes características).

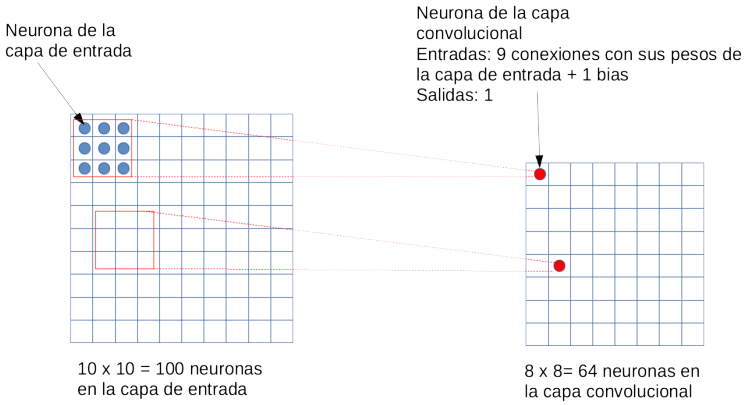


Figura 4.3. Ejemplo convolución imágenes 10x10.

En el siguiente nivel de profundidad a la capa de convolución se le suele añadir una capa de *pooling*. La operación de *pooling* tiene como objeto realizar una compresión de la información obtenida en la capa de convolución. En esta ocasión, se define nuevamente una porción de la informa-

ción (*ventana*) a comprimir y mediante una operación (las más usadas con la de máximo *max-pooling* o media *average-pooling*) realizada en la sección, se calcula el valor de la salida. La ventana de la operación de *pooling* no es deslizante, de forma que los elementos de entrada a la operación únicamente son utilizados una vez. La figura 4.4 muestra de forma gráfica en qué consistiría la operación de pooling en dos dimensiones usando un tamaño de *ventana* de 2.

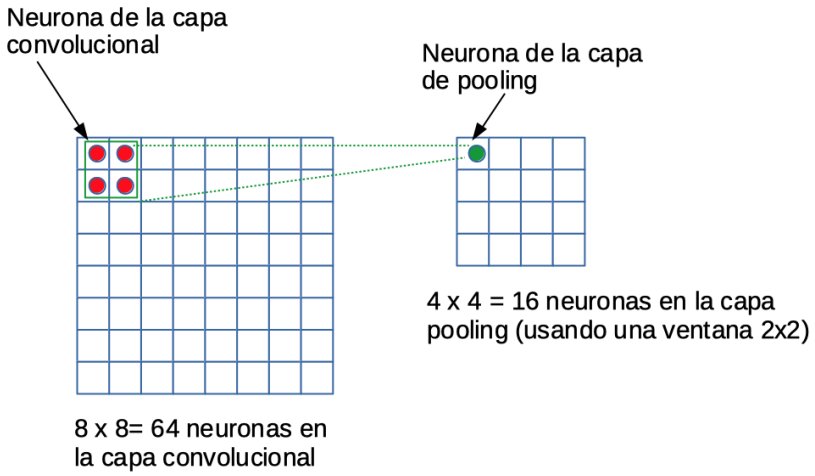


Figura 4.4. Ejemplo pooling ventana 2x2.

La operación *max-pooling* elige el valor máximo de las entradas, mientras que *average-poooling* calcula la media de las mismas. Como ya se ha dicho, la operación de *pooling* está destinada a realizar una compresión de la capa anterior. Como es lógico, si la capa convolucional anterior aplica-

ba diferentes filtros, entonces es necesario realizar esta misma operación para cada uno de los filtros.

4.2.2. Convolución con textos codificados.

En el capítulo 3 se detalló cómo las redes neuronales requerían valores numéricos a su entrada, y cuáles eran las técnicas de codificación habituales en problemas de procesamiento de lenguaje natural.

La sección 3.2 describía la técnica BoW, y la sección 3.4 y siguientes diferentes implementaciones de la técnica *Word Embedding*. Se realizó una comparativa del impacto que tenía la técnica de codificación en un modelo de clasificación de textos y se comprobó que los mejores resultados se obtenían codificando con *embedding* mediante una capa adjunta a la entrada del modelo que se entrenaba junto con el mismo.

Para describir de forma adecuada la arquitectura y funcionamiento de una *CNN* para clasificación de textos se utilizará en los ejemplos la codificación *W2V embedding de Google* que ofrece un *embedding* pre-entrenado que produce vectores de dimensión 300. El motivo de esta elección de *embedding* para explicar el modelo es el poder aislar el entrenamiento de la codificación del resto de la red (aunque en posteriores experimentos se entrenará la capa de embedding junto con el resto del modelo, ya que produjo los mejores resultados como se mostró en el capítulo 3. También se ha descartado la técnica de codificación BoW en esta ocasión porque dicha técnica tiene el inconveniente de no representar el orden de las palabras, y éste es significativo en la extracción de características que pretende el modelo convolucional.

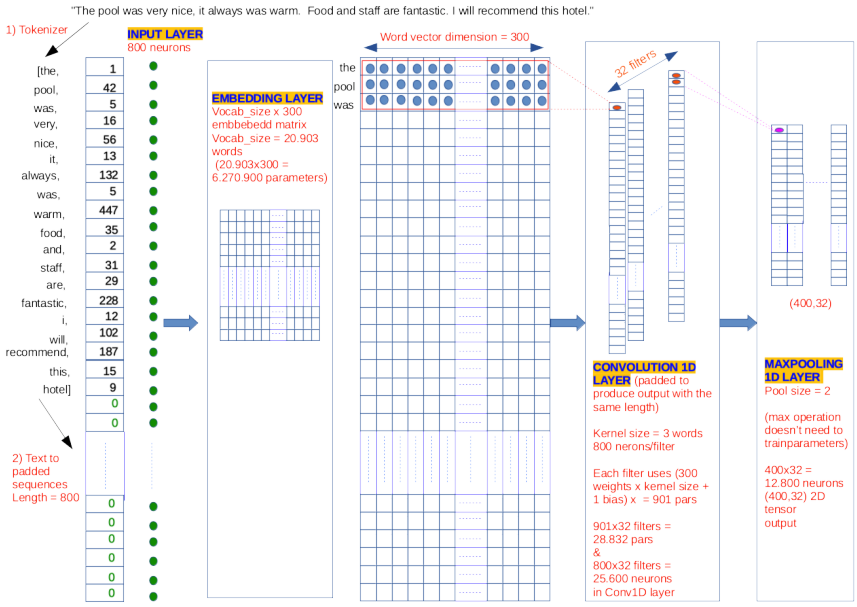


Figura 4.5. Modelo CNN para clasificación de textos. Parte 1.

las figura 4.5 y la figura 4.6 representan las dos partes de un mismo modelo de red *CNN* para la clasificación de comentarios de turistas expresados mediantes textos.

El modelo ha sido implementado mediante un programa Python que se encuentra incluído en el apéndice de códigos A.13. Se ha utilizado la librería *Keras* (Keras API, n.d.) y utilizando convoluciones de una dimensión sobre la codificación de palabras adyacentes en el comentario, como se propone en (Kim, 2014).

La descripción de las capas del modelo representados en la figura 4.5 y en la figura 4.6 es el siguiente:

- Como ya se comentó, para este ejemplo se utilizó la codificación *W2V embedding de Google* expuesta en la sección 3.5.3. Para poder utilizar esta codificación es necesaria la realización previa de una serie de pasos:

  1. Utilizando la clase *Tokenizer* de la librería *keras* generamos un diccionario, en el que a cada palabra se le asigna un número entero (las palabras más frecuentes en los comentarios corresponderán con número menores, siendo la más frecuente el número 1, ya que el 0 se reserva para relleno *padding*).

  2. Cada comentario (en el ejemplo se toma como muestra el comentario *"The pool was very nice, it always was warm. Food and staff are fantastic. I will recommend this hotel."*) se separa en *tokens* realizando algunas transformaciones (eliminando signos de puntuación o convirtiendo a minúsculas para reconocer la misma palabra capitalizada).

  3. Se debe elegir una longitud máxima para los comentarios, ya que las redes neuronales exigen un tamaño fijo en la entrada. Es habitual elegir la longitud del comentario más largo, pero en esta ocasión, y dados los resultados obtenidos con diferentes longitudes en el capítulo 3 para este tipo de codificación se optó por elegir como longitud máxima de comentario 800.

  4. Los comentarios transformados en listas de tokens se codifican

en *secuencias*. Una *secuencia* en este ejemplo es un vector de enteros de 800 componentes (longitud máxima elegida) en el que cada componente corresponde con la codificación como entero asignada por el *Tokenizer* de las palabras del comentario, truncando el mismo a la longitud elegida si es mayor, o rellenado con ceros (*padding*) si es menor. En la figura se puede observar la secuencia codificada para el comentario, rellenando con *padding* la misma hasta llegar a las 800 componentes. Como puede observarse, si una misma palabra aparece varias veces en el comentario, la codificación de la misma debe ser igual (observar la codificación para la palabra *"was"*.

5. Finalmente, estas secuencias son las que servirán como entrada a la capa de *Embedding*.

- La capa de *Embedding* de la libreía Keras nos permite suministrarle la *matriz de embedding* que ha sido previamente generada, en el que cada palabra del vocabulario extraído del *Tokenizer* se codificó con el embedding pre-entrenado del *W2V Google*. En esta ocasión, cada palabra del vocabulario se transformará en un vector de 300 componentes de números flotantes, en el que palabras con un significado similar serán cercanas en el espacio del vector. Los parámetros de esta capa (que no son necesarios entrenar porque se suministran a la misma) son el número de palabras total del vocabulario (20.903 en nuestro caso de estudio), multiplicado por el número de componentes del vector 300, que es justamente la dimensión de la *matriz de embedding*.

- La salida de la *capa de Embedding*, y entrada a la capa convolucional,

es justamente la representación del comentario del turista mediante una matriz de números flotantes de 800x300, en el que cada fila de esta matriz corresponde con cada palabra del comentario codificado.

- La operación de convolución realizada por la capa correspondiente de la figura 4.5 es una convolución en una dimensión (*conv1D*), ya que un comentario (texto) debe interpretarse como una secuencia de palabras. Lo que se pretende en esta operación de convolución es extraer características analizando grupos reducidos de palabras adyacente del comentario. En el modelo, se ha utilizado un tamaño de *kernel = 3*, lo que significa que la operación de convolución se realizará con grupos de tres palabras codificadas, y que la ventana de convolución se desplazará palabra a palabra.

- Es conveniente recordar que la operación de convolución se realiza entre todas las palabras de la ventana. En nuestro caso concreto (*kernel = 3*) y palabras codificadas con 300 componentes, la operación de convolución involucrará 900 parámetros. Como ya se ha comentado anteriormente, la operación de convolución de un *filtro* se realiza con los mismos parámetros para todos los deslizamientos de la ventana, por lo que los parámetros, y la entrada de *bias* siguen siendo los mismos. La aplicación de un *filtro* de convolución en la secuencia codificada generará una columna de información que corresponde con una característica extraída de la ventana correspondiente. Como puede observarse en el modelo, al comentario codificado se le aplicaron 32 filtros diferentes, lo que se necesitarán en la capa convolucional 25.600 neuronas (correspondientes a las 800 filas x 32 filtros

diferentes), que utilizan un total de 28.832 parámetros (correspondientes a las 300 pesos aplicados a las componentes de cada palabra multiplicados por el tamaño de la ventana 3 más una entrada de bías, y todo multiplicado nuevamente por los 32 filtros aplicados). En el código correspondiente al apéndice A.13 puede observarse cómo en la capa convolucional *conv1D* se aplicó el parámetro *padding='same'* que añade una fila de relleno al principio y al final del comentrio codificado. Éste es el motivo que la salida de la capa convolucional tenga 800 filas (el mismo tamaño que la secuencia codificada), ya que la ventana en cada filtro podrá desplazarse por todas las filas. La salida de la capa convolucional del modelo será pues un tensor (800,32).

- A continuación de la operación de convolución, la capa *MaxPooling* realiza una compresión de la salida de la capa convolucional. En el modelo representado se ha elegido una tamaño de ventana de pooling *pool_size = 2*. Esto quiere decir, que se elige el máximo de cada dos filas. Recordar que la ventana de pooling no se desliza por cada fila, de forma que una fila sólo es procesada en una operación de pooling. Como salida de la capa de pooling tendremos pues un tensor (400,2). La operación *MaxPooling* no requiere de parámetros a entrenar.

- Ya en la figura 4.6 se puede ver cómo la capa de aplanado *flatten* convierte el tensor de salida de la capa *Maxpooling* en una única dimensión con 12.800 neuronas (400 x 32).

- La salida de esta capa ya es la entrada a una capa densa de 50 neuronas destinada a la clasificación. Al tratarse de una capa densa, todas

las entradas se conectan a cada una de las neuronas que tendrán además una entrada *bias*. El número de parámetros a entrenar en esta capa será 640.050 ((12.800 + 1 bias) x 50 neuronas).

- Finalmente, al tratarse de una clasificación binaria (los comentarios se clasifican como *"Good"* o *"Bad"*) se utiliza una capa de salida con una única neurona y 51 parámetros a entrenar (las conexiones de la capa anterior más la entrada de *bias*)

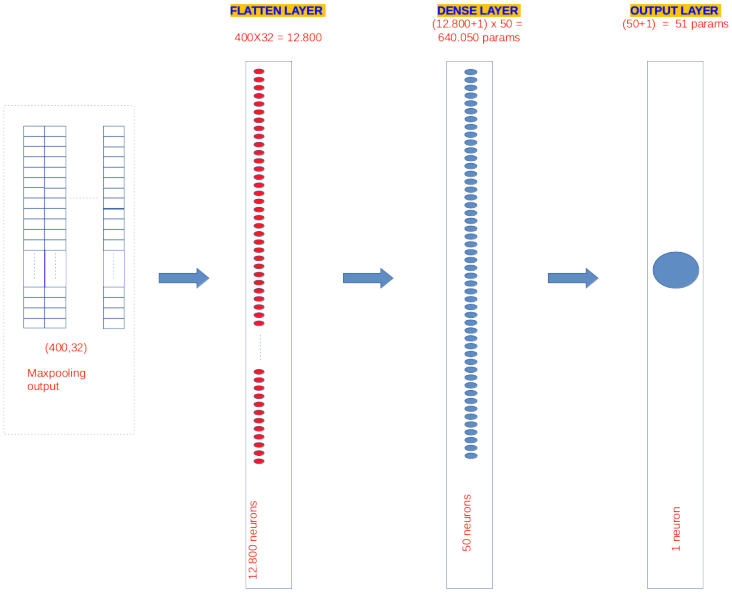


Figura 4.6. Modelo CNN para clasificación de textos. Parte 2.

En la figura 4.7 podemos ver la salida del método *summary* de la clase *model* de Keras para el ejemplo presentado.

```
Tensor x_train.ndim =  2  x_train.shape =  (9640, 800)  x_train.dtype =  int32
Tensor x_test.ndim =  2  x_test.shape =  (80, 800)  x_test.dtype =  int32
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, 800, 300)          6270900
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 800, 32)           28832
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 400, 32)           0
_________________________________________________________________
flatten_3 (Flatten)          (None, 12800)             0
_________________________________________________________________
dense_1 (Dense)              (None, 50)                640050
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 51
=================================================================
Total params: 6,939,833
Trainable params: 668,933
Non-trainable params: 6,270,900
```

Figura 4.7. Resumen del modelo generado por Keras.

El número de filtros a aplicar, así como el tamaño de la ventana de convolución son los parámetros más habituales a configurar en una *CNN*. El *padding* para hacer la convolución en todas las palabras lo hemos dejado constante, y el paso de la ventana *stride* también lo hemos dejado constante y a una unidad (en análisis de texto parece tener más sentido elegir palabras adyacentes para la operación de convolución).

La tabla 4.2 muestra las diferentes pruebas realizadas con la red del modelo presentado en las figura 4.7, para un conjunto de test de 2.785 muestras (1.408 comentarios etiquetados como ”Good“ y 1.377 comentarios etiquetados como ”Bad“), después de haber sido entrenadas con 9.640 muestras balanceadas.

En la figura 4.8 podemos ver de forma gráfica la evolución de la precisión en función del número de filtros y el tamaño del kernel.

Tabla 4.2. Experimentos para el modelo de la figura 4.7.

N.Filters	Kern.Size	"Good"hits	"Bad"hits	"Good"fails	"Bad"fails	% Accu
16	2	1.236	1.279	98	172	90,31
16	3	1.227	1.290	87	181	90,38
16	4	1.206	1.283	94	202	89,37
16	5	1.231	1.286	91	177	90,38
32	2	1.241	1.282	95	167	90,59
32	3	1.226	1.279	98	182	89,95
32	4	1.220	1.280	97	188	89,77
32	5	1.225	1.285	92	183	90,13
64	2	1.223	1.303	74	185	90,7
64	3	1.251	1.286	91	157	91,1
64	4	1.235	1.289	88	173	90,63
64	5	1.245	1.285	92	163	90,84
128	2	1.234	1.292	85	174	90,7
128	3	1.243	1.292	85	165	91,02
128	4	1.236	1.296	81	172	90,92
128	5	1.221	1.298	79	187	90,45

De la tabla 4.2 podemos sacar las siguientes conclusiones:

- El mejor resultado se obtiene aplicando en la capa convolucional 64 filtros diferentes y un tamaño de kernel 3. Se alcanza un **91,1 %** de precisión.
- Usando este tipo de codificación (*Embedding preentrenado de Google*) en la tabla 3.8 en las que no se utilizaba una capa convolucional previa a la capa densa se alcanzaba un 86,61 %, lo que supone que añadir la pareja de capas convolucional - maxpool **ha añadido 4.48 puntos de precisión**.
- Para tres de los diferentes números de filtro, el tamaño de kernel tres es el que mejor se comporta.

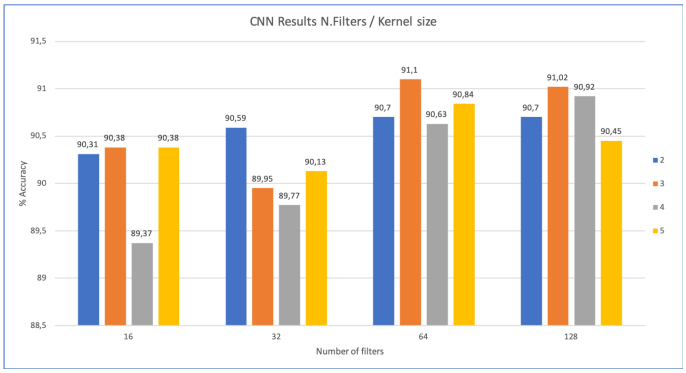


Figura 4.8. Experimentos CNN para el modelo de la figura 4.7

- En general, aumentar el número de filtros benefició, pero a partir de 64 filtros empeoró.
- En la tabla 4.2 se puede observar también como realiza más aciertos de predicción en muestras etiquetadas como "Bad". En general, en clasificación de textos este resultado es habitual por la carga de significado de los términos negativos.

Para completar el estudio, se modificó el modelo presentado en la figura 4.7, añadiendo una nueva pareja de capas convolucional y maxpool, dando como resultado el presentado en la figura 4.9, en el que se utilizan dos parejas de capas convolucionales y maxpooling, utilizando en ambas 16 filtros diferentes en la convolución, con un tamaño de kernel 2, y un tamaño de ventana de pooling de también 2.

El código definitivo para poder hacer la comparativa entre ambos modelos, haciendo variar el número de filtros y el tamaño del kernel de con-

volución es el que se adjunta en el apéndice de códigos A.14.

```
Tensor x_train.ndim =  2  x_train.shape =  (9640, 800)  x_train.dtype =  int32
Tensor x_test.ndim =  2  x_test.shape =  (80, 800)  x_test.dtype =  int32
Nº filtros aplicados:  16  Tamano kernel:  2
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 800, 300)          6270900
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 800, 16)           9616
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 400, 16)           0
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 400, 16)           528
_________________________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 200, 16)           0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3200)              0
_________________________________________________________________
dense_1 (Dense)              (None, 50)                160050
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 51
=================================================================
Total params: 6,441,145
Trainable params: 170,245
Non-trainable params: 6,270,900
```

Figura 4.9. Resumen del modelo de dos Convoluciones - Maxpool generado por Keras.

Los resultados obtenidos se presentan en la tabla 4.3 y gráficamente en la figura 4.10. De ellos podemos obtener las siguientes conclusiones:

- El mejor resultado en precisión se obtiene utilizando 64 filtros y tamaño de kernel 2 en ambas capas convolucionales, con un **92,14 %** de precisión. Este resultado supera el obtenido con una única capa convolucional.
- En general, los resultados en precisión, para todas las combinaciones de filtros y tamaños de kernel mejoraron al añadir una nueva pareja de capas convolucional - maxpooling. Este resultado se puede observar en la figura 4.11.
- Nuevamente se puede observar cómo las predicciones de comenta-

Tabla 4.3. Experimentos para el modelo de la figura 4.9.

N. Filter	Kern.Size	"Good"hits	"Bad"hits	"Good"fails	"Bad"fails	% Accu
16	2	1.255	1.295	82	153	91,56
16	3	1.229	1.283	94	179	90,2
16	4	1.245	1.259	118	163	89,91
16	5	1.241	1.301	76	167	91,27
32	2	1.270	1.289	88	138	91,89
32	3	1.243	1.293	84	165	91,06
32	4	1.229	1.296	81	179	90,66
32	5	1.224	1.294	83	184	90,41
64	2	1.257	1.309	68	151	92,14
64	3	1.272	1.286	91	136	91,85
64	4	1.240	1.286	91	168	90,7
64	5	1.259	1.292	85	149	91,6
128	2	1.243	1.302	75	165	91,38
128	3	1.275	1.283	94	133	91,85
128	4	1.257	1.292	85	151	91,53
128	5	1.247	1.298	79	161	91,38

rios de turistas etiquetadas como "Bad" fueron más precisas.

Por último, y dados los resultados en el capítulo anterior en el que se realizó una comparativa de los diferentes métodos de *embedding* reflejados en la tabla 3.8, realizamos un último experimento, utilizando una capa de *embedding* que se entrenara junto con el resto de la red del modelo de la figura 4.9. Para ello, se eliminó el entrenamiento del *Google Embedding* y se volvieron a ejecutar los entrenamientos y predicciones. Los resultados son los mostrados en la tabla 4.4.

Como puede observarse en la gráfica 4.11, al añadir al modelo 4.9 una capa de embedding que se entrenara junto a él no se mejoraron los resultados obtenidos al utilizar un embedding pre-entrenado. Es importante recordar que cuando se utilizó una capa de embedding a entrenar jun-

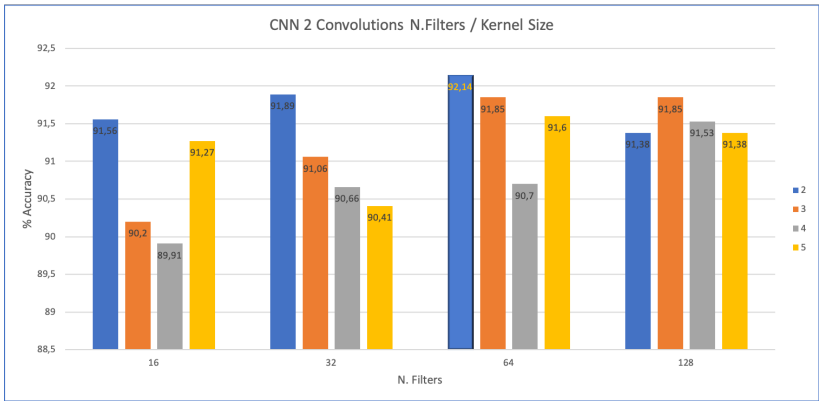


Figura 4.10. Comparativa de precisión para el modelo de la figura 4.9

to con un modelo formado únicamente por una red densa los resultados mejoraron. La conclusión que podemos obtener, es que la convolución es capaz de extraer mejor las características cuando se utiliza un embedding pre-entrenado con un conjunto de datos más grandes, como puede ser el de Google. Por este motivo será interesante probar a entrenar la capa de embedding junto con el modelo de red convolucional cuando trabajemos con juntos de entrenamiento más grande.

Como conclusión final para este modelo de redes *CNN* decir que los mejores resultados de precisión que se obtuvieron fueron del 92,14 % utilizando dos pares de capas convolucionales - maxpool con 64 filtros y usando un tamaño de *kernel* de 2. Este resultado es muy aproximado al obtenido con el algoritmo *nu-SVC* pero ligeramente peor, siempre teniendo en cuenta el conjunto de datos de entrenamiento no es excesivamente grande.

Tabla 4.4. Experimentos para el modelo con dos capas convolucionales y capa de embedding entrenando.

N. Filter	Kern.Size	"Good"hits	"Bad"hits	"Good"fails	"Bad"fails	% Accu
16	2	1.245	1.272	105	163	90,38
16	3	1.261	1.274	103	147	91,02
16	4	1.229	1.294	83	179	90,59
16	5	1.244	1.291	86	164	91,02
32	2	1.234	1.280	97	174	90,27
32	3	1223	1.292	85	185	90,31
32	4	1.228	1.286	91	180	90,27
32	5	1.225	1.296	81	183	90,52
64	2	1.243	1.269	108	165	90,2
64	3	1.238	1.268	109	170	89,98
64	4	1.242	1.276	101	166	90,41
64	5	1.214	1.293	84	194	90,02
128	2	1.230	1.282	95	178	90,2
128	3	1.256	1.279	98	152	91,02
128	4	1.204	1.296	81	204	89,77
128	5	1.231	1.290	87	177	90,52

4.3. Long Short Term Memory networks (LSTM).

4.3.1. Recurrent Neural Network (RNN).

Las redes neuronales recurrentes nos permiten, a diferencia de las redes neuronales convencionales usar conocimiento adquirido de instantes anteriores. Son redes con bucles que permiten que la información persista (Colah, 2015).

La figura 4.12 muestra una representación de este bucle, en el que se representa A como parte de la red neuronal x_t como la entrada en un instante t, y h_t como la salida de la red en ese mismo instante. La manera en la que se implementa el bucle es mediante la repetición de múltiples

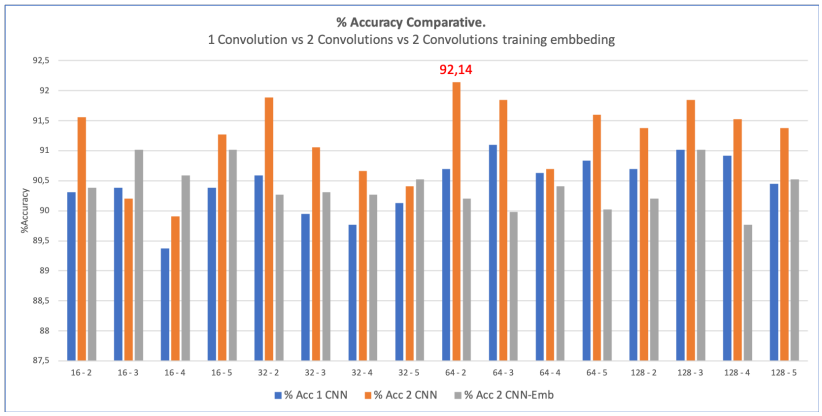


Figura 4.11. Comparativa de precisión para los modelos de las figuras 4.7 - 4.9- Añadiendo una capa de Embedding a entrenar

copias de la misma red, en la que cada una pasa información a su sucesor (desenredando el bucle), como se muestra en la figura 4.13.

Las RNN tienen excelentes resultados en múltiples campos del aprendizaje máquina, como puede ser el reconocimiento del lenguaje, pero presentan algún problema como las dependencias a largo plazo. En determinadas ocasiones, para poder realizar predicciones útiles, puede ser necesaria la información de instantes de tiempos muy anteriores al actual. Teniendo en cuenta cómo se realiza la estructura desenredada presentada en la figura 4.13 para estas ocasiones, la distancia entre las neuronas o redes relacionadas sería grande, y las Recurrent Neural Networks (RNNs) tendrían grandes dificultades en aprender a conectar la información (Hochreiter *et al.*, 2001).

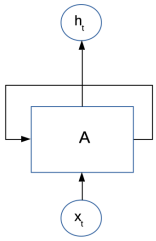


Figura 4.12. Representación bucle Redes Neuronales Recurrentes (Colah, 2015).

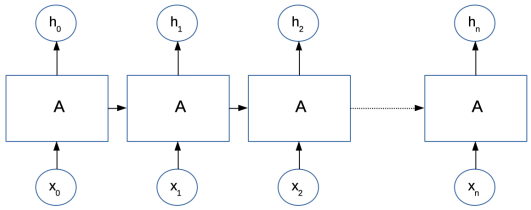


Figura 4.13. Representación desenredada Redes Neuronales Recurrentes (Colah, 2015).

Para resolver este problema surgen las LSTM que se explican en las siguiente sección.

4.3.2. Long Short Term Memory Networks LSTM.

Las redes LSTM son un tipo especial de RNN capaces de aprender dependencias a largo plazo. Fueron introducidas por Hochreiter y Schmidhuber en el año 1.997 (Hochreiter y Schmidhuber, 1997).

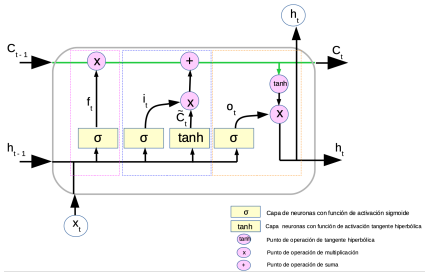


Figura 4.14. Detalle del interior de una celda LSTM.

Las redes LSTM tienen también una estructura de conexión en cadena. Cada una de las celdas representadas en la figura 4.14 contiene cuatro capas diferentes de neuronas, y la explicación de cada uno de los elementos es la siguiente:

- x_t es el vector de entrada en el instante t, y h_t es la salida de la celda en ese mismo instante y C_t representa el estado de la celda también en ese mismo instante. h_{t-1} es la salida de la celda que representa el instante anterior. Cada una de las lineas del diagrama transporta un vector. Los rectángulos amarillos corresponden con una capa de una red neuronal y los círculos rosas puntos de operación.

- La conexión superior que se ha representado en color *verde* representa el estado de memoria que contiene la celda. C_{t-1} es el estado proveniente de la celda que representa el estado anterior, mientras que C_t será el estado de la celda actual. En puntos posteriores se explica cómo se actualiza este estado.

- La primera operación en la figura 4.14 que se ha resaltado con un

rectángulo punteado en color rosa servirá para decidir que información continuará en el estado. La combinación de una capa de neuronas con función de activación sigmoide, y una operación producto es lo que se conoce como una puerta (**gate** en su nomenclatura habitual). Hay que tener en cuenta que una función sigmoide devuelve un número entre cero y uno, de forma que llevada a sus extremos, un cero significará que nada pasará y un uno significará que pasará toda la información. Para esta *gate* concreta, y a la que hemos denominado f_t (*forget gate layer*) se le asigna la función de aprender qué información debe recordarse o olvidarse del estado. Cuando f_t sea uno permitirá pasar la información de C_{t-1}. La ecuación del *forget gate layer* es $f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$. Como puede comprobarse se trata de una capa densa estándar con tantas neuronas como tenga la concatenación de la salida de la celda anterior con el vector de entrada, y que deberá de entrenar tantos parámetros como neuronas más la entrada de *bias*. Los pesos a entrenar en esta *forget gate layer* se le ha denominado W_f.

- La segunda operación de la celda *LSTM* y que en la figura 4.14 se ha resaltado con un rectángulo punteado en color azul servirá para con qué información se actualizará el estado de la celda. La salida de la capa sigmoide que se ha denominado i_t con la operación de multiplicación vuelve a ser una puerta. A ésta se le conoce como *input gate layer*, y decidirá qué valores se actualizarán. La ecuación del *input gate layer* es $i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$. Por otro lado, la salida de la capa tangente hiperbólica *tanh* crea un vector de nuevos candidatos a estado para ser añadido mediante la operación suma (+) condicionado por el *input gate layer* i_t. La ecuación de este nuevo can-

didato es $\widetilde{C}_t = tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right)$. De esta forma, el estado de la celda C_t se actualiza según la fórmula $C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t$.

- La última operación de la celda *LSTM* y que en la figura 4.14 se ha resaltado con un rectángulo punteado en color naranja servirá para decidir que irá en la salida h_t. Se modifica el estado calculado C_t mediante una operación de tangente hiperbólica *tanh* (que dará valores entre -1 y 1) y se multiplicará por la salida de lo que denominaremos *output gate layer* (puerta de salida), que nuevamente se encarga de decidir qué irá a la salida. La ecuación del *output gate layer* será $o_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right)$ y la de la salida será $h_t = o_t * tanh\left(C_t\right)$.

El número de parámetros que se deben entrenar en una capa LSTM de es $params\_to\_train = [(num\_units + input\_dim + 1) * num\_units] * 4$.

Cada una de las capas de neuronas de la celda tiene como entradas la concatenación de $[h_{t-1}, x_t]$, más una entrada de bias (este es el motivo de que se le sume uno). Esta cantidad hay que multiplicarla por el número de unidades, y por supuesto, por cuatro (por las cuatro capas de neuronas que hay en cada unidad).

Para realizar los experimentos, se desarrolló un programa Python incluído en el apéndice de códigos A.15 que implementaba haciendo uso de la librería *Keras* el modelo representado en la figura 4.15, haciendo variar el número de celdas LSTM. Para poder comparar con los experimentos realizados anteriormente con otros tipos de modelos, se utilizó una capa de embedding pre-entrenada, en concreto, el embedding W2V Google que genera vectores de trescientas componentes, antes de la capa LSTM, y dos capas densas, una de cincuenta neuronas, y la última con una única

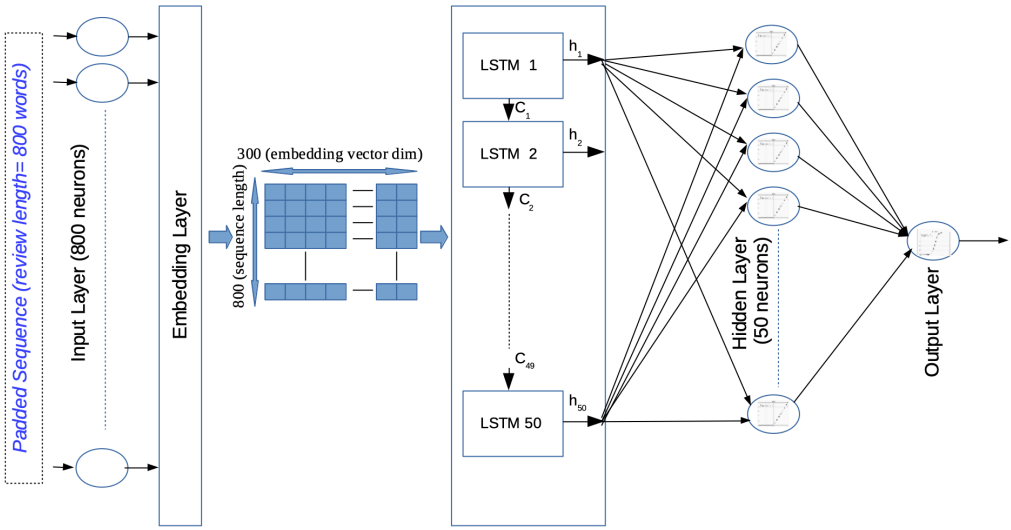


Figura 4.15. Modelo una capa LSTM 50 celdas

neurona.

Es importante tener en cuenta en el código de apéndice A.15 y en el resto de códigos que usen capas *LSTM* que hubo que modificar la codificación de las secuencias antes de introducirlas en la capa de embedding. En esta ocasión, el *padding* (relleno con ceros hasta completar y truncar al tamaño máximo de longitud del comentario) se ha añadido al comienzo de la secuencia, y no al final de la misma como en ejemplos anteriores. El motivo de esta modificación es que las capas LSTM trabajan en secuencia, y el *padding* de ceros al final de la secuencia causaba de algún modo que la red aprendiera esta secuencia de ceros, y no era capaz de entrenarse para

distinguir otras secuencias.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 800, 300)          6270900
_________________________________________________________________
lstm_3 (LSTM)                (None, 50)                70200
_________________________________________________________________
dense_5 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 51
=================================================================
Total params: 6,343,701
Trainable params: 72,801
Non-trainable params: 6,270,900
```

Figura 4.16. Modelo una capa LSTM 50 celdas

En la figura 4.16 podemos ver el resumen generado por el programa, con el detalle de los parámetros a entrenar. Como puede observarse, la capa de embedding consta de 6.270.900 parámetros no entrenables (ya que se trata de un embedding pre-entrenado, el W2V de Google). El número de parámetros es resultado de multiplicar el tamaño del vocabulario obtenido en las muestras de entrenamiento (20.902 palabras diferentes) más la entrada reservada para el padding (total 20.903) por el número de componentes de los vectores resultantes al aplicar el embedding a las palabras de los comentarios (300 componentes). El resultado pues es (20,903x300 = 6,270,900 parámetros). En lo que refiere a la capa LSTM de cincuenta celdas, el número de parámetros a entrenar es de 70.200. Este número es el resultado de aplicar la fórmula:

$$params\_to\_train = [(num\_units + input\_dim + 1) * num\_units] * 4 \quad (4.7)$$

explicada en párrafos anteriores. En esta ocasión, el número de unidades *LSTM* son 50, y el *input_dim* es la dimensión de los vectores que

representan cada palabra en la secuencia, y que hemos dicho que es 300. Por lo tanto el número de parámetros será [(50 + 300 + 1) * 50] * 4 = 70.200 parámetros a entrenar. Por último, la salida de la capa *LSTM* tiene un tamaño de 50, que conectada a una capa densa de 50 neuronas da un total de 50 x 50 = 2.500 a los que hay que sumar la entrada *bias* de las neuronas, haciendo un total de 2.550 parámetros para primera capa densa. Por último, la capa de salida tiene una única neurona a la que se conectan las 50 neuronas de la capa anterior. Lo que añade 51 parámetros más a entrenar (al añadir su correspondiente entrada de *bias*).

Tabla 4.5. Experimentos LSTM utilizando codificación Google W2V y entrenando capa embedding.

Codification	N. Cells	"Good"hits	"Bad"hits	"Good"fails	"Bad"fails	% Accu
Google W2V	30	1.250	1.277	100	158	90,74
Google W2V	50	1.263	1.274	103	145	91,1
Google W2V	100	1.224	1.290	87	184	90,27
Training Emb.	30	1.227	1.277	100	181	89,81
Training Emb.	50	1.239	1.260	117	169	89,73
Training Emb.	100	1236	1.258	119	172	89,55

Para poder comparar el impacto que tiene en el modelo usar un embedding pre-entrenado como el W2V Google, se modificó el programa Python anterior para sustituir la capa de embedding por una que se entrene junto con el modelo (lo que implica tener que entrenar 6.270.900 parámetros más). El código modificado se puede consultar en el apéndice de códigos A.16. Igual que antes, se entrenaron diferentes modelos variando el número de celdas entre treinta, cincuenta y cien. La tabla 4.5 muestra los resultados obtenidos de dichos experimentos, y gráficamente se resumen en la figura 4.17.

Tal y como se muestra resaltado en color verde en la tabla 4.5, el mejor

resultado se obtuvo cuando utilizamos 50 unidades *LSTM* con el embedding W2V de Google, obteniendo una precisión del 91,1 %. Comparamos este resultado con el obtenido en el modelo equivalente sin añadir la capa *LSTM* (utilizando Google W2V con una dimensión en la capa de entrada de 800 palabras y una red densa como la añadida al final de este mismo modelo) y que se mostró en la tabla 3.8 y en el que se obtuvo un 86,61 % de precisión. Como se puede observar, la mejora al añadir la capa *LSTM* de 50 unidades fue de **4,49 %**.

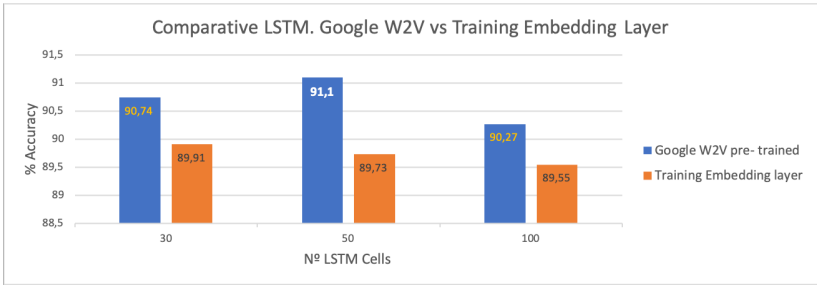


Figura 4.17. Comparativa de precisión LSTM usando codificación Google W2V y entrenando capa de embedding

De la figura 4.17 y de la correspondiente tabla 4.5 podemos sacar además las siguientes conclusiones:

- Aumentar el número de unidades *LSTM* no conlleva mejorar los resultados. Se obtuvieron mejores resultados con 50 unidades que con 100 unidades, pese a que la longitud de las secuencias era de 800 palabras. El número de unidades es dependiente del problema, y hay que realizar diferentes experimentos en su búsqueda.

- A diferencia con los modelos en los que solo usamos capas densas,

el utilizar un embedding pre-entrenado si mejoró los resultados en casi un 2 % con respecto a entrenar una capa de embedding con el modelo.

- Usar una capa de embedding que se entrena junto con el modelo también mejoró (aunque no tanto como usando el W2V de Google) los resultados de los modelos de capa densa en un 3,2 %.
- La capa *LSTM* se comportó de forma sensible a la codificación de las secuencias, y se tuvo que cambiar el relleno *padding* al comienzo de las mismas. Esto hace pensar que ajustando el tamaño de las secuencias podrían obtenerse otros resultados. Para los experimentos se ha decidido mantener el tamaño a 800 para poder comparar con los otros experimentos.

4.4. Arquitecturas combinadas CNN - LSTM.

Para completar el conjunto de experimentos planteados en este capítulo se realizó un modelo mixto de arquitectura usando capas convolucionales CNN y recurrentes LSTM. El objeto es plantear si realizando una extracción de características previa, las capas LSTM mejoraban su resultado. Para ello se combinaron los siguientes modelos:

- Se realizó una combinación del modelo de la figura 4.7 y el de la figura 4.16. En concreto:
  - Como método de codificación *embedding W2V Google*.

- Capa convolucional con 64 filtros y *kernel size* de tamaño 2.
- Capa *Maxpooling* con tamaño de ventana 2.
- Capa con 50 celdas LSTM.
- Capa densa de 50 neuronas.
- Capa densa de 1 neuronas para ralizar la clasificación binaria.
- El modelo resultantees el que se muestra en la figura 4.18.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 800, 300)          6270900
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 800, 64)           38464
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 400, 64)           0
_________________________________________________________________
lstm_1 (LSTM)                (None, 50)                23000
_________________________________________________________________
dense_1 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 51
=================================================================
Total params: 6,334,965
Trainable params: 64,065
Non-trainable params: 6,270,900
```

Figura 4.18. Modelo mixto convolucional - LSTM sin dropout.

- De forma similar, se generó un segundo modelo, esta vez realizando dos operaciones de convolución antes de la capa *LSTM*. El modelo resultante es el correspondiente a la figura 4.19
- Para completar los experimentos mixtos, y con el fin de comprobar si se estaban sobreentrenando (*overfitting*)los modelos se parametrizaron las capas internas con *dropout*. *Dropout* es un método mediante el cual se desactivan en la fase de entrenamiento neuronas de forma aleatoria, evitando así las dependencias generadas entre las mismas

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 800, 300)          6270900
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 800, 64)           38464
_________________________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 400, 64)           0
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 400, 64)           8256
_________________________________________________________________
max_pooling1d_3 (MaxPooling1 (None, 200, 64)           0
_________________________________________________________________
lstm_2 (LSTM)                (None, 50)                23000
_________________________________________________________________
dense_3 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 51
=================================================================
Total params: 6,343,221
Trainable params: 72,321
Non-trainable params: 6,270,900
```

Figura 4.19. Modelo mixto convolucional dos capas - LSTM sin dropout.

y que puedan aprender patrones de sus neuronas vecinas. La figura 4.20 y la figura 4.21 representan los modelos de los experimentos anteriores, añadiendo *dropout* a las capas internas convolucionales y densas.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_14 (Embedding)     (None, 800, 300)          6270900
_________________________________________________________________
conv1d_9 (Conv1D)            (None, 800, 64)           38464
_________________________________________________________________
dropout_4 (Dropout)          (None, 800, 64)           0
_________________________________________________________________
max_pooling1d_9 (MaxPooling1 (None, 400, 64)           0
_________________________________________________________________
lstm_14 (LSTM)               (None, 50)                23000
_________________________________________________________________
dense_24 (Dense)             (None, 50)                2550
_________________________________________________________________
dropout_5 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_25 (Dense)             (None, 1)                 51
=================================================================
Total params: 6,334,965
Trainable params: 64,065
Non-trainable params: 6,270,900
```

Figura 4.20. Modelo mixto convolucional - LSTM con dropout.

La tabla 4.6 muestra una comparativa de los experimentos realizados con modelos mixtos *CNN - LSTM . Dense*. Para todos ellos como ya se comentó se utilizó codificación *Google W2V* con secuencias de tamaño 800 y con relleno *padding* al final. El número de muestras utilizadas en el entre-

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_13 (Embedding)     (None, 800, 300)          6270900
_________________________________________________________________
conv1d_7 (Conv1D)            (None, 800, 64)           38464
_________________________________________________________________
dropout_1 (Dropout)          (None, 800, 64)           0
_________________________________________________________________
max_pooling1d_7 (MaxPooling1 (None, 400, 64)           0
_________________________________________________________________
conv1d_8 (Conv1D)            (None, 400, 64)           8256
_________________________________________________________________
dropout_2 (Dropout)          (None, 400, 64)           0
_________________________________________________________________
max_pooling1d_8 (MaxPooling1 (None, 200, 64)           0
_________________________________________________________________
lstm_13 (LSTM)               (None, 50)                23000
_________________________________________________________________
dense_22 (Dense)             (None, 50)                2550
_________________________________________________________________
dropout_3 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_23 (Dense)             (None, 1)                 51
=================================================================
Total params: 6,343,221
Trainable params: 72,321
Non-trainable params: 6,270,900
```

Figura 4.21. Modelo mixto convolucional dos capas - LSTM con dropout.

Tabla 4.6. Experimentos modelos CNN-LSTM-Dense.

Codification	Dropout	"Good"hits	"Bad"hits	"Good"fails	"Bad"fails	% Accu
1 CNN-LSTM	No	1.163	1.319	58	245	89,12
2 CNN-LSTM	No	1.267	1.288	89	141	91,74
1 CNN-LSTM	Yes	1.245	1.297	80	163	91,27
2 CNN-LSTM	Yes	1.244	1.294	83	164	91,13

namiento fue nuevamente de 9.640 balanceadas. El código utilizado para realizar los experimentos se muestra en el apéndice de códigos A.17.

De la figura 4.22 y comparándolo con los resultados obtenidos en los experimentos en los que no se combinaron las dos arquitecturas *CNN* y *LSTM* podemos obtener las siguientes conclusiones:

- La combinación de capas *CNN* y *LSTM* mejoró ligeramente el resultado obtenido si utilizamos únicamente *LSTM*, pero empeoró también ligeramente el resultado obtenido cuando utilizamos el mismo modelo eliminando la capa *LSTM* ver figura 4.11. Esto implica, que realizar las convoluciones tiene un impacto más determinante y positivo en el problema en estudio.

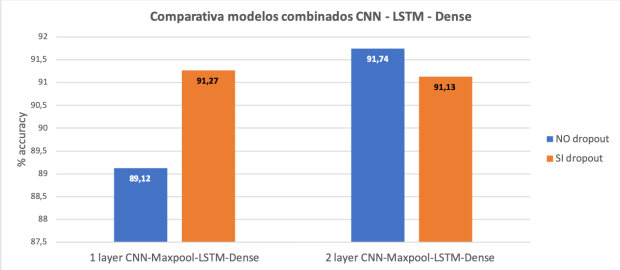


Figura 4.22. Comparativa de modelos mixtos.

- Las muestras utilizadas no producen *overfitting*. El impacto de usar *dropout* no fue relevante en este experimento.

Como conclusión final de los experimentos realizados en este capítulo, se ha podido comprobar que:

- Con el número de muestras reducido que se ha utilizado (9.640) la variante *nu-SVC* de la familia de algoritmos *SVM* usando una función *kernel* lineal ha obtenido el mejor resultado en precisión con un 92,57 %.

- Con una precisión considerable, las CNN se comportan bien con este modelo. Cuando se usaron 64 filtros, la precisión llegó a alcanzar un 92,14 %, aproximándose bastante al resultado de algoritmos SVM pese a la baja cantidad de muestras.

- Las capas LSTM para este problema en concreto mejoraron el resultado del obtenido únicamente con capas *densas*, pero no aportaron tanto como las CNN. Estas últimas, mejoraron el rendimiento de las LSTM en los modelos combinados.

- Se comprobó en los modelos combinados que no se estaba produciendo *overfitting*, siendo irrelevante el introducir ajustes como el *dropout*.
- En todos los modelos probados se observa que la predicción de comentarios negativos es mejor siempre que la de positivos. El número de fallos en comentarios positivos se duplica como normal general.
- Los resultados obtenidos en precisión son muy positivos, superando el 92 % en algunos modelos. Sin embargo, los modelos *deep-learning* suelen requerir de un conjunto de datos de entrenamiento superior al que tenemos, por lo que a falta de muestras de este dominio concreto parece conveniente utilizar técnicas que permitan suplir esta carencia.

Los resultados en precisión obtenidos por los modelos presentados en este capítulo permiten ya realizar predicciones sobre las opiniones de los turistas. El hecho de que el conjunto de datos de entrenamiento específico del dominio que hemos usado (extraído de portales especializados como *Tripadvisor* o *Booking*) fuera reducido nos indica que es posible aumentar algún punto la precisión de los mismos. En los siguientes capítulos se presentarán algunas técnicas que tratarán de evitar este inconveniente.

Capítulo 5

Técnicas de mejora cuando el conjunto de datos de entrenamiento es reducido.

En el capitulo 3 se repasaron diferentes métodos de codificación para poder representar los comentarios de los turistas (textos en inglés) de forma numérica, y se presentó el estudio comparativo de resultados obtenidos para nuestro dominio completo y el impacto que tenía en la precisión de los modelos.

En el capítulo 4 se introdujeron diferentes técnicas de *Machine Learning ML*, concretamente la familia de algoritmos SVM y diferentes modelos de redes neuronales basados en *Redes Neuronales Convolucionales CNN y Long Short Term Memory LSTM*. Pese a los buenos resultados obtenidos, el tamaño del conjunto de datos de entrenamiento es un indicio de que la precisión de los modelos tiene aún margen de mejora.

En el presente capítulo se presentan diferentes soluciones a un problema relativamente habitual cuando se aplican técnicas de DL: cómo mejo-

rar los resultados de los modelos cuando se dispone de un conjunto de entrenamiento con muestras reducidas.

Al final del mismo se podrá acceder a los resultados obtenidos en este trabajo de investigación sobre la aplicación de las técnicas a los modelos más prometedores estudiados en capítulos anteriores.

5.1. Distant Supervision

Distant supervision es una técnica de apoyo a los métodos *deep learning* para resolver el problema de la obtención de datos etiquetados para entrenar los modelos. Cómo ya se vio en el capítulo 3 no es tarea sencilla obtener datos estructurados y etiquetados de portales especializados o redes sociales, ya que son fuentes de información tecnológicamente muy cambiantes y requieren de una investigación continua de las fuentes y programación de robots para la extracción. Tampoco resulta sencillo, ni mucho menos económicamente viable, contratar humanos expertos en el dominio (en nuestro caso no requiere de conocimientos adicionales, ya que se trata de etiquetar un comentario de un turista como bueno o malo) para etiquetar miles o cientos de miles de de muestras.

El uso de emoticones y cadenas de símbolos para representar emociones en los textos (y más concretamente en su uso en las redes sociales) ofrece igualmente una oportunidad en el procesamiento del lenguaje natural y en el análisis de sentimientos. La técnica *distant supervision* es utilizada para el etiquetado de textos en base a la aparición de estos símbolos, muy frecuentes en mensajes cortos de diferentes redes sociales, y que no

podrían ser usados en problemas de clasificación mediante *deep learning* si no son previamente etiquetados (Suttles e Ide, 2013).

Del mismo modo, y aunque no sea objeto de este trabajo de investigación, el uso de *distant supervision* es de gran utilidad para la extracción de relaciones entre entidades. Frecuentemente, dentro del procesamiento del lenguaje natural y el análisis de sentimientos, es necesario discriminar la entidad sobre la cuál se está expresando una opinión o un sentimiento. Concretamente, en el dominio de opiniones de turistas podría darse el caso que en un mismo comentario existieran valoraciones positivas sobre algún aspecto de un servicio hotelero, y negativas sobre otros dentro del mismo comentario. En las clasificaciones presentadas en este trabajo no se ha realizado una discriminación por entidades, pero si fuera necesario es especialmente complicado obtener datos etiquetados que hicieran esta discriminación. Por este motivo, la técnica *distant learning* puede ser aplicada para realizar un procesamiento previo del comentario y relacionarlo con entidades características del dominio para su posterior clasificación (Broß, 2013).

La aplicación de esta técnica se considera un paso previo a otras como las que se verán en secciones posteriores (por ejemplo *transfer learning*) en el sentido que es necesario generar una gran cantidad datos etiquetados a partir de un conjunto más reducido, que puedan ser utilizados posteriormente. Lamentablemente, los datos generados suelen tener problemas de ruido (errores a la hora de etiquetar) (Su P, 2019).

En relación con la aplicación de *distant learning*, existen herramientas que permiten el etiquetado de datos en base a la construcción de reglas personalizadas (Ratner *et al.*, 2017). Para su uso en *Python* existe el frame-

work *Snorkel Metal* (Snorkel, n.d.).

En el desarrollo de este trabajo de investigación no se abordó la implantación de *distant supervision*, debido a que como se podrá comprobar en la siguiente sección pudimos acceder a bases de datos existentes con numerosos datos etiquetados que permitían el entrenamiento de los diferentes modelos mediante *transfer learning*.

5.2. Transfer Learning

Los modelos y técnicas de *machine learning* y en concreto de *deep learning* que se han tratado en capítulos anteriores (exceptuando aquellos que en los que usamos *embedding* ya entrenados como el caso de *Google w2v y Groove*) fueron concebidos para entrenarse y usarse de forma aislada. Los métodos y técnicas que aplican *transfer learning* tratan de sobreponerse a esta situación, tratando de usar el conocimiento que se haya adquirido en otra tarea relacionada (Dipanjan Sarkar, 2018). En concreto, en el problema de estudio en este trabajo *clasificación de las opiniones de turistas*, que forma parte del grupo de problemas de *análisis de sentimientos* las técnicas de *transfer learning* son especialmente relevantes, ya que se pueden usar modelos preentrenados con datos obtenidos sobre una gran variedad de productos (Pan e Yang, 2010).

Transfer learning añade a las técnicas de aprendizaje profundo dos ventajas fundamentalmente:

- Por un lado aumenta el rendimiento del modelo. Desde el inicio ya

se parte con un conocimiento (el transferido desde la tarea fuente) y no desde cero. Al final, este este conocimiento de partida se enriquece con conocimiento específico de la nueva tarea a realizar.

- Por otro lado se optimiza el tiempo para entrenar y afinar el modelo, ya que parte del mismo contiene el conocimiento de la tarea fuente.

Al la hora de aplicar técnicas de *transfer learning* se debe tener en cuenta:

- Cuál es el conocimiento que se desea transferir. Hay que identificar qué partes de la tarea fuente que contiene el conocimiento son aplicables a la tarea destino.
- Cuándo es conveniente o no transferir conocimiento. Debemos analizar si la transferencia está mejorando o no la tarea destino.
- Cómo realizar la transferencia. Dependiendo de la tipología de las tareas, la aplicación de técnicas de *transfer learning* requieren de conocimiento específico, ya sea para adaptar el conocimiento, o bien para introducirlo en el modelo de la nueva tarea.

La enorme difusión que ha tenido en los últimos años las técnicas de *machine learning* y en concreto *deep learning* han traído consigo que diferentes equipos de investigación pongan a disposición de la comunidad modelos, configuraciones y datos con los que han conseguido buenos resultados, y que forman la base fundamental en la aplicación de las técnicas de *transfer learning*, principalmente aplicada a técnicas de reconocimiento de imágenes, pero también a técnicas de procesamiento del lenguaje natural como las que aplicamos en este trabajo. Algunas de las técnicas que más se usan son las siguientes:

- **Extracción de características**. Como ya se ha comentado anteriormente, muchos de los modelos que hemos desarrollado en capítulos anteriores se concebieron para entrenarse y usarse de forma asilada. Algunos de ellos estaban formados por diferentes capas de neuronas con funciones diferentes. Las capas de embedding aprendían una representación densa de palabras de un vocabulario (expresadas en un idioma concreto), o las capas convolucionales, mediante la aplicación de filtros aprendían características locales de los comentarios. El conocimiento aprendido en estas capas puede ser transportable a otros problemas de similares características. De hecho, y aunque no tocó explicarlo en ese momento, ya realizamos *transfer learning* al utilizar los pesos de la matriz de embedding preentrenados, cuando utilizamos los embeddings *W2V Google* y *Glove* de la sección 3.4.

- **Afinado (tunning)**. En esta ocasión la transferencia no se realiza mediante la utilización de capas del modelo origen ya entrenadas y que no vuelven a entrenarse, sino que se realizar un entrenamiento adicional de las mismas en el modelo destino para una mejor adaptación al dominio.

- **Uso de datos de otros problemas**. Uno de los inconvenientes de las técnicas de *deep learning* es que se necesitan grandes volúmenes de datos para entrenar los modelos. En ocasiones, no es posible obtener todos los datos necesarios, por lo que es interesante utilizar datos de otros problemas. En concreto, en nuestro caso de aplicación, se explicó en el capítulo 3 cómo se realizó la extracción de información de los portales turísticos, pero como se comentó en ese momento, el

cambio en la estructura y organización de los mismos no permitía una extracción continua de los comentarios de los turistas. Existen bases de datos compartidas con comentarios escritos en el idioma *inglés* sobre otros productos (como pueden ser películas, o restaurantes). Pese a no ser exactamente el mismo dominio (no es lo mismo expresar la satisfacción sobre los servicios o instalaciones de los hoteles que críticas sobre una película), los datos pueden servir para aumentar y mejorar el entrenamiento de nuestro problema, con cierta adaptación del dominio fuente.

- **Uso de modelos ya entrenados**. Existen modelos ya entrenados por diferentes grupos de investigación, y que han sido puestos a disposición de la comunidad. En concreto, las aportaciones más significativas se han realizado en el campo del reconocimiento de imágenes.

- **Creación de nuevos datos de entrenamiento *data augmentation***. Mediante diferentes transformaciones podemos aumentar nuestro conjunto de datos de entrenamiento, generando nuevas muestras que mejoren el entrenamiento del modelo. A modo de ejemplo puede consultarse la propuesta por (Kobayashi, 2018).

Para poder aplicar las diferentes técnicas comentadas en párrafos anteriores utilizaremos diferentes conjuntos de datos de entrenamiento. Estas son las referencias de algunos de ellos.

- Conjunto de comentarios de clientes de Amazon con más de treinta y cinco millones de comentarios (McAuley y Leskovec, 2013) y construido por Xiang Zhang (Zhang *et al.*, 2015). Este conjunto de datos está disponible en la URL referenciada en (Zhang, n.d.).

- El conjunto de datos Yelp Dataset, disponible en la URL referenciada en (YelpDataset, n.d.) dispone de 3.6 GB de comentarios de clientes de diferentes negocios de restauración.
- IMDB es un conjunto de comentarios de crítica de películas, utilizado ampliamente en la bibliografía sobre NLP. Está disponbile en la referencia (IMDB, n.d.).

En la sección posterior se comentarán los resultados de investigación obtenidos en este trabajo al aplicar algunas de las técnicas de *transfer learning* vistas anteriormente.

5.3. Experimentos con Transfer Learning y resultados.

De las técnicas vistas en la sección anterior se han seleccionado las siguientes:

- Uso de capas de codificación *embedding* entrenadas previamente. En concreto se utilizó *Google Word2Vec* para producir vectores densos de 300 componentes a partir de secuencias codificadas de longitud 800, que presentó mejores resultados que *Glove* como se pudo comprobar en capítulos anteriores.
- Uso de conjuntos de entrenamiento públicos, de dominios parecidos (opiniones de usuarios de otros servicios). En concreto:
  - IMDB. Críticas de películas.

- Yelp. 3,6GB de opiniones de usuarios de servicios de restauración.
- Amazon. Treinta y cinco millones de comentarios sobre productos.

La estrategia que se utilizó para realizar los experimentos fue la siguientes:

- En primer lugar se realizaron experimentos con los datos de *IMDB*, utilizando el modelo de red neuronal que mejores resultados obtuvo en capítulo 4. El motivo de esta elección, es que es un conjunto de datos más amplio que el obtenido de los portales especializados (25.000 muestras frente a las 9.640 obtenidas), pero lo suficientemente pequeño para comprobar el rendimiento del modelo sin consumir un gran número de horas de computación.
- Posteriormente a utilizar *IMDB*, y en función de los resultados obtenidos, se procedió a realizar entrenamientos del modelo utilizando datos de *Amazon* y *Yelp*, ya que debido a su tamaño, el entrenamiento se prolongaba durante más de un mes en máquinas de altas prestaciones.

La tabla 5.1 muestra la estructura por capas de los modelos utilizados en estos experimentos. Las posteriores tablas de resultados y gráficas harán referencia a la columna *Nombre* para representar el modelo. Para poder tener una referencia comparativa, en el capítulo 4 en la tabla 4.3 se pudo comprobar que el *Modelo 2* obtuvo una precisión del 92,14 % sin aplicar técnicas de *transfer learning*.

Tabla 5.1. Descripción de los modelos utilizados en experimentos Transfer Learning.

Nombre	Descripción de capas
Modelo 1	Capa de Embedding entrenada junto con el modelo Capa convolucional 64 filtros, kernel size = 2 Capa maxpooling window size = 2 Capa convolucional 64 filtros, kernel size = 2 Capa maxpooling window size = 2 Capa aplanado Capa densa 50 neuronas, función activación relu Capa densa 1 neurona, función de activación sigmoide
Modelo 2	Capa de Embedding pre-entrenada Google W2V Capa convolucional 64 filtros, kernel size = 2 Capa maxpooling window size = 2 Capa convolucional 64 filtros, kernel size = 2 Capa maxpooling window size = 2 Capa aplanadoCapa densa 50 neuronas, función activación relu Capa densa 1 neurona, función de activación sigmoide

Los resultados obtenidos al realizar entrenamientos con el conjunto de datos de *IMDB* son los mostrados en la tabla 5.2.

Tabla 5.2. Experimentos usando IMDB.

Modelo	"Good"hits	"Bad"hits	"Good"fails	"Bad"fails	%Accu
Modelo 1	1.126	1.037	340	282	77,67
Modelo 2	1.075	1.143	234	333	79,64

Como se puede observar, ninguno de los experimentos usando el conjunto de datos de *IMDB* mejoró el resultado inicial de 92,14 %. Sin embargo, si se observa comparando entre los resultados del *Modelo 1* y *Modelo 2* que mejora el resultado en 1,97 % si se utiliza una de las técnicas *transfer learning* explicadas (usar una capa de *embedding* entrenada como Google W2V).

Sobre el *Modelo 2* se realizó un nuevo entrenamiento utilizando los da-

tos específicos del dominio (los utilizados en los entrenamiento del capítulo 4). Con este segundo entrenamiento, la precisión definitiva del modelo fue de un **91,7** % que aunque no mejoró la precisión inicial del 92,14 % aumentó considerablemente la conseguida antes de realizar el segundo entrenamiento.

A la vista de estos resultados se realizó el entrenamiento usando el conjunto de datos de *Amazon* que contiene comentarios de usuarios sobre sus productos. Al tratarse de un conjunto de entrenamiento muy grande el tiempo empleado para entrenar los modelos superaba las cuatros semanas. Por este motivo, se realizaron los experimentos sobre el *Modelo 2* que obtuvo mejores resultados en el experimento previo. En la tabla 5.3 se muestran los resultados obtenidos. La fila etiquetada como *Modelo 2. 2nd Train* hace referencia a que se realizó un segundo entrenamiento del modelo utilizando los datos específicos del dominio (los utilizados en los entrenamiento del capítulo 4).

Tabla 5.3. Experimentos usando Amazon data set.

Modelo	"Good"hits	"Bad"hits	"Good"fails	"Bad"fails	%Accu
Modelo 2	1.330	1.248	129	78	92,57
Modelo 2. 2nd Train	1.271	1.285	92	137	91,78

Como se puede comprobar, en esta ocasión, si se mejoró la precisión del modelo inicial al aplicar *transfer learning* usando un conjunto de datos de entrenamiento mayor, logrando un resultado final del **92,57 %**. Sin embargo, se observa también que un segundo entrenamiento utilizando los datos del dominio no mejoraron los resultados, sino todo lo contrario. La explicación que encontramos en la ausencia de mejora es que la proporción de datos del dominio es muy pequeña en relación al conjunto

total.

Con estos resultados, se optó por realizar un último experimento, utilizando los datos del conjunto de entrenamiento de *Yelp*, descartando realizar un segundo entrenamiento con los datos del dominio, ya que no mejoraron los resultados en la prueba anterior realizada con el conjunto de *Amazon*. Los resultados obtenidos en esta ocasión se muestran en la tabla 5.4.

Tabla 5.4. Experimentos usando Yelp data set.

Modelo	"Good"hits	"Bad"hits	"Good"fails	"Bad"fails	%Accu
Modelo 2	1.329	1.323	54	79	95,22

Como se puede observar, la aplicación de la técnica de *transfer learning* en dos de sus modalidades: usando una capa de *embedding* entrenada previamente como es *Google Word2Vec*, y usando un conjunto de datos de entrenamiento externo como *Yelp data set* logra mejorar los resultados obtenidos, con una precisión final de **95,22 %**.

Las conclusiones obtenidas en este trabajo de investigación, en lo referente a la aplicación de técnicas de *distant learning* y *transfer learning* se pueden resumir en:

- Las técnicas de *distant learning* son especialmente útiles para construir conjuntos de datos de entrenamiento etiquetados a partir de datos no etiquetados. También son ampliamente utilizadas para etiquetar diferentes aspectos de un mismo comentario. En este trabajo de investigación no fueron utilizadas porque se disponía de conjuntos de datos de entrenamiento externos correctamente etiquetados.

- Las técnicas *transfer learning* utilizadas se centraron en la utilización de capas de *embedding* externas y el uso de datos de entrenamiento de fuentes externas. Concretamente se utilizó para los experimentos el *embedding Google Word2Vec* y los conjuntos de entrenamiento que hemos referenciado como *IMDB, Amazon y Yelp*.

- Se ha podido comprobar que la aplicación de técnicas de *transfer learning* no siempre mejoran los resultados. Es importante conocer cuál es la tarea que se está transfiriendo y la información que maneja. Más aún, si tenemos en cuenta que los entrenamientos con conjuntos de datos grandes (como el caso de *Amazon y Yelp*) usaban más de un mes de cómputo de un ordenador de altas prestaciones.

- La aplicación conjunta de dos de las técnicas de *transfer learning* mejoraron de manera considerable el resultado en precisión del modelo, logrando un alto porcentaje de aciertos **95,22 %**. La explicación de este resultado se encuentra en que el conjunto de entrenamientos de *Yelp* tiene un dominio parecido al problema original en estudio, ya que ambos son comentarios de usuarios de servicios (*Yelp* en restauración, y el problema original son comentarios de usuarios de servicios hoteleros). A tenor de los resultados, también se observa que el *embedding de Google Word2Vec* produce mejores resultados en modelos entrenados con conjuntos de datos grandes.

En este capítulo se han analizado diferentes técnicas para evadir el problema de disponer de conjuntos de datos de entrenamiento reducidos. Se ha mejorado considerablemente el resultado de precisión de los modelos del capítulo 4. En el siguiente, se introducirán nuevas técnicas destinadas a mejorar la precisión combinando predicciones de diferentes modelos.

Capítulo 6

Ensemble

En el capítulo 3 se realizó un repaso a las diferentes técnicas de codificación de textos en vectores numéricos, para poder usar técnicas DL como las expuestas en el capítulo 4. Los experimentos realizados con redes neuronales CNN y LSTM mostraron que los resultados en predicción eran muy prometedores pese a tener un número de muestras reducido para el uso de estas técnicas.

En el capítulo 5 se expusieron algunas alternativas para poder esquivar el problema de los conjuntos de datos de entrenamiento reducidos, haciendo uso de la técnica conocida como *transfer learning*. Sin embargo, ha quedado en evidencia que las técnicas DL tienen una alta varianza en lo que a efectividad se refiere. Es muy complicado saber a priori qué modelo es el que se ajusta mejor al dominio, cuáles son los ajustes que deben realizarse, o cómo elegir los datos para realizar el entrenamiento.

Las técnicas de *Ensemble* tratan de abordar estos problemas, mejorando la estrategia mediante la combinación de diferentes elementos.

6.1. Ensemble

Ensemmbling es la técnica mediante la cual se combinan dos o más modelos de *machine learning* para obtener uno con mayor poder de predicción (George Kyriakides, 2019).

Las redes neuronales son métodos no lineales, lo cual permiten implementar modelos muy flexibles. Sin embargo, como se ha visto en secciones anteriores tienen cierta componente aleatoria (la red se inicializa con unos pesos aleatorios, y el conjunto de datos de entrenamiento, así como el orden en el que se escogen influyen en él). Los dos principales problemas que aparecen en el entrenamiento de los modelos de *deep learning* son el *bias* y la *varianza*. El *bias o sesgo* hace referencia a la diferencia entre la predicción y la realidad, y normalmente responde a la incapacidad del modelo en comportarse como el sistema objetivo. La *varianza* en el contexto de la estadística hace referencia a la diferencia de las muestras con respecto de la media, sin embargo, cuando se habla de redes neuronales hace referencia a la sensibilidad de la predicción con respecto al conjunto de entrenamiento. Un modelo, que se ajusta demasiado al conjunto de entrenamiento (lo cual se conoce como *overfitting*) podría tener malas predicciones cuando se encuentra con una muestra que no pertenece a dicho conjunto. Los modelos muy sensibles al conjunto de datos de entrenamiento tienen una *varianza* alta. Cuando tratamos de mejorar el sesgo haciendo que el modelo se ajuste mejor a la muestra se aumenta la varianza (normalmente se produce *overfitting*), y si se trata de disminuir la varianza simplificando el modelo sin hacerlo depender tanto del entrenamiento aumentará el sesgo (*underfitting*).

Utilizando técnicas de *ensemble* se pueden atenuar los problemas de *bias* y *varianza*, ya que al combinar diferentes modelos se puede reducir el error del conjunto mientras se conserva la complejidad de los modelos individuales.

Las técnicas de *ensemble* consisten pues en la utilización de diferentes modelos, haciendo variar alguno de los siguientes elementos (Brownlee, n.d.):

- El conjunto de datos de entrenamiento.
- Los modelos que pertenecen al *ensemble*.
- La forma en la que se combinan los resultados de los modelos.

6.1.1. Ensemble mediante la variación del conjunto de datos de entrenamiento

Una de las estrategias habituales para realizar *ensembles* consiste en usando el mismo modelo, realizar variaciones en el conjunto de datos que se utiliza para entrenarlo. De esta forma, los diferentes modelos entrenados pueden tendrá una perspectiva relativamente diferente y en conjunto podrán lograr predicciones más estables. Entre las técnicas más habituales se encuentran:

- **Random splits (particiones aleatorias)**. En esta técnica se entrenará el mismo modelo, pero tomando particiones aleatorias para el conjunto de entrenamiento y el conjunto de prueba.

La forma de realizarlo es haciendo uso de la función *train_test_split* de la librería scikit-learn (SciKitProject, n.d.), que permite expresar mediante un tanto por uno la fracción de datos de un conjunto que será utilizado como prueba o test. El funcionamiento general de la técnica es el siguiente:

- Con todo el conjunto de datos se realiza una partición aleatoria de todos, tomando un fracción de los mismos como test y el resto como datos de entrenamiento.
- Se genera un nuevo modelo que se entrena con los datos obtenidos anteriormente
- Guardar el modelo para poder ser usado en las predicciones del *ensemble*.

El parámetro configurable de esta técnica es el número de *splits* o particiones aleatorias que se generan, así como la fracción de datos que se reserva para las pruebas de validación. No existe ninguna regla para la elección de estos parámetros por lo que se recomienda la experimentación y observación de los resultados. La librería scikit-learn (SciKitProject, n.d.) contiene la función *train_test_split* que nos permite partir el conjunto de datos como hemos especificado anteriormente, mediante el parámetro *test_size* podremos especificar la fracción (en tanto por uno) de las muestras que se destinarán a test.

- **k-fold cross validation (validación cruzada)**. En esta técnica se divide el conjunto de entrenamiento en k partes iguales y se usan igualmente k modelos iguales. El modelo j usará como conjunto de prue-

ba la partición *j*, y como conjunto de entrenamiento el resto de conjuntos. En general, el método *k-fold cross validation* reduce el sesgo o el optimismo de los métodos durante el test. El funcionamiento general de la técnica es el siguiente:

- Se mezclan y barajan los datos de entrenamiento y test.
- Se dividen estos datos en k partes. Se toma una de las partes como test, y el resto de partes como entrenamiento.
- Entrenar el modelo con las k-1 partes y evaluar con la parte reservada para test.
- Guardar el modelo para poder ser usado en las predicciones del *ensemble*

Esto se hace con cada parte, utilizándola como data test y usando el resto como entrenamiento. Como las partes se conservan, cada muestra habrá participado una vez como test y k-1 veces como dato de entrenamiento del resto de modelos.

No existe una regla generalizada para darle valor al parámetro *k*. Debe ser un valor que haga el tamaño de las particiones medianamente representativo. Se debe tener en cuenta que a medida que aumenta *k* el tamaño del conjunto de entrenamiento en cada partición aumenta con respecto al de tamaño del conjunto de test, y puede hacer el sesgo mayor. En *deep learning* valores de k habituales suelen ser 5 o 10. En cualquier caso, debe evitarse dejar particiones de resto con pocas muestras. Existen además diferentes variantes de la técnica, como pueden ser llevar el número de particiones a los extremos

(dos particiones, una de test y otra de datos o particiones de una sola muestra, lo que equivale casi a entrenar con el total de los datos y usar una sola muestra de test), o bien barajar los datos nuevamente en cada iteración.

La librería scikit-learn (SciKitProject, n.d.) tiene las herramientas para poder aplicar la técnica *k-fold cross validation* en Python sin tener que progamarlo desde cero. Por medio del uso de la función *KFold* se podrá dividir un conjunto de datos y etiquetas en k particiones diferentes teniendo la posibilidad de barajarlos de forma aleatoria previamente.

- **Bootstrap Agregation o Bagging (empaquetado)**. Esta técnica tradicionalmente utilizada en árboles de decisión consiste en construir nuevos conjuntos de datos de entrenamiento, llamados *bootstraps* a partir del conjunto de datos de entrenamiento original. La forma en la que que se forman estos nuevos conjuntos de datos es la que da nombre a la técnica *bagging*, ya que, una vez decidido el tamaño de los conjuntos (que puede ser igual que el del conjunto de partida) se van tomando muestras del conjunto original, con la peculiaridad de que una vez tomada una muestra para el subconjunto, ésta se devuelve a la *bolsa* de muestras de partida, pudiendo ser tomada de nuevo (por lo que una misma muestra puede encontrarse repetida varias veces en los subconjuntos).

  La manera de funcionar es similar a las anteriores: cada modelo del *ensemble* es entrenado con un *bootstrap* formado como se ha comentado anteriormente, para posteriormente ser usado en la predicción. La técnica *bootstrap* ha sido utilizada ampliamente en algoritmos con

una alta varianza.

La librería scikit-learn (SciKitProject, n.d.) nos ofrece la función *resample* que nos permite seleccionar diferentes *bootstrap* con remplazamiento.

Cualquiera de las técnicas de ensemble mediante variación de los datos de entrenamiento vistas: *Random splits, k-fold cross validation, o bootstrap agregation* genera un conjunto de modelos exactamente iguales, que han sido entrenados con diferentes conjuntos de datos (los obtenidos mediante la aplicación de las técnicas). Posteriormente a esta fase, tenemos que realizar una predicción mediante la combinación de las predicciones de cada uno de los modelos individuales. Las maneras más habituales de combinar las predicciones, y que han sido desarrolladas en este trabajo de investigación son las siguientes:

- **Media aritmética de las predicciones individuales**. Como su nombre indica, el resultado de la predicción del ensemble se calcula mediante la media aritmética de cada una de las predicciones individuales, es decir, se suman las predicciones de cada uno de los modelos entrenados con los subconjuntos de datos y se divide por el número de modelos utilizados en el *ensemble*.
- **Media ponderada de las predicciones individuales**. El principal problema de calcular la predicción del *ensemble* calculando la media de la predicción expresada en probabilidad de cada uno de los modelos que lo componen es que todos contribuyen en igual medida. Cuando en experimentos anteriores ya se ha comprobado que la precisión de alguno de ellos es generalmente mejor no parece ra-

zonable calcular simplemente la media de las predicciones. Por este motivo es importante dar más valor a la predicción emitida por aquellos modelos que en general se comportan mejor. La manera de hacerlo es dar mayor *peso* a las predicciones de los modelos más precisos. La media ponderada que sirva de predicción al ensemble será entonces la suma de los productos de las probabilidades de predicción de cada modelo que lo componen por su peso correspondiente. El valor de los pesos deberá ser un número real comprendido entre 0 y 1, teniendo en cuenta que la suma total de los pesos no debe superar a 1 (para que la probabilidad de predicción del *ensemble* esté igualmente normalizada entre 0 y 1. Para encontrar el valor de cada peso existen diferencias estrategias:

- **Buscando combinaciones de pesos mediante fuerza bruta**. Esta técnica consiste en hacer variar los pesos de forma iterativa en pasos discretos (por ejemplo en valores de 0.1), controlando siempre que la suma de los mismos no supere el valor 1. Una vez se hayan probado todas las combinaciones se deberá elegir el vector de pesos que haya maximizado la precisión del conjunto total del ensemble. La funcionalidad *product* de la librería *itertools* de Python nos permite realizar todas las posibles combinaciones de un conjunto que le suministremos tomando el número de modelos que tenemos en el *ensemble*. Así, pasándole todos los valores entre 0 y 1 con diferencia de 0.1 décima podríamos probar todas las posibles combinaciones de pesos con un solo decimal. Es importante reseñar que para cada combinación de posibles pesos, hay que comprobar su precisión con un subconjunto de datos, que será una pequeña extracción del

conjunto original. Si elegimos pocas muestras para este subconjunto podemos obtener pesos erróneos, por el contrario, a medida que aumentamos el número de muestras para elegir los pesos el número de horas de computación crece drásticamente.

- **Asignando pesos mediante un algoritmo de búsqueda**. Como se explicó anteriormente la técnica de búsqueda de pesos para la media ponderada mediante fuerza bruta puede tener un alto coste computacional. Para evitarlo, se pueden utilizar algoritmos de optimización. En este trabajo de investigación, y a modo de ejemplo se ha utilizado el algoritmo de *Evolución Diferencial* publicado en el año 1.995 (Storn y Price, 1995), incluido en la librería *SciPy* (SciPy, n.d.).

- **Votación de los modelos**. La predicción de cada uno de los modelos, tiene como salida un número real entre 0 y 1, que indicará la probabilidad de que el comentario evaluado sea de la clase *"Good"* bueno, o de la clase *"Bad"* malo. Las técnicas de media aritmética y ponderada realizan cálculos con esta probabilidad. La técnica de *Voting* o *votación de los modelos* por el contrario seleccionará como predicción la que más se haya repetido entre los modelos del ensemble. La técnica toma el nombre del hecho que entre los modelos votan aquella predicción más probable. En caso que el conjunto de modelos sea par, y para este trabajo de investigación, hemos elegido dar preferencia a la clase *"Bad"* por haber tenido más certeza en experimentos anteriores.

Para probar la precisión de los métodos de *ensemble* mediante variación

de los datos de entrenamiento, combinando sus predicciones mediante media aritmética o media ponderada se ha desarrollado el código incluido en el apéndice de códigos A.18. En el mismo, se evalúan las técnicas *Random splits*, *k-fold cross validation*, *o bootstrap agregation* con un modelo de red neuronal básico de perceptrón multicapa, con una capa oculta de 50 neuronas y una capa de salida con una única neurona, como el presentado en el capítulo 3 en figura 3.10. Para realizar la codificación de los comentarios de los turistas se utilizó previo al modelo una capa de *embedding* entrenada previamente *Google pre-trained Word2Vec* que genera vectores de dimensión 300 a partir de secuencias de longitud 800, como los presentados en la sección 3.5.3. Para obtener una referencia de la ganancia en precisión obtenida al aplicar las técnicas de *ensemble* vistas hasta el momento debemos tener en cuenta, que la precisión obtenida con el modelo comentado, para ese tamaño de entrada fue de **86,61 %** como se reflejó en la tabla 3.8 del mismo capítulo 3.

Los experimentos se han realizado con el conjunto de entrenamiento usado en capítulos anteriores, formado por 9.640 comentarios completamente balanceados (igual número de comentarios positivos y negativos), un pequeño conjunto de test formado por 80 comentarios, que para formar las particiones se ha añadido al conjunto de datos de entrenamiento (formando así un conjunto de 9.720 muestras). Para comprobar los resultados se utilizó una conjunto de validación independiente de 2.785 comentarios.

La figura 6.1 muestra de forma gráfica las precisiones obtenidas con las técnicas anteriormente descritas. La tabla 6.1 muestra el detalle de los resultados obtenidos. Como se puede observar, la aplicación de la técnica

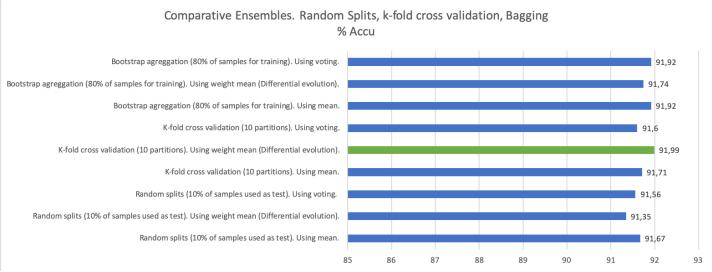


Figura 6.1. Comparativa técnicas de ensemble variando los datos de entrenamiento con un modelo MLP.

en todas sus variantes mejoró con el mismo conjunto de entrenamiento la precisión del modelo original que era de un **86,61 %**. El mejor resultado se obtuvo usando *k-fold cross validation* y calculando la media ponderada de las predicciones, obteniendo un **91,99 %**, lo que significa que se aumentó la precisión en un **5,38 %**.

Tabla 6.1. Comparativa técnicas de ensemble variando los datos de entrenamiento con un modelo MLP.

Codification	"Good"hits	"Bad"hits	"Good"fails	"Bad"fails	% Accu
Rand-splits. Mean.	1.254	1.299	78	154	91,67
Rand-splits. W-Mean.	1.253	1.291	86	155	91,35
Rand-splits. Voting.	1.249	1.301	76	159	91,56
K-fold. Mean.	1.257	1.297	80	151	91,71
K-fold. W-Mean.	1.262	1.300	77	146	91,99
K-fold. Voting.	1.253	1.298	79	155	91,60
Bagging. Mean.	1.260	1.300	77	148	91,92
Bagging. W-Mean.	1.295	1.260	117	113	91,74
Bagging. Voting.	1.253	1.307	70	155	91,92

6.1.2. Ensemble mediante la combinación de predicciones de modelos heterogéneos.

En la sección anterior se pudo comprobar cómo variar la forma en la que se utilizan los datos de entrenamiento puede mejorar el rendimiento de los modelos *machine learning*. Para ello, con un mismo modelo entrenado con diferentes subconjuntos de datos formábamos un *ensemble* y combinábamos sus predicciones mediante técnicas como la media aritmética o ponderada de las mismas o votación de las predicciones obtenidas.

Otra variante de las técnicas de ensemble consiste en utilizar en el mismo modelos de diferente estructura, que pueden haber sido entrenados o no con los mismos conjuntos de datos, para posteriormente combinar sus predicciones y mejorar la precisión del conjunto. En los párrafos siguientes se presentan tres aproximaciones diferentes de construir un *ensemble* con modelos heterogéneos.

- **Ensemble mediante la combinación de las predicciones de los modelos individuales**. Los pasos a seguir usando esta estrategia son los siguientes:
    - En primer lugar, debemos tener salvados a disco los diferentes modelos que queremos que participen en el ensemble. Estos modelos pueden haber sido entrenados con conjuntos de datos diferentes, pudiéndose dar el caso como se verá en secciones posteriores que hayan sido entrenados utilizando técnicas como *transfer learning*. Sin embargo, es muy importante tener en cuenta para que el ensemble sea efectivo, que debemos haber

guardado previamente no solo la configuración o estructura de los modelos, sino también los pesos calculados durante su entrenamiento, así como todos los objetos necesarios que fueron utilizados para la codificación de las entradas durante su entrenamiento. Si estos elementos no fueron guardados, no se podrán realizar las codificaciones de forma adecuada para poder entrenar o hacer predicciones en el ensemble. El módulo *pickle* de Python permite la serialización y salvado a disco de objetos. Resulta especialmente útil en la construcción de *ensembles* para guardar aquellas construcciones necesarias para la codificación de los conjuntos de datos como pueden ser los *Tokenizers* utilizados. Para salvar la estructura de los modelos existen dos alternativas posibles: utilizar el método *save* del objeto *Model* de la librería *keras* (Keras API, n.d.), o bien salvar la estructura a formato *JSON* haciendo uso de los metodos *to_json* o *model_from_json* de la propia librería. En lo que refiere a los pesos obtenidos durante el entrenamiento de los modelos, se pueden salvar y recuperar junto con el modelo usando los métodos *save* o *load* del objeto *Model*, o bien se pueden salvar y recuperar por separado haciendo uso de los métodos *load_weights* y *save_weights*. Es importante anotar que puede haber conflictos con las versiones de las librerías a la hora de salvar y cargar los modelos. Esto es principalmente notorio cuando se usan en el *ensemble* modelos que han sido entrenados en diferentes máquinas (cosa bastante habitual cuando se trata de modelos que consumen gran cantidad de recursos de computación durante su entrenamiento).

- Una vez se han recuperado los modelos que forman el ensemble se pueden combinar las predicciones de ambos utilizando algunas de las técnicas vistas en la subsección 6.1.1:
  - Media aritmética de las predicciones de los modelos del ensemble.
  - *Voting* o votación entre las predicciones de los modelos del ensemble.
  - Media aritmética ponderada de las predicciones de los modelos del ensemble. Aparte de las técnicas ya vistas para calcular los pesos (mediante algoritmos de optimización o fuerza bruta) hay que añadir una tercera manera basada en la experiencia previa. Analizando los resultados previos de los modelos que componen el *ensemble* se pueden asignar pesos de forma manual, siempre que se respete la regla de que la suma de los pesos sea uno.

  En el apéndice de códigos A.19 se encuentra el programa Python utilizado en la sección posterior para realizar los experimentos de *ensemble* mediante la combinación de las predicciones de los modelos individuales, utilizando diferentes variantes como media aritmética, *voting* o media ponderada (mediante algoritmos de optimización, fuerza bruta o asignando pesos de forma manual).

- **Stacked Ensemble o Stacking**. Las técnicas vistas en el ítem anterior (media o media ponderada) combinan las predicciones del *ensemble*

de una forma *lineal* dando pesos a las predicciones de forma proporcional al rendimiento que expresaban en un conjunto de test. Otra alternativa consiste en conectar las salidas de los modelos que forman el *ensemble* a una nueva red neuronal (habitualmente conocida como *meta-learner*), encargada de aprender la mejor manera de combinar las predicciones de los diferentes modelos, como propone la técnica de *stacked generalization* (Wolpert, 1992). Existen dos aproximaciones diferentes a la hora de realizar *stacking*:

- **Realizar el entrenamiento del *meta-learner* sin estar unido al resto de modelos del *ensemble***. Para ellos deben seguirse los siguientes pasos:
  - Cargamos los N modelos ya entrenados que componen el *ensemble*.
  - Preparamos un conjunto de entrenamiento para la red neuronal anexa encargada de aprender la mejor manera de combinar las predicciones (*meta-learner*). Este conjunto de entrenamiento puede ser el resultante de una combinación de las muestras usadas como test en los modelos ya entrenados con sus correspondientes etiquetas, y así evitar *overfitting*. Supongamos que obtenemos un subconjunto de M muestras. Debemos tener en cuenta que cada uno de los modelos individuales entrenados producen una salida de un número real comprendido entre cero y uno (nuestro caso de estudio se basa en la clasificación binaria de comentarios de turistas). El nuevo conjunto de entrenamiento será entonces el formado por M muestras consistentes en un

vector de N componentes reales (las N predicciones de los modelos), con sus correspondientes etiquetas. Esto querrá decir, que la entrada a la red *meta-learner* será un tensor de MxN dimensiones.

- Construido el modelo *meta-learner* y su correspondiente *training set* se procederá su entrenamiento.

Una vez se ha entrenado el *meta-learner*, para poder realizar predicciones con el *ensemble* se deben realizar los siguientes pasos: por un lado debemos codificar el comentario de forma individual para cada uno de los modelos del *ensemble*, utilizando la técnica de codificación y el *tokenizer* correspondiente en cada caso. Posteriormente construir el tensor con las predicciones de los modelos que servirá como entrada a la red *meta-learner* que será la que finalmente de como salida la predicción total del ensemble. La figura 6.2 muestra de forma esquemática cómo

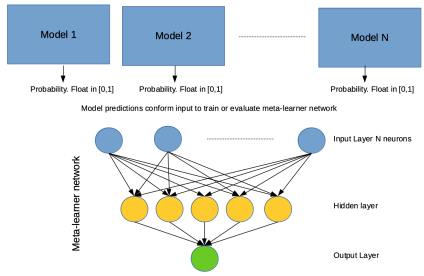


Figura 6.2. Representación de Stacking con meta-learner desconexo.

implementar un *stacking ensemble* en el que el *meta-learner* se encuentra desconectado del conjunto de modelos del ensemble,

utilizando las predicciones de los anteriores para entrenarse o realizar predicciones conjuntas. El código incluido en el anexo de códigos A.20 contiene el programa Python utilizado para realizar los experimentos de la sección posterior.

- ***stacking ensemble* en una única red neuronal**. En esta aproximación cada uno de los modelos ya entrenados se concatenan en una cabecera. La salida de ellos se conectará a la red *meta-learner*. En esta aproximación no tenemos que realizar un paso intermedio que es generar el conjunto de datos de entrenamiento del *meta-learner*, ya que al tener conectado todo el *ensemble* en una única red podemos entrenarlo todo junto. La principal desventaja es que realizar el montaje de todo el ensemble conjunto trae consigo una serie de inconvenientes:
  - Al tratarse del ensamblaje de un conjunto de modelos independientes, la entrada del modelo global (el *ensemble*) debe ser conformada mediante la concatenación de las entradas de todos los modelos. En nuestro caso de estudio, esto significa, que un comentario de un turista deberá codificarse utilizando las técnicas individuales de cada uno de los modelos de la cabecera, teniendo en cuenta diferentes aspectos como la forma en la que se ha hecho el relleno de las secuencias (*padding*), como la forma en el que se han truncado las mismas, o el *Tokenizador* utilizado para formarlas.
  - Los modelos pueden venir entrenados o no con sus respectivos pesos. En caso de trabajar con modelos que han

requerido de una alta carga computacional probablemente han sido entrenados en máquinas diferentes, por lo que la probabilidad de que haya discrepancias en las librerías usadas para su construcción y entrenamiento aumente.

- Este tipo de estructuras permite seleccionar aquellas partes de la misma que requieren de ser entrenadas nuevamente en conjunto, pero requiere de un conocimiento amplio de los modelos utilizados en el *ensemble*, para no emplear más tiempo de computación en el caso de que se hayan usado capas entrenadas con conjuntos de datos diferentes (como pueden ser capas de *embedding*.

Afortunadamente la librería *Keras* (Keras API, n.d.) nos ofrece métodos para poder construir la entrada conjunta de todos los modelos de cabecera, o concatenar sus salidas para conectarlas de forma adecuada en forma de capa de red neuronal al *meta-learner*. Los atributos de los modelos y sus capas nos permitirán también cambiar algunas características, como por ejemplo marcarlas como *no entrenables* o modificar su nombre para que se único en la red global (requerimiento para que pueda ser compilado). El código mostrado en el apéndice de códigos A.21 muestra el programa Python utilizado para los experimentos de la sección posterior. En él se puede ver cómo se construye el modelo por capas. La figura 6.3 muestra de forma esquemática un *stacking ensemble* de modelos heterogéneos conectados a la misma red *meta-learner* lo que permitiría un entrenamiento conjunto o separado.

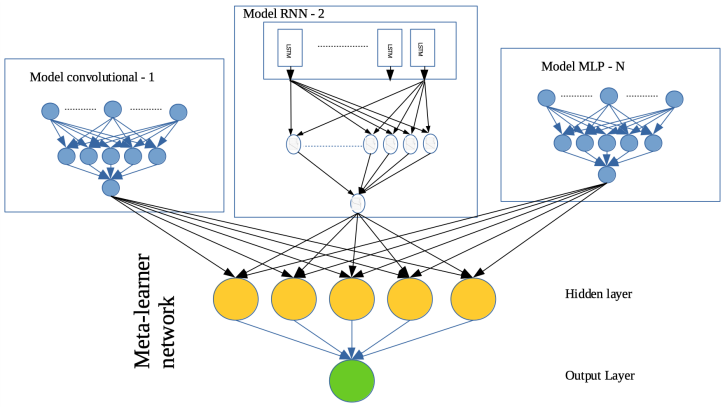


Figura 6.3. Representación de Stacking con modelos de cabecera integrados en la red.

6.2. Experimentos y resultados con Ensembles

En esta sección se muestran los experimentos realizados con *ensembles* basados en los modelos más prometedores usados en capítulos anteriores.

En lo que refiere a las técnicas de *ensemble* basadas en la modificación de los datos de entrenamiento para reducir la varianza, vimos en la sección 6.1.1 cómo la utilización de *random splits*, *k-fold cross validation*, o *bagging* conseguían un aumento en precisión de hasta 5 % en una red básica perceptrón multicapa.

En el capítulo 4 se presentaron diferentes alternativas de modelos *deep learning* para el problema en estudio (la clasificación de opiniones de turistas). En concreto, el modelo presentado en la figura 4.9 formado por dos

capas convolucionales con 64 filtros cada una, tamaño de *kernel* 2, y capas de *maxpooling* siguiendo a las convolucionales con un tamaño de ventana 2, obtuvo un excelente resultado de un 92,14 % de precisión para un conjunto de datos de entrenamiento relativamente reducido (9.640 muestras balanceadas). Para comprobar el rendimiento de este mismo modelo aplicando las técnicas de *ensemble* de variación de datos anteriores usamos el programa Python listado en el apéndice de códigos como A.18, obteniendo los resultados mostrados en la tabla 6.2. Como puede comprobarse en la misma, el mejor resultado se obtiene aplicando la técnica de *Random Splits* usando el 10 % de las muestras totales como test. El resultado obtenido fue de un **92,78 %** lo que supone un incremento en precisión del 0,64 % a un modelo que ya obtuvo muy buenos resultados.

Tabla 6.2. Comparativa técnicas de ensemble variando los datos de entrenamiento con un modelo de dos capas convolucionales. Figura 4.9

Codification	"Good"hits	"Bad"hits	"Good"fails	"Bad"fails	% Accu
Rand-splits. Mean.	1.279	1.305	72	129	92,78
Rand-splits. W-Mean.	1.280	1.296	81	128	92,50
Rand-splits. Voting.	1.266	1.310	67	142	92,50
K-fold. Mean.	1.275	1.302	75	133	92,53
K-fold. W-Mean.	1.245	1.304	73	163	91,53
K-fold. Voting.	1.262	1.312	65	146	92,42
Bagging. Mean.	1.254	1.308	69	154	91,99
Bagging. W-Mean.	1.257	1.283	94	151	91,20
Bagging. Voting.	1.238	1.314	63	170	91,63

En la figura 6.4 se muestra de forma gráfica los resultados obtenidos.

En lo referente a las técnicas de *ensemble* variando la arquitectura se usaron dos conjuntos de modelos diferentes. El primero de los conjuntos estaba formado por modelos entrenados únicamente por un conjunto de datos reducidos, los obtenidos de la extracción realizada en este trabajo

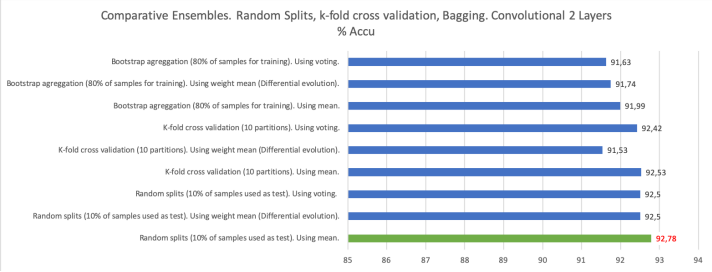


Figura 6.4. Comparativa técnicas de ensemble variando los datos de entrenamiento con un modelo de dos capas convolucionales. Figura 4.9

de investigación de los portales de *Tripadvisor* y *Booking* (aunque se usó parcialmente la técnica de *transfer learning* para la codificación *embedding* de la primera capa). El segundo de los conjuntos añadía al primero dos nuevos modelos entrenados con un conjunto de datos más amplios usando técnicas de *transfer learning* vistas en el capítulo 5.

El primer *ensemble* (entrenado con el conjunto de datos reducido) estaba formado por los siguientes modelos:

1. Un modelo formado por las siguientes capas:

    - Una capa de embedding entrenada junto con el resto de la red para codificar secuencias de 800 componentes en vectores de 300 dimensiones. Para la codificación de las secuencias se usó *padding* y truncado por el final de la misma (*post padding*).

    - Una capa convolucional de 64 filtros, con un un tamaño de *kernel* de 2.

- Una capa Maxpool con tamaño de ventana de 2.
- Otra capa convolucional de 64 filtros, con un un tamaño de *kernel* de 2.
- Otra capa Maxpool con tamaño de ventana de 2.
- Una capa de aplanado.
- Una capa densa de 50 neuronas con función de activación *relu*.
- Una capa densa de salida de una neurona con función de activación *sigmoide*.

2. Un modelo igual, que el anterior, pero sustituyendo la capa de **embedding** a entrenar junto con el modelo, por una capa de *embedding* entrenada previamente (*Google Word2Vec*) que igualmente usa secuencias de entrada de 800 componentes con *post padding* y truncado por el final de la misma y genera vectores de 300 dimensiones.
3. Un modelo formado por las siguientes capas:
   - Una capa de *embedding* entrenada previamente (*Google Word2Vec*) que usa secuencias de entrada de 800 componentes con *post padding* y truncado por el final de la misma y genera vectores de 300 dimensiones.
   - Una capa convolucional de 64 filtros, con un un tamaño de *kernel* de 2.
   - Una capa Maxpool con tamaño de ventana de 2.

- Una capa de aplanado.
- Una capa densa de 50 neuronas con función de activación *relu*.
- Una capa densa de salida de una neurona con función de activación *sigmoide*.

4. Un modelo formado por las siguientes capas:
   - Una capa de embedding entrenada junto con el resto de la red para codificar secuencias de 800 componentes en vectores de 300 dimensiones. Para la codificación de las secuencias se usó *padding* y truncado por el comienzo de la misma (*pre padding*). Es importante hacer notar que la técnica de truncado y de *padding* en los modelos que usan capas *LSTM* cambia con respecto al resto, ya que como se explicó en el capítulo 4 el *post padding* provocaba un efecto de olvido en la misma. Esta diferencia provocó que la codificación de secuencias no fuera uniforme a la hora de formar las entradas de las cabeceras.
   - Una capa de 50 celdas LSTM.
   - Una capa densa de 50 neuronas con función de activación *relu*.
   - Una capa densa de salida de una neurona con función de activación *sigmoide*.
5. Un modelo igual, que el anterior, pero sustituyendo la capa de **embedding** a entrenar junto con el modelo, por una capa de *embedding* entrenada previamente (*Google Word2Vec*) que igualmente usa secuencias de entrada de 800 componentes con *pre padding* y truncado

por el comienzo de la misma y genera vectores de 300 dimensiones.

La aplicación de las diferentes técnicas de *ensemble* con modelos heterogéneos se llevó a cabo con los siguientes códigos:

- Para calcular la predicción total del *ensemble* mediante la combinación de las predicciones individuales usando media aritmética, media aritmética ponderada, y *voting* se usó el código incluido en el apéndice de códigos A.19.
- Para implementar la técnica de *stacking*, separando los modelos individuales para generar un conjunto de datos de entrenamientos de predicciones que formen la entrada de un una red neuronal *meta-learner*, se usó el código incluido en el apéndice de códigos A.20. Como red neuronal *meta-learner* se usó una red perceptrón multicapa con una capa oculta de 30 neuronas y función de activación *relu* y una capa de salida con una única neurona y función de activación *sigmoide*.
- Para implementar la técnica de *stacking* integrando los modelos individuales en la cabecera al *meta-learner* formado por una red neuronal perceptrón multicapa se usó el código incluido en el apéndice de códigos A.21. La representación gráfica generada por el propio código se muestra en la figura 6.5.

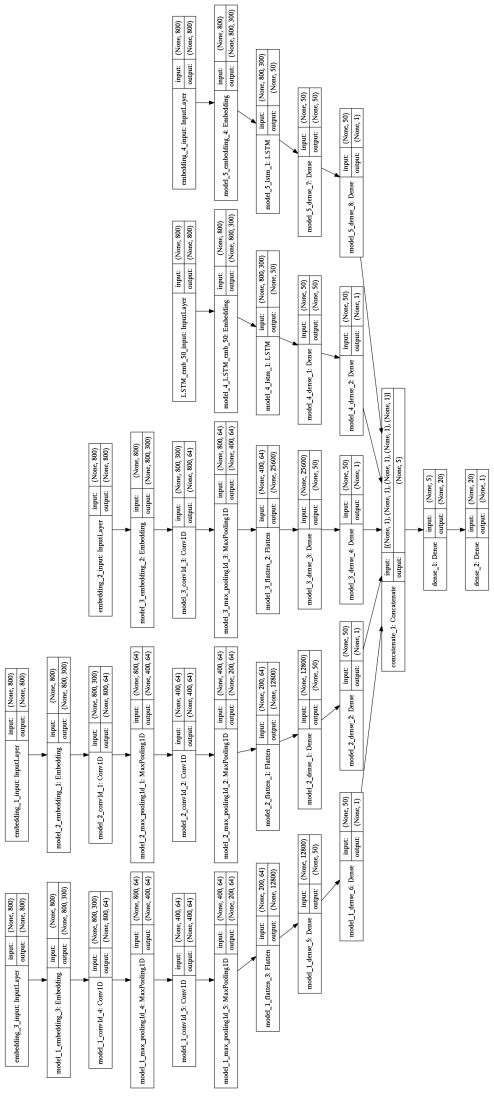


Figura 6.5. Stacking Ensemble formado por 5 modelos unidos a una red MLP.

La tabla 6.3 muestra los resultados obtenidos al aplicar las técnicas de *ensemble* vistas en este capítulo sobre modelos usados en el capítulo 4 que fueron entrenados con un conjunto de datos reducido. La figura 6.6 muestra los mismos resultados de forma gráfica.

Tabla 6.3. Comparativa técnicas de ensemble usando 5 modelos entrenados con un conjunto de datos reducido.

Codification	% Accu
1.- Ensemble. Using mean.	93,18
2.- Ensemble . Using weight mean (manual weights).	93,32
3.- Ensemble. Using voting.	93,36
4.- Ensemble. Separated Stacking + MLP 30 neurons.	92,39
5.- Ensemble. Integrated Stacking. No re-trained.	92,06
6.- Ensemble. Integrated Stacking. Re-trained	91,63

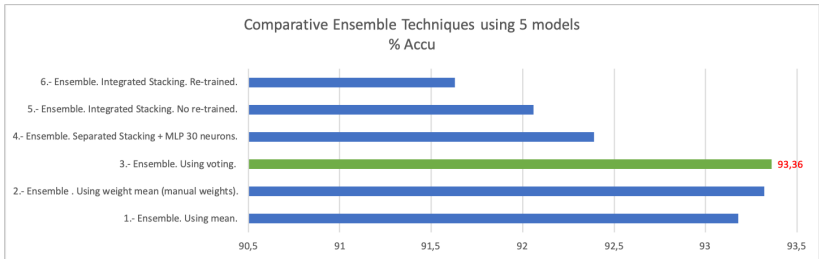


Figura 6.6. Comparativa técnicas de ensemble usando 5 modelos entrenados con conjunto de datos reducido.

Como se puede comprobar en la tabla, el mejor resultado se obtuvo utilizando un sistema de votación (*voting*) entre los modelos del *ensemble* logrando un **93,36 %** de precisión, lo que supone una mejora del 1,22 % sobre el mejor resultado en precisión de los modelos tomados de forma individual.

Para completar los experimentos se realizó un segundo *ensemble* que incluía los modelos individuales descritos anteriormente y dos nuevos modelos que fueron entrenados con un conjunto amplio de muestras utilizando técnicas de *transfer learning*. Estos dos nuevos modelos incluidos tenían las siguientes características:

- Un modelo con una capa de *embedding* entrenada previamente mediante *Google Word2Vec*, con dos capas convolucionales unidas a dos capas *maxpool* mas una densa de 50 neuronas y una de salida (igual en estructura al primer de los modelos del ensemble anterior). Entrenada con el conjunto de datos Yelp Dataset (YelpDataset, n.d.) con 3.6 GB de comentarios de clientes de diferentes negocios de restauración. Este modelo, presentado en el capítulo 5 obtuvo una excelente precisión del 95,22 %.

- Un modelo igual que al anterior, pero con una capa de *embedding* entrenada junto con el resto de la red utilizando un conjunto de comentarios de clientes de Amazon con más de treinta y cinco millones de comentarios (Zhang, n.d.). Este modelo, presentado en el capítulo 5 obtuvo una excelente precisión del 92,57 %.

La figura 6.7 muestra el *Stacking Ensemble* formado por los 7 modelos comentados anteriormente.

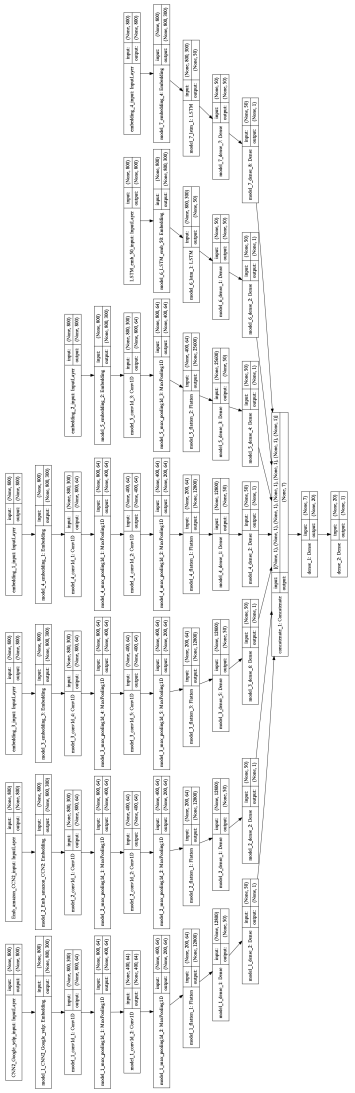


Figura 6.7. Stacking Ensemble formado por 7 modelos unidos a una red MLP.

Los códigos utilizados para realizar los experimentos con este nuevo ensemble de 7 modelos fueron los mismos que los usados anteriormente, ya que estaban generalizados para cargar nuevos modelos de disco, especificando los objetos necesarios para la codificación de sus secuencias de entrada.

Los resultados obtenidos se muestran en la tabla 6.4 y de forma gráfica en la figura 6.8.

Tabla 6.4. Comparativa técnicas de ensemble usando 7 modelos.

Codification	% Accu
1.- Ensemble. Using mean.	94,08
2.- Ensemble . Using weight mean (manual weights).	95,37
3.- Ensemble. Using voting.	94,47
4.- Ensemble. Separated Stacking + MLP 30 neurons.	91,63
5.- Ensemble. Integrated Stacking. No re-trained.	92,85
6.- Ensemble. Integrated Stacking. Re-trained	92,06

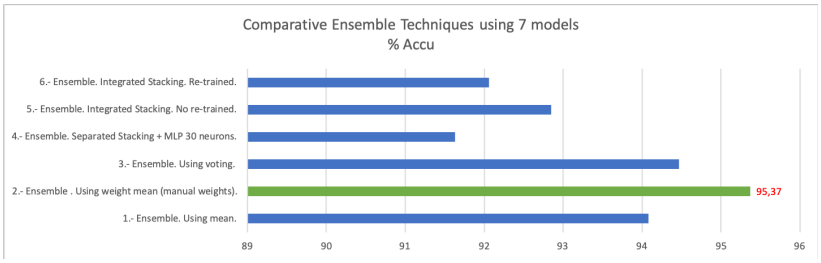


Figura 6.8. Comparativa técnicas de ensemble usando 7 modelos.

Como puede observarse, el margen de mejora en la precisión del modelo entrenado con técnicas de *transfer learning* usando el conjunto de datos de entrenamiento de *Yelp* era escaso. Aún así se pudo mejorar con un ensemble de media ponderada utilizando pesos manuales logrando una

precisión del **95,37 %**, lo que supone un aumento de un 0,15 %.

Las conclusiones obtenidas en este trabajo de investigación, en lo referente a la aplicación de técnicas de *ensemble* se pueden resumir en:

- De forma general, las técnicas de *ensemble* mejoran la precisión de los modelos individuales que lo componen, reduciendo su varianza y su sesgo.
- Cuando el conjunto de datos de entrenamiento es reducido, y la precisión del modelo tiene margen de mejora, usar técnicas de *ensemble* mediante la variación de sus datos de entrenamiento mejora de forma considerable los resultados. En el ejemplo mostrado con una red perceptrón multicapa cuyos resultados se presentan en la tabla 6.1 se observó que se mejoró en un 5,38 %.
- Cuando el conjunto de datos de entrenamiento es reducido, pero la precisión del modelo individual ya es bastante buena, la aplicación de técnicas de *ensemble* mediante la variación de datos no aumenta de forma considerable, pero aún así se consiguió una nueva mejora logrando una precisión del 92,78 %.
- Cuando el conjunto de datos es reducido, pero utilizando técnicas de *ensemble* que incluye modelos heterogéneos se obtuvo un excelente resultado mediante el sistema de votación de las predicciones individuales, como se mostró en la tabla 6.3, obteniendo una precisión del 93,36 %. La aplicación de esta técnica debe tenerse en cuenta para problemas en los que no tenemos demasiadas muestras de entrenamiento o disponemos de tiempo reducido de computación

para realizar los entrenamientos.

- La aplicación de las técnicas vistas para modelo heterogéneos requieren tener en consideración la codificación utilizada en cada uno de los modelos utilizados en el *ensemble*. Para ello es necesario guardar en archivos la estructura del modelo, los pesos, y los objetos necesarios para realizar la codificación en secuencias de sus entradas (sus *tokenizers* o incluso el método por el cual se realizó el *padding*). Los modelos utilizados en un los *ensembles* consumieron un gran número de horas de computación en diferentes máquinas (superando en algunas ocasiones el mes y medio de entrenamiento). Es muy importante mantener de forma homogénea las versiones de Python y librerías adicionales en todas las máquinas utilizadas.

- Utilizar *ensembles* con modelos entrenados anteriormente (especialmente aquellos más complejos) requieren un gran espacio de memoria para almacenar las matrices con los pesos. La figura 6.7 muestra la complejidad de la estructura para un *ensemble* de 7 modelos.

- En los experimentos realizados con *Stacking Ensembles Integrados* no se observó ninguna mejora en realizar un segundo entrenamiento de todos los modelos. Esto es debido a que algunos de ellos ya utilizaron estas muestras en su entrenamiento, y para los dos que utilizaron técnicas de *transfer learning* el número de muestras utilizadas en su entrenamiento individual eran proporcionalmente mucho mayor. Por este motivo, para *ensembles* que contienen modelos individuales entrenados con muestras amplias y que ya han obtenido excelentes resultados no compensa realizar un nuevo entrenamiento y es mejor definir las capas de los mismos como no entrenables.

- Para concluir, aunque se usaron modelos individuales en el *ensemble* que ya obtuvieron excelentes resultados, se pudo mejorar la precisión en conjunto, obteniendo un máximo de **95,37 %**.

En este capítulo se han mostrado las técnicas de *ensemble* más habituales y su aplicación al problema en estudio (la clasificación de opiniones de turistas en el sector hotelero). Se han utilizado en el mismo los modelos ya entrenados en capítulos anteriores, mejorando el rendimiento de los mismos. La aplicación completa de estas técnicas también ha mostrado que para problemas con reducido conjunto de datos de entrenamiento el margen de mejora es grande en relación a la utilización de modelos individuales.

Conclusiones

La industria del turismo tiene una importancia vital en la economía de las Islas Canarias. En la actualidad su aportación al PIB es del 35 % y supone casi el 40 % de los empleos, al recibir anualmente millones de turistas.

Para poder mantener, o incluso mejorar la calidad de los servicios que ofrecemos a nuestros visitantes, es muy importante conocer la opinión que tienen de los mismos. Los métodos tradicionales de captación de opiniones (como pueden ser las encuestas de satisfacción) son excesivamente dirigidas, y no suelen ser muy bien aceptadas por los clientes.

El uso masivo de internet ha introducido una gran oportunidad para todos los mercados, incluido el turismo. Los consumidores expresan su opinión a través de RRSS y portales especializados de forma libre y espontánea. Las empresas ya han empezado a utilizar esta información de forma masiva, pero en muchas ocasiones no resulta sencillo por dos razones fundamentales:

- El volumen de información es elevado y se encuentra disperso en

diversas plataformas.

- Los datos no suelen estar estructurados, o si tienen estructura esta suele ser muy cambiante para poderse adaptar a las nuevas tecnología, por lo que su acceso suele ser complejo.

Los motivos anteriores hacen necesario el uso de técnicas automatizadas para su obtención y procesamiento. Una de las disciplinas más en auge actualmente dentro de la inteligencia artificial IA, el procesamiento del lenguaje natural PLN, incluye técnicas que podemos aplicar para resolver estos problemas. En concreto, el problema en estudio de este trabajo de investigación, la clasificación de las opiniones escritas en RRSS por parte de los turistas que visitan nuestras islas, estaría incluida dentro de lo que conocemos como *análisis de sentimientos*.

Las técnicas que hemos utilizado están clasificadas dentro del *aprendizaje supervisado*. Para poder enseñar al algoritmo o entrenar el modelo se utiliza un conjunto de datos (conocidos como datos de entrenamiento) que han sido previamente etiquetados (clasificados). El hecho de usar muestras ya clasificadas da nombre a este tipo de aprendizaje máquina ML, ya que se considera que ha sido *supervisado* previamente por un humano. Para ello se ha realizado un análisis exhaustivo de diferentes modelos de aprendizaje profundo DL con los que hemos obtenidos buenos resultados de precisión a la hora de clasificar las opiniones como buenas o malas.

La extracción de los datos de entrenamiento y prueba fue realizada de

dos portales especializados: *Booking* y *Tripadvisor*, realizando una búsqueda de opiniones de usuarios cuyo destino de vacaciones hubiera sido las Islas Canarias. Se desarrollaron programas en Python para la extraccción y procesamiento, y como resultado de los mismos se obtuvieron tres conjuntos diferentes de datos (que denominamos *data sets*):

- Un primer conjunto de muestras de entrenamiento *training data set*. Con muestras bien balanceadas (igual número de comentarios positivos que negativos).
- Un segundo conjunto más reducido de prueba *test data set*, para hacer una valoración de la técnica.
- Un tercer conjunto independiente para valorar más detalladamente los resultados de los diferentes experimentos *evaluation data set*.

Todos los conjuntos de datos se guardaron en ficheros con formato Comma Separated Values (CSV).

Antes de poder aplicar técnicas DL de clasificación al dominio del problema en estudio (opiniones escritas en inglés por parte de nuestros turistas), es necesario realizar una transformación de las mismas en vectores numéricos. A lo largo del capítulo 3 se analizaron diversas técnicas de codificación de las cuales pudimos obtener los siguientes resultados:

- La aplicación de la técnica BoW, utlizada con un ejemplo de red neuronal simple del tipo Multilayer Perceptron (MLP) con una única capa oculta de 50 neuronas tuvo una precisión máxima del 91,1 %. Además se da circunstancia de que este resultado se obtiene sin realizar un procesamiento previo de los comentarios de los turistas (sin

eliminar *stopwords*). Es además concluyente, que la elección del tamaño de las secuencias, y la decisión de realizar procesamiento previo influyen de forma significativa, ya que la diferencia entre el mejor de los resultados de la técnica y el peor es de un 13,33 % de precisión.

- En lo que refiere al tamaño de la entrada (que para la técnica BoW hace referencia al número de palabras a considerar en el vocabulario total de todas las muestras), de forma general, cuantas mayor era el tamaño de la entrada, mejor se comportaba la codificación, pero a partir del tamaño 1.600 no se obtuvo ninguna mejora.
- En la aplicación de la técnica de codificación de *Embedding* se utilizaron diferentes variantes y las aplicamos a la misma red MLP utilizada con BoW, haciendo variar el tamaño de las secuencias de entrada:
  - Mediante el uso de la técnica W2V.
  - Usando un embedding pre-entrenado *Googgle W2V*.
  - Usando un embedding pre-entrenado *GloVe*.
  - Entrenando una capa de embedding junto al resto del modelo

  El mejor resultado en la aplicación de esta técnica se obtuvo cuando se entrenó la capa de embedding junto con la red MLP al utilizar secuencias de tamaño 400 y vectores de 25 componentes. La precisión obtenida fe en esta ocasión del 91,85 %, y nuevamente volvió a ser significativo la diferencia con el modelo que obtuvo la peor

precisión.

- En lo que refiere al tamaño de las secuencias elegidas en la técnica de embedding (en esta ocasión hace referencia al tamaño de los comentarios de los turistas), en casi todos los experimentos los mejores resultados se obtuvieron cuando se ajustaban a un tamaño de secuencia de 800.
- Como principal conclusión en el impacto que tienen las técnicas de codificación elegidas en la clasificación de opiniones es importante destacar que:
  - Es importante realizar un estudio previo del comportamiento de diferentes técnicas de codificación, ya que la elección puede influir de forma significativa en los resultados de precisión.
  - Igualmente aporta un valor significativo experimentar con diferentes tamaños de secuencias o vectores de embedding. Los resultados en la codificación dependen en gran medida de estos valores, y estos a su vez, están fuertemente ligados al dominio. En nuestro caso de estudio los comentarios de los turistas tenían un tamaño variable, pero acotados en un rango no muy extenso. Si se eligieran otras fuentes de datos diferentes a las usadas sería necesario realizar algunas pruebas de rendimiento en la codificación para seleccionar las más prometedoras a usar posteriormente en experimentos que requieran un alto coste de entrenamiento.

Una vez realizado el análisis de impacto en la elección de la técnica

de codificación en el dominio de nuestro problema, en el capítulo 4 se realizó un estudio de las técnicas más usadas actualmente en problemas de clasificación en el ámbito del análisis de sentimiento, como son el uso de SVM o de técnicas de DL, concretamente redes del tipo CNN y LSTM. Las conclusiones a extraer de este estudio fueron las siguientes:

- La aplicación del algoritmo SVM en su variante *nu-SVC* tuvo un resultado en precisión del 92,57 %, teniendo un comportamiento más estable cuando usamos un kernel lineal o un kernel sigmoide. Las funciones kernel lineal son muy adecuadas en dominios con muchas características (como es el caso de los textos), y son menos costosas en computación, por lo que se convierte en el kernel más adecuado para nuestro caso de estudio.

- Para medir la eficacia utilizar capas convolucionales se realizó un experimento con una red CNN que añadía una capa convolucional (con 64 filtros y kernel 3) y maxpooling (de windows size 2)al modelo simple MLP, utilizando ambas el mismo tipo de codificación (en este caso fue Google W2V). El resultado obtuvo una precisión del 91,1 %. Comparando este modelo con el mismo sin aplicar la operación de convolución hubo una diferencia de 4,48 % de precisión, por lo que pudimos comprobar que efectivamente la convolución mejoraba las clasificaciones.

- Al añadir una segunda operación de convolución, con una nueva convolución y maxpooling con las mismas características que las mencionadas anteriormente se volvió a mejorar el resultado en precisión, obteniendo un 92,14 %.

- Se estudió el comportamiento de modelos tipo LSTM con el mismo tipo de codificación usado anteriormente (Google W2V), y usando junto a ella la misma red MLP. Como se comentó en el capítulo 4 es importante cambiar el *padding* de las secuencias al utilizar este tipo de redes, ya que si se usa relleno con ceros al final de las mismas aparece un efecto de olvido que impide su entrenamiento. La mejor precisión para este tipo de redes se logró al utilizar 50 celdas, con un valor del 91,1 %.

- Tanto en los modelos CNN como LSTM, el comportamiento al usar embedding pre-entrenados fue mejor. Esta hecho no ocurrió cuando se usó únicamente la red MLP, lo que induce a pensar que la representación vectorial obtenida por los embeddings pre-entrenados es más adecuada para poder extraer características o detectar series, lo cual parece lógico al utilizar un número de muestras mucho mayor.

- Para comprobar si las redes LSTM mejoraban el comportamiento utilizando previamente una capa convolucional, se experimentó añadiendo parejas de capas convolucional - maxpooling al modelo anterior, y se obtuvo una ligera mejora que en ninguno de los casos mejoró el resultado de precisión al usar únicamente el modelo equivalente CNN.

- Las principales conclusiones obtenidas en el estudio de estas técnicas ML al dominio de nuestro problema fueron las siguiente:

    - El algoritmo SVM tiene unos resultados excelentes cuando el número de muestras es bajo (como en nuestro caso de estudio que son 9.640 muestras de entrenamiento), superando ligera-

mente a los modelos DL probados.

- Las redes CNN tienen igualmente muy buen comportamiento aún cuando el conjunto de entrenamiento es reducido. Es importante reseñar que todas las técnicas de DL requieren de conjuntos de entrenamiento grandes para obtener buenos resultados.
- Las técnicas CNN y LSTM potenciaron su precisión al utilizar embeddings pre-entrenados, por lo que podemos concluir que favorecen la extracción de características y la detección de series.
- De forma general, todos los modelos obtienen mejores resultado en la predicción de muestras negativas. Este resultado es habitual en clasificación de textos debido a que los términos negativos aportan mayor carga de significado.

Como ya se ha comentado, las técnicas DL suelen requerir de un número mayor de nuestras de entrenamiento que el que hemos usado en la aplicación de nuestro problema. Por este motivo, en el capítulo 5 se utilizaron técnicas *Transfer Learning* para investigar la mejora que producen en su aplicación.

En concreto, de las técnicas expuestas en el capítulo 5 se aplicaron las siguientes:

- *Extracción de características*. Esta técnica ya fue aplicada en el momento que utilizaron embeddings pre-entrenados como los comentados anteriormente.

- *Uso de datos de otros problemas*. En esta ocasión se probó con conjuntos de entrenamientos externos:
  - IMDB. Críticas de películas.
  - Yelp. Opiniones de usuarios de servicios de restauración.
  - Amazon.Comentarios sobre productos.

Para poder realizar el entrenamiento de los modelos con conjuntos de datos tan grandes fue necesario utilizar máquinas de alto rendimiento con 376 GB de memoria con 80 núcleos. Debido a que el tiempo de entrenamiento era del orden de semanas se aplicó como estrategia realizar pruebas con los conjuntos de entrenamiento más reducidos para seleccionar los modelos más prometedores.

En relación con la aplicación de técnicas *transfer learning* a nuestro caso de estudio pudimos obtener las siguientes conclusiones:

- Las técnicas de *distant learning* son especialmente útiles para construir conjuntos de datos de entrenamiento etiquetados a partir de datos no etiquetados. También son ampliamente utilizadas para etiquetar diferentes aspectos de un mismo comentario. En este trabajo de investigación no fueron utilizadas porque se disponía de conjuntos de datos de entrenamiento externos correctamente etiquetados.
- La aplicación de técnicas *transfer learning* no mejoran los resultados en todos los experimentos. Es importante conocer el dominio de aplicación, es decir, para que tarea se está transfiriendo el conocimiento y cuál es la información utilizada en la misma. Además, si

tenemos en cuenta los tiempos de entrenamiento cuando el conjunto de datos es grande, se debe seguir una estrategia ordenada para poder detectar el camino más prometedor.

- Tras seleccionar el conocimiento a transferir se obtuvo una precisión del **95,2 %**, utilizando el conjunto de entrenamientos *Yelp* con la codificación embedding de *Google W2V*. Para nuestro caso de estudio, el conjunto de datos de entrenamiento seleccionado (comentarios de usuarios de servicios de restauración) tiene bastante similitud con los comentarios de turistas sobre servicios hoteleros, por lo que el aprendizaje transferido tiene muy buenos resultados. La técnica de codificación, como ya se comentó, potencia además la extracción de características al usar un número mayor de muestras. El modelo que estuvo esta alta precisión fue el mencionado anteriormente (dos parejas de capas convolucionales - maxpooling, precedidas de una capa de embedding Google W2V, y unida una MLP con 50 neuronas en la capa oculta). En este caso, no mejoró realizar un segundo entrenamiento del modelo con las muestras del dominio original, ya que la proporción en número con el conjunto de *Yelp* era irrelevante.

Una vez solventado el problema que genera el disponer de un conjunto de datos de entrenamiento reducido mediante la aplicación de *transfer learning*, y obteniendo un nivel de precisión elevado en las predicciones, se procedió a dar un paso más en la mejora de los modelos. Las técnicas de *ensemble* están destinadas a reducir la alta varianza y el sesgo de los modelos DL, en los que siempre es complicado a priori cómo seleccionar los datos de entrenamiento o cuál de los modelos se ajusta mejor al dominio de estudio.

En el capítulo 6 se aplicaron diferentes formas de implantación de ensembles:

- Realizando variaciones en el conjunto de datos de entrenamiento que entrenan un mismo modelo.
- Variando los modelos que pertenecen al *ensemble*.
- Variando La forma en la que se combinan los resultados de los modelos.

Las conclusiones obtenidas en este trabajo de investigación sobre la aplicación de técnicas de *ensemble* en nuestro caso de estudio fueron:

- Cuando se aplicaron las técnicas que variaban el conjunto de datos de entrenamiento como *random splits*, *k-fold cross validation*, o *bagging* se consiguió hasta un aumento en la precisión del 5 % en una red MLP. Estos resultados concluyen que en aquellos casos en el que sea imposible aplicar una técnica de **transfer learning** para aumentar el conjunto de datos de entrenamiento, la aplicación de *ensembles* de este tipo es una alternativa muy interesante que debe tenerse en cuenta.
- La aplicación de *random splits* a un modelo de dos capas convolucionales logró incrementar la precisión del modelo obteniendo un 92,78 % de precisión, sin tener que recurrir a técnicas de *transfer learning*, superando así ligeramente el resultado obtenido en los experimentos de SVM.
- Aplicando la combinación de resultados mediante votación de un

ensemble formado por diferentes modelos, que fueron entrenados con un conjunto de datos reducido, se un obtuvo un 93,36 %, nuevamente sin tener que recurrir a técnicas de *transfer learning*. Este resultado, convirtió este experimento en la mejor alternativa a nuestro caso de estudio, sin usar datos de otros dominios.

- Cuando ampliamos el *ensemble* con dos modelos que fueron entrenados con *transfer learning* que obtenían ya grandes precisiones de forma individual, utilizando una combinación de resultados mediante media ponderada (ajustando los pesos de los resultados individuales de forma manual en base a la experiencia adquirida) se obtuvo el mejor de los resultados globales de todos los experimentos, con un 95,37 % de precisión.
- Las principales conclusiones obtenidas de la investigación sobre los métodos de *ensemble* aplicados fueron los siguientes:
  - Ante un problema en el que tenemos un único modelo DL, y el conjunto de datos es reducido es muy beneficioso aplicar técnicas de variación de datos como *random splits*, *k-fold cross validation*, o *bagging*.
  - En el caso de tener diferentes modelos DL entrenados con un conjunto de datos reducido, aplicar un sistema de combinación de resultados como media ponderada o votación mejora los resultados en precisión. En concreto, en nuestro caso de estudio, el sistema de votación se comportó adecuadamente, y consiguió una precisión elevada con un conjunto de datos reducido.

- En el caso de *ensembles* que contengan modelos afinados o enriquecidos con técnicas de *transfer learning* no se obtuvieron grandes mejoras, pero en cualquier caso la aplicación de las mismas mejoró ligeramente el resultado.

Todo lo dicho anteriormente son las principales conclusiones obtenidas al realizar una investigación y experimentación exhaustiva de diferentes técnicas de ML, y más concretamente de DL al problema de clasificación de opiniones expresadas por turistas que visitaron las Islas Canarias. La precisión obtenida en la clasificación de opiniones animan a aplicar las técnicas estudiadas en herramientas que ayuden a mantener o mejorar una industria tan importante en la economía de nuestras islas.

Líneas abiertas.

El área de investigación de ML y concretamente de DL es amplio y cambiante. En este trabajo se ha realizado un estudio profundo aplicado al problema de clasificación de opiniones de los turistas, pero aún quedan muchas cosas por explorar. A continuación se presentan algunas de ellas:

- El trabajo realizado se ha centrado en la clasificación de opiniones de los turistas con el objetivo de conocer si el comentario emitido se puede considerar bueno o malo. Las técnicas desarrolladas pueden ser incorporadas ya a diferentes herramientas de utilidad en la mejora del sector. La mas inmediata es la monitorización del grado de satisfacción de los turistas por zonas. Los portales especializados de los que hemos extraído los datos permiten ya realizar filtros en las

consultas para una región concreta. Realizando una monitorización básica podremos de una forma sencilla alertar de diferentes situaciones:

  - Una región con pocos comentarios totales y amplio porcentaje de comentarios buenos podría necesitar una mayor publicidad, por lo que se podría potenciar con campañas de marketing. Si por el contrario, tuviera un porcentaje de comentarios negativos ya sabríamos el motivo de su escaso éxito.

  - Una región con muchos comentarios totales y amplio porcentaje positivos indica que está bien gestionada y en principio solo requiere mantener el nivel de satisfacción. Si por el contrario, sus comentarios fueran negativos requiere revertir la tendencia mejorando sus servicios.

- Para mejorar la utilidad actual de los modelos generados sería muy conveniente realizar una detección de los diferentes aspectos a los que hace referencia el comentario. En un texto breve caben diferentes opiniones sobre diferentes aspectos, que el usuario no ha separado en dos comentarios. La detección de la polaridad de comentarios sobre aspectos diferentes de los servicios del sector turístico permitiría no solo un análisis de satisfacción por regiones, sino también por servicios.

- La ampliación a otras fuentes de información, en concreto aquellas RRSS en las que el modo de comunicación es diferente al utilizado en los portales especializados del sector permitiría mejorar la utilidad de las herramientas generadas:

- Filtrando tópicos concretos que permitan identificar lugares o atracciones turísticas, alertando a sus responsables de los resultados obtenidos.
- Localizando el origen de los comentarios. Podríamos analizar la polaridad no solo por el destino, sino también por el origen de los turistas. De esta forma se podría adecuar algunos de los servicios a los usos y costumbres de determinadas procedencias.

- Ampliar el análisis del sentimiento expresado en la opinión. En el capítulo 2 se hizo referencia a la construcción de tuplas de información a partir de los comentarios, que recojan diferentes características como la entidad, el aspecto, el sentimiento, el titular o el instante en el que se ha emitido. En los párrafos anteriores ya se ha resaltado la importancia de detectar el aspecto o la entidad, sin embargo el resto de características permitiría completar la información que nos ofrece esta valiosa fuente.

Los párrafos anteriores enumeran las líneas de investigación abiertas para este trabajo. Los modelos generados y las precisiones conseguidas permiten la aplicación de los mismos en herramientas de monitorización del sector turístico. En cualquier caso, tanto las técnicas DL como las fuentes de información evolucionan y cambian frecuentemente por lo que se requerirá de una actualización continua.

Apéndice A

Códigos utilizados

A.1. Apéndice: items.py

```
# -*- coding: utf-8 -*-

import scrapy

class HotelSentimentItem(scrapy.Item):
    title = scrapy.Field()
    content = scrapy.Field()
    stars = scrapy.Field()

class TripAdvisorReviewItem(scrapy.Item):
    # Fields to extract from Tripadvisor
    # tripadvisor_spyder.py
    title = scrapy.Field()
    content = scrapy.Field()
    review_stars = scrapy.Field()
    review_date = scrapy.Field()
    reviewer_location = scrapy.Field()


class BookingReviewItem(scrapy.Item):
    # Fields to extract fro Booking
    # booking_spyder.py
    title = scrapy.Field()
```

```
    score = scrapy.Field()
    positive_content = scrapy.Field()
    negative_content = scrapy.Field()
    review_date =scrapy.Field()
    reviewer_location = scrapy.Field()
```

A.2. Apéndice: tripadvisor_spider.py

```
import scrapy

from hotel_sentiment.items import TripAdvisorReviewItem


class TripadvisorSpider(scrapy.Spider):
    name = "tripadvisor"
    start_urls = [
        "https://www.tripadvisor.co.uk/Hotels-g187479-
            Tenerife_Canary_Islands-Hotels.html"
    ]

    def parse(self, response):
        for href in response.xpath('//div[@class="listing_title"]/a/@href
            '):
            url = response.urljoin(href.extract())
            yield scrapy.Request(url, callback=self.parse_hotel)

        next_page = response.xpath('//div[@class="unified pagination
            standard_pagination"]/child::*[2][self::a]/@href')
        if next_page:
            url = response.urljoin(next_page[0].extract())
            yield scrapy.Request(url, self.parse)

    def parse_hotel(self, response):
        for href in response.xpath('//div[starts-with(@class,"quote")]/a/
            @href'):
            url = response.urljoin(href.extract())
            yield scrapy.Request(url, callback=self.parse_review)
```

```
        next_page = response.xpath('//div[@class="unified␣pagination␣"]/
            child::*[2][self::a]/@href')
        if next_page:
            url = response.urljoin(next_page[0].extract())
            yield scrapy.Request(url, self.parse_hotel)


    def parse_review(self, response):
        item = TripAdvisorReviewItem()

        item['title'] = response.xpath('//div[@class="quote"]/text()').
            extract()[0][1:-1]
        item['content'] = response.xpath('//div[@class="entry"]/p/text()'
            ).extract()[0]
        item['review_stars'] = response.xpath('//span[@class="rate␣sprite
            -rating_s␣rating_s"]/img/@alt').extract()[0]
        item['reviewer_location'] = response.xpath('//div[@class="
            location"]/text()').extract()[0]
        item['review_date'] = response.xpath('//span[@class="ratingDate␣
            relativeDate"]/@content').extract()

        return item
```

A.3. Apéndice: booking_spider.py

```
import scrapy
from scrapy.loader import ItemLoader
from hotel_sentiment.items import BookingReviewItem

# maximo numero de páginas a obtyener de cada hotel
max_pages_per_hotel = 6

class BookingSpider(scrapy.Spider):
    name = "booking"
    start_urls = [
        "https://www.booking.com/searchresults.en-gb.html?aid=356984&
            label=gog235jc-country-en-gb-gb-unspec-es-com-L%3Aen-O%3Ax11
            -B%3Achrome-N%3AXX-S%3Abo-U%3Ac-H%3As&lang=en-gb&sid=
```

```
            e4ce8c840d65e7df8712a2df06268dbd&sb=1&src=country&src_elem=
            sb&error_url=https%3A%2F%2Fwww.booking.com%2Fcountry%2Fgb.en
            -gb.html%3Faid%3D356984%3Blabel%3Dgog235jc-country-en-gb-gb-
            unspec-es-com-L%253Aen-O%253Ax11-B%253Achrome-N%253AXX-S%253
            Abo-U%253Ac-H%253As%3Bsid%3De4ce8c840d65e7df8712a2df06268dbd
            %3Binac%3D0%26%3B&ss=Costa+Del+Silencio%2C+Canary+Islands%2C
            +Spain&checkin_monthday=&checkin_month=&checkin_year=&
            checkout_monthday=&checkout_month=&checkout_year=&room1=A%2
            CA&no_rooms=1&group_adults=2&group_children=0&ss_raw=costa+
            del+silencio&ac_position=0&ac_langcode=en&dest_id=900040737&
            dest_type=city&search_pageview_id=17cd7243c13c03d6&
            search_selected=true&search_pageview_id=17cd7243c13c03d6&
            ac_suggestion_list_length=5&ac_suggestion_theme_list_length
            =0"

    ]

    pageNumber = 1

    # parse diferent hotels
    def parse(self, response):
        for hotelurl in response.xpath('//a[@class="hotel_name_link␣url
            "]/@href'):
            url = response.urljoin(hotelurl.extract())
            yield scrapy.Request(url, callback=self.parse_hotel)

        next_page = response.xpath('//a[starts-with(@class,"paging-next")
            ]/@href')
        if next_page:
            url = response.urljoin(next_page[0].extract())
            yield scrapy.Request(url, self.parse)

    # parse each hotel
    def parse_hotel(self, response):
        reviewsurl = response.xpath('//a[@class="show_all_reviews_btn"]/
            @href')
        url = response.urljoin(reviewsurl[0].extract())
        self.pageNumber = 1
        return scrapy.Request(url, callback=self.parse_reviews)
```

```
#  parse the reviews
def parse_reviews(self, response):
    if self.pageNumber > max_pages_per_hotel:
        return
    for rev in response.xpath('//li[starts-with(@class,"review_item")
        ]'):
        item = BookingReviewItem()
        title = rev.xpath('.//a[@class="review_item_header_content"]/
            span[@itemprop="name"]/text()')
        if title:
            item['title'] = title[0].extract()
            positive_content = rev.xpath('.//p[@class="review_pos"]//
                span/text()')
            if positive_content:
                item['positive_content'] = positive_content[0].
                    extract()
            negative_content = rev.xpath('.//p[@class="review_neg"]//
                span/text()')
            if negative_content:
                item['negative_content'] = negative_content[0].
                    extract()
            item['score'] = rev.xpath('.//span[@itemprop="
                reviewRating"]/meta[@itemprop="ratingValue"]/
                @content')[0].extract()
            item['review_date'] = rev.xpath('.//meta[@itemprop="
                datePublished"]/@content')[0].extract()
            item['reviewer_location'] = rev.xpath('.//span[@class="
                reviewer_country"]/span[@itemprop="nationality"]/
                span[@itemprop="name"]/text()')[0].extract()

            yield item

    next_page = response.xpath('//a[@id="review_next_page_link"]/
        @href')
    if next_page:
        self.pageNumber += 1
        url = response.urljoin(next_page[0].extract())
        yield scrapy.Request(url, self.parse_reviews)
```

A.4. Apéndice: Preparar_tripadvisor_csv.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import pandas as pd

def get_class(stars):
    score = int(stars[0])
    if score > 3:
        return 'Good'
    else:
        return 'Bad'

df = pd.read_csv('itemsTripadvisor.csv', encoding='utf-8')

# Eliminamos duplicados
df.drop_duplicates(inplace=True)

# Eliminar comentarios con 3 estrellas
df = df[df['stars'] != '3 of 5 bubbles']

# Concatenamos título y contenido
df['full_content'] = df['title'] + '. ' + df['content']

# Transformamos la valoración en estrellas en la etiqueta "Good" o "Bad
df['true_category'] = df['stars'].apply(get_class)

df = df[['full_content', 'true_category']]

# Grabamos a csv
df.to_csv('Datos_Tripadvisor.csv', header=False, index=False, encoding='
    utf-8')

# imprimimos resumen de comentarios etiquetados
print(df['true_category'].value_counts())
```

A.5. Apéndice: Preparar_booking_csv.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# defino un funcion para transformar el score en Good o Bad segun
# si es mayor o menor que 5
def get_class(score):
    valor = int(score)
    if valor >= 5:
        return 'Good'
    else:
        return 'Bad'

# devuelve title+negative_content si true_category es 'Good'
#  devuelve title+positive_content si true_category es 'Bad'
def get_fullcontent(fila):
    if fila['true_category'] == 'Good':
        return str(fila['title'])+'. '+str(fila['positive_content'])
    else:
        return  str(fila['title'])+'. '+str(fila['negative_content'])



df = pd.read_csv('items_entrenarBooking_datos_entrenarv2.csv', encoding='
    utf-8')

# Eliminamos duplicados
df.drop_duplicates(inplace=True)


df['true_category'] = df['score'].apply(get_class)

df['full_content'] = df.apply(get_fullcontent,axis=1)

# me quedo solo con las columnas que me interesan
df = df[['full_content', 'true_category']]
df.to_csv('items_entrenarBooking_datos.csv',header=False, index=False,
    encoding='utf-8');
```

A.6. Apéndice: BoW_codification_example.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from keras.preprocessing.text import Tokenizer
import pandas as pd
from nltk.corpus import stopwords

def remove_stop_words_english(lor):
    sw = stopwords.words('english')
    new_lor =list()
    for i, review in enumerate(lor):
        new_review = [word for word in review.lower().split() if word not
            in sw]
        new_lor.append('␣'.join(new_review))
    return new_lor


################## MAIN BODY ####################

# read train file
workdir = '/home/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python/'
filename = workdir+'train_file.csv'
# read train data set as a data frame with three columns:
# "index" "comentario" "valor"
df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
list_of_reviews = df_train['comentario']

# if num_words is asigned, it takes the num_words - 1 more
# used words of the tokenizer.  Index 0 is reserved
# 100 words are choosen

# clean list of reviews to compare the tokenizer
list_of_reviews_cleaned = remove_stop_words_english(list_of_reviews)
#print(list_of_reviews_cleaned)
tokenizer2 = Tokenizer(num_words=100)
tokenizer2.fit_on_texts(list_of_reviews_cleaned)
encoded_docs = tokenizer2.texts_to_matrix(list_of_reviews_cleaned, mode='
```

```
    count')

i = 4820
print("Review:␣",i)
print(list_of_reviews[i])
print(list_of_reviews_cleaned[i])
print(encoded_docs[i])
```

A.7. Apéndice: Comparative_Simple_NeuralNetworkBoW_completo.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os
from keras import backend as K
import numpy as np
import tensorflow as tf
import random as rn
from keras.preprocessing.text import Tokenizer
from keras.models import Sequential
from keras.layers.core import Dense
#from keras.utils import plot_model
import pandas as pd
from nltk.corpus import stopwords

# print tokenizer sorted by the index
# of the vector that it will generate
# to number passed in limit
# This function is diferent to print(tokenizer.word_index)
# you need to use a loop to sort the output
def print_vocabulary(t,limit):
    iteration = 0
    for w in sorted(t.word_index, key=t.word_index.get, reverse=False):
        print("[",w,",", t.word_index[w],"]",end='␣', flush=True)
        iteration = iteration + 1
        if (iteration == limit):
```

```
            return


# print review - coded vector for BoW
def print_review_vector(lor,vec):
    for indice in range(len(lor)):
        print("Review [",indice,"]")
        print(len(lor[indice]),"--",len(vec[indice]))
        print(lor[indice])
        print(vec[indice])

def print_nonzero_components(t,v):
    # index 0 is reserved as padding model in tokenizer so first
        component
    # of a vector is always 0
    iteration = 1
    for w in sorted(t.word_index, key=t.word_index.get, reverse=False):
        if (v[t.word_index[w]] != 0):
            print("componente [",iteration,"] palabra = ", \
                w," apariciones = ",v[t.word_index[w]])
        iteration = iteration + 1
        if (iteration == v.size - 1):
            return


def remove_stop_words_english(lor):
    sw = stopwords.words('english')
    new_lor =list()
    for i, review in enumerate(lor):
        # print(i)
        # print(review)
        new_review = [word for word in review.lower().split() if word not
            in sw]
        # print(new_review)
        new_lor.append(' '.join(new_review))
        # print(new_lor[i])
    return new_lor
```

```
################## MODELS DEFINITION ##############################
# function to evaluate a model
# dataframe contains two cols: review, label
# tk is the tokenizer to encode the reviewsç
# clean specifies if it has to clean or not the reviews to be predicted
# cod_mode. 'binary', 'count', 'tfidf' or 'freq'
# return values: list with
# Good reviews, Bad reviews, Good predictions hits, Bad predictions hits,
# Good predictions fails, Bad predictions fails

def evaluate_model(model,dataframe,tk,clean,cod_mode):

    f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    positivos = 0   # total 'Good' reviews
    negativos = 0   # total 'Bad' reviews
    a_positivos = 0  # total predicted as 'Good' and labeled as 'Good'
    a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'
    lol, y = dataframe['comentario'], dataframe['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)
    # CHANGE TO CLEAN (OR NOT) REVIEWS
    if clean:
        lol_cleaned = remove_stop_words_english(lol)
    else:
        lol_cleaned = lol
    x = tk.texts_to_matrix(lol_cleaned, mode=cod_mode)
    print("Tensor␣x.ndim␣=␣",x.ndim, "␣x.shape␣=␣",x.shape \
                    ,"␣vector.dtype␣=␣",x.dtype)

    tam = len(dataframe)
    for i in list(range(tam)):
        vector = x[i].reshape(1,len(x[i]))
        prediction = model.predict(vector, verbose=0)
        prediction = int(round(prediction[0][0]))
        if (y[i] == 1):
            if (prediction == 1): a_positivos = a_positivos + 1
```

```
            else: f_negativos = f_negativos + 1
            positivos = positivos + 1
        else:
            if (prediction == 0): a_negativos = a_negativos + 1
            else: f_positivos = f_positivos + 1
            negativos = negativos + 1

    return [positivos,negativos,a_positivos,a_negativos,f_positivos,
        f_negativos]

# print results calculated in evaluate_model
def print_results(results):
    print("Comentarios positivos =",results[0])
    print("Comentarios negativos =",results[1])
    print("Aciertos en positivos =",results[2])
    print("Aciertos en negativos =",results[3])
    print("Fallos en positivos =",results[4])
    print("Fallos en negativos =",results[5])
    print("Porcentaje aciertos =", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              "%")
    print("Porcentaje fallos =", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
              "%")




# define a multilayer perceptron network. using dense layers
# relu for activation function in hidden layers and sigmoid in output
# optimizer adam and binary_crossentropy as loss function
# output layer only have 1 neuron
# num_inputs dimension of the input vector to the network
# layers is list. layers.size is number of hidden layers.
# layer[i] specifies number o neurons of hidden layer i+1
def multilayer_perceptron(num_inputs,layers):
```

```
    assert len(layers) > 0, \
        "Error, el número de capas ocultas debe ser mayor que 0"
    model = Sequential()
    first_layer = True
    for n in layers:
        if first_layer:
            model.add(Dense(n, input_shape=(num_inputs,), activation='
                relu'))
            first_layer = False
        else:
            model.add(Dense(n, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=[
        'accuracy'])
    model.summary()
    # uncomment next line to plot the model. Change filename in to_file
    #plot_model(model, to_file='model.png', show_shapes=True)
    return model




################## MAIN BODY ####################
# COMO HACER EL MODELO REPRODUCIBLE
# fix random seed for reproducibility
os.environ['PYTHONHASHSEED'] ='0'
np.random.seed(42) # para que numpy tenre aleatorios en un estado bien
    definido
rn.seed(12345)  # para que python parta de un estado bien definido
session_conf = tf.ConfigProto(intra_op_parallelism_threads=1,
    inter_op_parallelism_threads=1)
tf.set_random_seed(1234)
sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
K.set_session(sess)


# Parameters to run the tests
clean_reviews = False # to clean or do not clean the reviews
```

```
vector_dimension = [100,200,400,800,1600,0] # list of dimension of input
    layer
                                        # 100 - 100 neurons, 0 vocabulary
                                            size
verbose_train = 0 #  Integer. 0, 1, or 2. Verbosity mode. 0 = silent,
                  #1 = progress bar, 2 = one line per epoch
num_epochs = 20  # num of epochs
codif_vector = ['binary','count','tfidf','freq'] # diferent codfication
    BoW
codif = 'count'



# read train file
workdir = '/home/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python/'
filename = workdir+'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'
# read train data set as a data frame with three columns:
# "index" "comentario" "valor"
df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')


# extrat comments and value (converted 1 for 'Good' and 0 for 'Bad')
# into two lists
list_of_reviews, y_train = df_train['comentario'], df_train['valor'].
    apply( \
                                    lambda x: 1 if x == 'Good' else 0)


test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                    lambda x: 1 if x == 'Good' else 0)


# Loop diferent kinds of codification for Bow
for codif in codif_vector:
```

```
# Loop diferent kinds of input layer dimension
for dimension in vector_dimension:
    # CHANGE TO CLEAN (OR NOT) REVIEWS
    print ("
        ____________________________________________________________"
        )
    print ("Experimento: Bow Codification: ",codif,
           " input layer dimension: ", dimension)
    if clean_reviews:
        list_of_reviews_cleaned = remove_stop_words_english(
            list_of_reviews)
        test_reviews_cleaned = remove_stop_words_english(test_reviews
            )
    else:
        list_of_reviews_cleaned = list_of_reviews
        test_reviews_cleaned = test_reviews

    # create and fit a tokenizer, limiting it to 100 dim
    if dimension == 0:
        # create and fit a tokenizer, with all vocabulary
        tokenizer2 = Tokenizer()
    else:
        tokenizer2 = Tokenizer(num_words=dimension)

    tokenizer2.fit_on_texts(list_of_reviews_cleaned)


    x_train = tokenizer2.texts_to_matrix(list_of_reviews_cleaned,
        mode=codif)
    x_test = tokenizer2.texts_to_matrix(test_reviews_cleaned, mode=
        codif)


    print("Tensor x_train.ndim = ",x_train.ndim, " x_train.shape = ",
        x_train.shape \
          ," x_train.dtype = ",x_train.dtype)
    print("Tensor x_test.ndim = ",x_test.ndim, " x_test.shape = ",
        x_test.shape \
          ," x_test.dtype = ",x_test.dtype)
```

```
# x_train.shape[1] is dimensions of input vector
model = multilayer_perceptron(x_train.shape[1],[50])
model.fit(x_train, y_train, epochs=num_epochs, verbose=
    verbose_train)
loss, acc = model.evaluate(x_test, y_test, verbose=0)
print('Porcentaje de aciertos en el set de test: %f' %(acc*100))
resultados = evaluate_model(model,df_evaluation,tokenizer2,
    clean_reviews,codif)
print_results(resultados)
print ("
    ----------------------------------------------------------")
    )
```

A.8. Apéndice: SequencesExample.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Dec 18 10:49:18 2018

@author: carlos
"""
from keras.preprocessing.text import Tokenizer,text_to_word_sequence

# lee un fichero de texto separado por fin de lineas en un string
def file2string(filename):
    file = open(filename, 'r')
    text = file.read()
    file.close()
    return text

# load the document
workdir = '/home/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python/'
filename = workdir+'ReviewSamples.txt'
text = file2string(filename)
list_of_reviews = text.split('\n') # genero una lista. Cada review un
    elemento
```

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(list_of_reviews) # lo entreno con los comentarios

# se podría imprimir tokenizer.word_index, pero para que aparezca
    ordenado
# por el valor hay que hacer el bucle
print("Palabra␣con␣la␣que␣corresponde␣cada␣índice:")
count = 0

for w in sorted(tokenizer.word_index, key=tokenizer.word_index.get,
    reverse=False):
    pair = str(tokenizer.word_index[w])+'-'+w
    print('|␣',pair.rjust(15),end='␣')
    count = count + 1
    if count == 4:
        print('|')

        count = 0
print ('\n')


print("\n\n\n----------␣LISTA␣DE␣REVIEWS␣----------")

encoded_docs = tokenizer.texts_to_sequences(list_of_reviews)

for indice in range(len(list_of_reviews)):
    print("--------------------------------------------------")
    print("Review␣[",indice,"]")
    print(list_of_reviews[indice])
    print(encoded_docs[indice])
    lwords = text_to_word_sequence(list_of_reviews[indice], \
        filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n', lower=True, split
            ='␣')
    for w in lwords:
        print("[",w,",", tokenizer.word_index[w],"]")
```

A.9. Apéndice: Embedding_Flatten_example.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 11 10:28:09 2019

@author: carlos
"""
import numpy as np
from keras.layers import Input, Flatten
from keras.models import Model
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.layers.embeddings import Embedding
from keras.models import Sequential

inputs = Input(shape=(3,2,4))

def generate_model(words_in_vocab,vdim,emb_matrix,num_inputs):
    model = Sequential()
    model.add(Embedding(words_in_vocab, vdim, weights=[emb_matrix], \
                  input_length=num_inputs, trainable=False))
    model.add(Flatten())
    model.summary()
    return model


list_of_reviews = ['hotel was nice', 'good food', 'hotel nice pool']
tokenizer = Tokenizer()
tokenizer.fit_on_texts(list_of_reviews)
longest_review = max(len(review.split()) for review in list_of_reviews)
# IMPORTANT: we have to add 1 to vocabulary_size because the embedding
    matrix
# needs an extra row (row 0) to represent the vector for pad in sequences
vocabulary_size = len(tokenizer.word_index)+1

encoded_reviews = tokenizer.texts_to_sequences(list_of_reviews)
encoded_sequences = pad_sequences(encoded_reviews, maxlen=longest_review,
```

```
    \
                                padding='post',truncating='post')
# Create a embedding matrix. rows = nwords in vocabulary + 1,
# we have to add 1 because index 0 is reserved with to encode the padded
# components, so there is one row more than number of words in tokenizer
# cols = num of components of the vectors used in embedding
# initialize matriz with zeroes
vector_dim = 5
# example of embedding_matrix
# [0.0 0.0 0.0 0.0 0.0] codification for padding components
# [1.1 1.2 1.3 1.4 1.5] codification vector for 'hotel'
# [2.1 2.2 2.3 2.4 2.5] codification vector for 'nice'
# [3.1 3.2 3.3 3.4 3.5] codification vector for 'was'
                         ]
# [4.1 4.2 4.3 4.4 4.5] codification vector for 'good'
# [5.1 5.2 5.3 5.4 5.5] codification vector for 'food'
# [6.1 6.2 6.3 6.4 6.5] codification vector for 'pool'

embedding_matrix = np.array([[0.0,0.0,0.0,0.0,0.0], \
                   [1.1,1.2,1.3,1.4,1.5], \
                   [2.1,2.2,2.3,2.4,2.5], \
                   [3.1,3.2,3.3,3.4,3.5], \
                   [4.1,4.2,4.3,4.4,4.5], \
                   [5.1,5.2,5.3,5.4,5.5], \
                   [6.1,6.2,6.3,6.4,6.5]])
print('Reviews␣in␣set:␣')
print(list_of_reviews)
print('Tokenizer:␣')
print(tokenizer.word_index)
print("Embedding␣matrix␣(example):")
print(embedding_matrix)
model  = generate_model(vocabulary_size,vector_dim,embedding_matrix, \
                        longest_review)
print('shape␣of␣encoded␣sequences␣',encoded_sequences.shape)
# model.layers[0] is first layer, embedding for this model
print('output␣of␣embedded␣layer␣',model.layers[0].output)
# model.layers[1] is second layer, flatten for this model
print('output␣of␣flatten␣layer␣',model.layers[1].output)
```

```
# model.layers[0] is first layer, embedding for this model
embedded_layer = Model(inputs=model.input,
                                    outputs=model.layers[0].output)
# model.layers[0] is second layer, flatten for this model
flatten_layer = Model(inputs=model.input,
                                    outputs=model.layers[1].output)
for index in range(0,len(list_of_reviews)):

    x = encoded_sequences[index].reshape(1,longest_review)
    embedded_output = embedded_layer.predict(x)
    flatten_output = flatten_layer.predict(x)
    print(list_of_reviews[index])
    print(encoded_sequences[index])
    print(embedded_output)
    print(flatten_output)
```

A.10. Apéndice: Word2Vec_MLPerceptron.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 20 20:01:59 2019

@author: carlos
"""
import os
import random as rn
import numpy as np
from keras import backend as K
import tensorflow as tf
from gensim.models import Word2Vec
import gensim.models.keyedvectors as word2vec
from keras.preprocessing.text import text_to_word_sequence, Tokenizer
import pandas as pd
from keras.preprocessing.sequence import pad_sequences
from numpy import zeros
from keras.models import Sequential
from keras.layers import Flatten
```

```
from keras.layers.core import Dense
from keras.layers.embeddings import Embedding
from gensim.scripts.glove2word2vec import glove2word2vec


################# MODELS DEFINITION ##############################
# function to evaluate a model
# dataframe contains two cols: review, label
# tk is the tokenizer to encode the reviewsç
# clean specifies if it has to clean or not the reviews to be predicted
# cod_mode. 'binary', 'count', 'tfidf' or 'freq'
# return values: list with
# Good reviews, Bad reviews, Good predictions hits, Bad predictions hits,
# Good predictions fails, Bad predictions fails

def evaluate_model(model,dataframe,tk,longest_review):

    f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    positivos = 0   # total 'Good' reviews
    negativos = 0   # total 'Bad' reviews
    a_positivos = 0  # total predicted as 'Good' and labeled as 'Good'
    a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'
    lol, y = dataframe['comentario'], dataframe['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)

    x_v = tk.texts_to_sequences(lol)
    x = pad_sequences(x_v,maxlen=longest_review,padding='post' \
                      ,truncating='post')
    print("Tensor␣x.ndim␣=␣",x.ndim, "␣x.shape␣=␣",x.shape \
                      ,"␣vector.dtype␣=␣",x.dtype)

    tam = len(dataframe)
    for i in list(range(tam)):
        vector = x[i].reshape(1,len(x[i]))
        prediction = model.predict(vector, verbose=0)
        prediction = int(round(prediction[0][0]))
        if (y[i] == 1):
            if (prediction == 1): a_positivos = a_positivos + 1
```

```
            else: f_negativos = f_negativos + 1
            positivos = positivos + 1
        else:
            if (prediction == 0): a_negativos = a_negativos + 1
            else: f_positivos = f_positivos + 1
            negativos = negativos + 1

    return [positivos,negativos,a_positivos,a_negativos,f_positivos \
            ,f_negativos]

# print results calculated in evaluate_model
def print_results(results):
    print("Comentarios positivos = ",results[0])
    print("Comentarios negativos = ",results[1])
    print("Aciertos en positivos = ",results[2])
    print("Aciertos en negativos = ",results[3])
    print("Fallos en positivos = ",results[4])
    print("Fallos en negativos = ",results[5])
    print("Porcentaje aciertos = ", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              "%")
    print("Porcentaje fallos = ", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
              "%")



# define a multilayer perceptron network. using a layer with
# embeded pretrained (pass a embedding matrix) and dense layers
# relu for activation function in hidden layers and sigmoid in output
# optimizer adam and binary_crossentropy as loss function
# output layer only have 1 neuron
# num_inputs dimension of the input vector to the network
# layers is list. layers.size is number of hidden layers.
# layer[i] specifies number o neurons of hidden layer i+1
# words_in_vocab number or word in the used vocabulary = rows in
    emb_matrix
# vdim = number of components in embedding vectors = cols in emb_matrix
def embedding_multilayer_perceptron(num_inputs,layers,words_in_vocab,vdim
```

```
 \
                                    ,emb_matrix):
    assert len(layers) > 0, \
        "Error, el número de capas ocultas debe ser mayor que 0"
    model = Sequential()
    # add and embedding layer without training
    model.add(Embedding(words_in_vocab, vdim, weights=[emb_matrix], \
                input_length=num_inputs, trainable=False))
    model.add(Flatten())
    for n in layers:
        model.add(Dense(n, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                metrics=['accuracy'])
    model.summary()
    return model



# read train file
workdir = '/home/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python/'
filename = workdir+'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'
filename_word2vec_model = workdir+'tourist_reviews_model.bin'
verbose_train = 2 #  Integer. 0, 1, or 2. Verbosity mode. 0 = silent,
                #1 = progress bar, 2 = one line per epoch
num_epochs = 20  # num of epochs
#seq_sizes  = [100,200,400,800,1600,0] # list of sizes to padd or cut
    reviews
                                #  0 max review length

seq_sizes  = [100,200,400,800,1600,0] # list of sizes to padd or cut
    reviews
                                #  0 max review length

# choose word2vec method. Only one can be true
google_w2v = False
train_w2v = False
```

```
glove_w2v = True


# COMO HACER EL MODELO REPRODUCIBLE
# fix random seed for reproducibility
os.environ['PYTHONHASHSEED'] ='0'
np.random.seed(42) # para que numpy tenre aleatorios en un estado bien
    definido
rn.seed(12345)  # para que python parta de un estado bien definido
session_conf = tf.ConfigProto(intra_op_parallelism_threads=1 \
                              , inter_op_parallelism_threads=1)
tf.set_random_seed(1234)
sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
K.set_session(sess)

# https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=
    sharing
# 3000000 words in dictionary
# produce vector 300 floats
model_google_file='/Users/carlos/Desktop/GoogleNews-vectors-negative300.
    bin'
# Glove model file
# http://nlp.stanford.edu/data/glove.6B.zip
# This zip file cotains embeedin for 50,100,200,300 dim vector
# extract the txt file and convert it using glove2word2vec in other file
# This last file is which use.
#model_glove_file='/Users/carlos/Desktop/glove.6B/glove.6B.300d.word2vec'
model_glove_file='/home/carlos/Escritorio/glove.6B/glove.6B.300d.word2vec
    .txt'
# read train data set as a data frame with three columns:
# "index" "comentario" "valor"
df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')


# extrat comments and value (converted 1 for 'Good' and 0 for 'Bad')
# into two lists
```

```
list_of_reviews, y_train = df_train['comentario'], df_train['valor'].
    apply( \
                                lambda x: 1 if x == 'Good' else 0)
test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)
tokenizer = Tokenizer()
tokenizer.fit_on_texts(list_of_reviews)
# number of words in tokenizer
# we have to add 1 because index 0 is reserved with to encode the padded
# components, so there is one row more than number of words in tokenizer
nwords = len(tokenizer.word_index) + 1
# encode the reviews into integers using tokenizer
encoded_reviews = tokenizer.texts_to_sequences(list_of_reviews)
encoded_test_reviews = tokenizer.texts_to_sequences(test_reviews)
# conver list of reviews, in a list of sentences splitted into words
lor_in_words = [ text_to_word_sequence(review) for review in
    list_of_reviews]

# number of components of vectors
vector_dim = 300
if google_w2v:
    print("Using␣Google␣pre-trained␣Word2Vec")
    model_emb = word2vec.KeyedVectors.load_word2vec_format( \
                                    model_google_file, binary=True)
if train_w2v:
    # Building de Word2Vec model. Accepted parameters:
    # size = dimensions of the output vector (default is 100)
    # window: size of the windows to consider neighbour words (default is
        5)
    # min_count: minimum number of occurences of a word to be
    #                   considered in training (default is 5)
    # workers: threads used in traingin (default is 3)
    # sg: 0 - CBOW (default) 1 - skip gram
    model_emb = Word2Vec(lor_in_words,size=vector_dim,min_count=1)
if glove_w2v:
    print("Using␣GloVe␣pre-trained␣Word2Vec")
    # binary False, you can read de file
    model_emb = word2vec.KeyedVectors.load_word2vec_format( \
                                    model_glove_file, binary=False)
```

```
# Create a embedding matrix. rows = number of words in vocabulary + 1
# because there will be zeros (in padded components)
# cols = num of components of the vectors used in embedding
# initialize matriz with zeroes
embedding_matrix = zeros((nwords, vector_dim))
for word, i in tokenizer.word_index.items():
    if word in model_emb:
        vector = model_emb[word]
        embedding_matrix[i] = vector


longest_review = max(len(review.split()) for review in list_of_reviews)


for s_size in seq_sizes:
    if s_size == 0:
        input_size = longest_review
    else:
        input_size = s_size
    # padd and truncate sequences to the specified size
    x_train = pad_sequences(encoded_reviews, maxlen=input_size, \
                                padding='post',truncating='post')
    x_test = pad_sequences(encoded_test_reviews, maxlen=input_size, \
                                padding='post',truncating='post')
    print("Tensor x_train.ndim =",x_train.ndim, " x_train.shape =", \
          x_train.shape," x_train.dtype =",x_train.dtype)
    print("Tensor x_test.ndim =",x_test.ndim, " x_test.shape =", \
          x_test.shape," x_test.dtype =",x_test.dtype)
    model_net = embedding_multilayer_perceptron(input_size,[50],nwords,\
                                        vector_dim,embedding_matrix)
    model_net.fit(x_train, y_train, epochs=num_epochs, verbose=
        verbose_train)
    loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
    print('Porcentaje de aciertos en el set de test: %f' %(acc*100))
    resultados = evaluate_model(model_net,df_evaluation,tokenizer,
        input_size)
    print_results(resultados)
    print ("————————————————————————————————————————————————————————")
```

A.11. Apéndice: Embedding_MLPerceptron.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 20 20:01:59 2019

@author: carlos
"""
import os
import random as rn
import numpy as np
from keras import backend as K
import tensorflow as tf
from keras.preprocessing.text import text_to_word_sequence, Tokenizer
import pandas as pd
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Flatten
from keras.layers.core import Dense
from keras.layers.embeddings import Embedding


################## MODELS DEFINITION ##############################
# function to evaluate a model
# dataframe contains two cols: review, label
# tk is the tokenizer to encode the reviewsç
# clean specifies if it has to clean or not the reviews to be predicted
# cod_mode. 'binary', 'count', 'tfidf' or 'freq'
# return values: list with
# Good reviews, Bad reviews, Good predictions hits, Bad predictions hits,
# Good predictions fails, Bad predictions fails

def evaluate_model(model,dataframe,tk,longest_review):

    f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    positivos = 0   # total 'Good' reviews
    negativos = 0   # total 'Bad' reviews
```

```
    a_positivos = 0  # total predicted as 'Good' and labeled as 'Good'
    a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'
    lol, y = dataframe['comentario'], dataframe['valor'].apply( \
                                    lambda x: 1 if x == 'Good' else 0)

    x_v = tk.texts_to_sequences(lol)
    x = pad_sequences(x_v,maxlen=longest_review,padding='post' \
                      ,truncating='post')
    print("Tensor x.ndim = ",x.ndim, " x.shape = ",x.shape \
                      ,"vector.dtype = ",x.dtype)

    tam = len(dataframe)
    for i in list(range(tam)):
        vector = x[i].reshape(1,len(x[i]))
        prediction = model.predict(vector, verbose=0)
        prediction = int(round(prediction[0][0]))
        if (y[i] == 1):
            if (prediction == 1): a_positivos = a_positivos + 1
            else: f_negativos = f_negativos + 1
            positivos = positivos + 1
        else:
            if (prediction == 0): a_negativos = a_negativos + 1
            else: f_positivos = f_positivos + 1
            negativos = negativos + 1

    return [positivos,negativos,a_positivos,a_negativos,f_positivos \
            ,f_negativos]

# print results calculated in evaluate_model
def print_results(results):
    print("Comentarios positivos = ",results[0])
    print("Comentarios negativos = ",results[1])
    print("Aciertos en positivos = ",results[2])
    print("Aciertos en negativos = ",results[3])
    print("Fallos en positivos = ",results[4])
    print("Fallos en negativos = ",results[5])
    print("Porcentaje aciertos = ", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              "%")
```

```
    print("Porcentaje fallos = ", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
              "%")



# define a multilayer perceptron network with an embedding layer
# that has to be trained together with dense layers
# relu for activation function in hidden layers and sigmoid in output
# optimizer adam and binary_crossentropy as loss function
# output layer only have 1 neuron
# num_inputs dimension of the input vector to the network
# layers is list. layers.size is number of hidden layers.
# layer[i] specifies number o neurons of hidden layer i+1
# words_in_vocab number or word in the used vocabulary = rows in
    emb_matrix
# vdim = number of components in embedding vectors = cols in emb_matrix
def embedding_multilayer_perceptron(num_inputs,layers,words_in_vocab,vdim
    ):
    assert len(layers) > 0, \
        "Error, el número de capas ocultas debe ser mayor que 0"
    model = Sequential()
    # add and embedding layer with training
    model.add( \
        Embedding(words_in_vocab,vdim,input_length=num_inputs, trainable=
            True))
    model.add(Flatten())
    for n in layers:
        model.add(Dense(n, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                  metrics=['accuracy'])
    model.summary()
    return model



# read train file
workdir = '/Users/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python/
```

```
    '
filename = workdir+'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'

verbose_train = 2 #  Integer. 0, 1, or 2. Verbosity mode. 0 = silent,
                  #1 = progress bar, 2 = one line per epoch
num_epochs = 10  # num of epochs
# seq_sizes  = [100,200,400,800,1600,0] # list of sizes to padd or cut
    reviews
                                  #  0 max review length
seq_sizes  = [100,200,400,800,1600,0]

# COMO HACER EL MODELO REPRODUCIBLE
# fix random seed for reproducibility
os.environ['PYTHONHASHSEED'] ='0'
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
np.random.seed(42) # para que numpy tenre aleatorios en un estado bien
    definido
rn.seed(12345)  # para que python parta de un estado bien definido
session_conf = tf.ConfigProto(intra_op_parallelism_threads=1 \
                              , inter_op_parallelism_threads=1)
tf.set_random_seed(1234)
sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
K.set_session(sess)


# read train data set as a data frame with three columns:
# "index" "comentario" "valor"
df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')


# extrat comments and value (converted 1 for 'Good' and 0 for 'Bad')
# into two lists
list_of_reviews, y_train = df_train['comentario'], df_train['valor'].
    apply( \
```

```
                                    lambda x: 1 if x == 'Good' else 0)
test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                    lambda x: 1 if x == 'Good' else 0)
tokenizer = Tokenizer()
tokenizer.fit_on_texts(list_of_reviews)
# number of words in tokenizer
# it is important to add 1 to calculate the total number of words,
    because
# this parameter is used in the embedding layer to build the embedding
    matrix
# and it has to take in account that the index 0 (used as padding) needs
# an extra row in the embedding matrix
nwords = len(tokenizer.word_index) + 1
print("Tamaño del vocabulario + 1 =",nwords)
# encode the reviews into integers using tokenizer
encoded_reviews = tokenizer.texts_to_sequences(list_of_reviews)
encoded_test_reviews = tokenizer.texts_to_sequences(test_reviews)
# conver list of reviews, in a list of sentences splitted into words
lor_in_words = [ text_to_word_sequence(review) for review in
    list_of_reviews]

# number of components of vectors
#vector_dim = 300
vector_dim = 25
longest_review = max(len(review.split()) for review in list_of_reviews)


for s_size in seq_sizes:
    if s_size == 0:
        input_size = longest_review
    else:
        input_size = s_size
    # padd and truncate sequences to the specified size
    x_train = pad_sequences(encoded_reviews, maxlen=input_size, \
                                padding='post',truncating='post')
    x_test = pad_sequences(encoded_test_reviews, maxlen=input_size, \
                                padding='post',truncating='post')
    print("Tensor x_train.ndim =",x_train.ndim, " x_train.shape =", \
          x_train.shape," x_train.dtype =",x_train.dtype)
```

```
print("Tensor x_test.ndim = ",x_test.ndim, " x_test.shape = ", \
      x_test.shape," x_test.dtype = ",x_test.dtype)
model_net = embedding_multilayer_perceptron(input_size,[50],nwords,\
                                            vector_dim)
model_net.fit(x_train, y_train, epochs=num_epochs, verbose=
    verbose_train)
loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
print('Porcentaje de aciertos en el set de test: %f' %(acc*100))
resultados = evaluate_model(model_net,df_evaluation,tokenizer,
    input_size)
print_results(resultados)
print ("------------------------------------------------------------")
```

A.12. Apéndice: SVMExample.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Sep  6 11:05:11 2019

@author: carlos
"""

import pandas as pd
from keras.preprocessing.text import Tokenizer
import numpy as np
from nltk.corpus import stopwords
from sklearn import svm
from sklearn.metrics import accuracy_score

################## PARAMETERS ####################
clean_reviews = False # True - remove stop words, False = do not remove

# Using codification BoW.  Dimension = 0 vectors will have vocabulary
    size
# as dimension. Dimension  = X vectors will have X as dimension, and only
# most X used words will be taken in account to code the texts
dimension = 800
```

```
# when using BoW select codification method:
# 'binary','count','tfidf','freq']
codif = 'freq'


# nu-values for experimentes
# nu values from 0.01 to 1 step 0.01 (last parameter of arange)
nuvalues = np.arange(0.01,1,0.01)
# kernel values for experiments
kernelvalues = ['linear','poly','rbf','sigmoid']
gammavalue=0.5


################## FUNCTIONS ####################

def remove_stop_words_english(lor):
    sw = stopwords.words('english')
    new_lor =list()
    for i, review in enumerate(lor):
        # print(i)
        # print(review)
        new_review = [word for word in review.lower().split() if word not
            in sw]
        # print(new_review)
        new_lor.append('␣'.join(new_review))
        # print(new_lor[i])
    return new_lor


################## MODELS DEFINITION ##############################
# function to evaluate a model
# dataframe contains two cols: review, label
# tk is the tokenizer to encode the reviewsç
# clean specifies if it has to clean or not the reviews to be predicted
# cod_mode. 'binary', 'count', 'tfidf' or 'freq'
# return values: list with
# Good reviews, Bad reviews, Good predictions hits, Bad predictions hits,
# Good predictions fails, Bad predictions fails
```

```
def evaluate_model(model,dataframe,tk,clean,cod_mode):

    f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    positivos = 0   # total 'Good' reviews
    negativos = 0   # total 'Bad' reviews
    a_positivos = 0  # total predicted as 'Good' and labeled as 'Good'
    a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'
    lol, y = dataframe['comentario'], dataframe['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)
    # CHANGE TO CLEAN (OR NOT) REVIEWS
    if clean:
        lol_cleaned = remove_stop_words_english(lol)
    else:
        lol_cleaned = lol
    x = tk.texts_to_matrix(lol_cleaned, mode=cod_mode)
    print("Tensor x.ndim = ",x.ndim, " x.shape = ",x.shape \
                    ," vector.dtype = ",x.dtype)

    tam = len(dataframe)
    for i in list(range(tam)):
        vector = x[i].reshape(1,len(x[i]))
        prediction = model.predict(vector)
        # prediction = int(round(prediction[0][0]))
        if (y[i] == 1):
            if (prediction == 1): a_positivos = a_positivos + 1
            else: f_negativos = f_negativos + 1
            positivos = positivos + 1
        else:
            if (prediction == 0): a_negativos = a_negativos + 1
            else: f_positivos = f_positivos + 1
            negativos = negativos + 1

    return [positivos,negativos,a_positivos,a_negativos,f_positivos,
        f_negativos]

# print results calculated in evaluate_model
def print_results(results):
```

```
    print("Comentarios positivos =",results[0])
    print("Comentarios negativos =",results[1])
    print("Aciertos en positivos =",results[2])
    print("Aciertos en negativos =",results[3])
    print("Fallos en positivos =",results[4])
    print("Fallos en negativos =",results[5])
    print("Porcentaje aciertos =", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              "%")
    print("Porcentaje fallos =", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
              "%")



# print experimente results in one line
def print_experiment_result(initialstring,results):
    print(initialstring,",",results[0],",",results[1],",",results[2],",",
        \
          results[3],",",results[4],",",results[5],",", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              ",", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2))


################## MAIN BODY ####################
# fix de random seed to prevent diferent results
np.random.seed(500)

# Load files

workdir = '/Users/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python/
    '
filename = workdir+'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'
# read train data set as a data frame with three columns:
# "index" "comentario" "valor"
df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
```

```
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')

# extract comments and value (converted 1 for 'Good' and 0 for 'Bad')
# into two lists
list_of_reviews, y_train = df_train['comentario'], df_train['valor'].
    apply( \
                                lambda x: 1 if x == 'Good' else 0)


test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)

if clean_reviews:
    list_of_reviews_cleaned = remove_stop_words_english(list_of_reviews)
    test_reviews_cleaned = remove_stop_words_english(test_reviews)
else:
    list_of_reviews_cleaned = list_of_reviews
    test_reviews_cleaned = test_reviews

# create and fit a tokenizer, limiting it to 100 dim
if dimension == 0:
    # create and fit a tokenizer, with all vocabulary
    tokenizer2 = Tokenizer()
else:
    tokenizer2 = Tokenizer(num_words=dimension)

tokenizer2.fit_on_texts(list_of_reviews_cleaned)


x_train = tokenizer2.texts_to_matrix(list_of_reviews_cleaned, mode=codif)
x_test = tokenizer2.texts_to_matrix(test_reviews_cleaned, mode=codif)

# model parameters:
#
# nu (0,1]  next to 0 allows many bad labeled training reviews. 1 = hard
    margin
#
```

```
# kernel 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable
# If none is given, ‘rbf will be used. If a callable is given it is used
    to
# precompute the kernel matrix.
#
# degree of the polynomial kernel 'poly'. Ignored by all other kernels.
# Default is degree = 3
print( "BowCode,Kernel,Nu,GoodRew,BadRew,Goodhits,Badhits,FalseGood,", \
      "FalseBad,%hits,%fails")
for k in kernelvalues:
    for nuv in nuvalues:
        print("================================================")
        if (k == 'sigmoid'):
            print("Mode BoW code: ",codif,"SVM, kernel = ",k,"nu = ",
                nuv, \
                  "gamma=",gammavalue,"\n\n")
            model_svm = svm.NuSVC(nu=nuv, kernel=k,gamma=gammavalue)
        else:
            print("Mode BoW code: ",codif,"SVM, kernel = ",k,"nu = ",
                nuv)
            model_svm = svm.NuSVC(nu=nuv, kernel)

        model_svm.fit(x_train,y_train)
        predictions = model_svm.predict(x_test)
        #print("Precisión SVM: ",accuracy_score(predictions,y_test)*100)
        resultados = evaluate_model(model_svm,df_evaluation,tokenizer2,
            clean_reviews,codif)
        #print_results(resultados)
        initial_string = codif+","+k+","+str(nuv)
        print_experiment_result(initial_string,resultados)
```

A.13. Apéndice: Word2Vec_CNN.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 20 20:01:59 2019
```

```
@author: carlos
"""
import os
import random as rn
import numpy as np
from keras import backend as K
import tensorflow as tf
import gensim.models.keyedvectors as word2vec
from keras.preprocessing.text import text_to_word_sequence, Tokenizer
import pandas as pd
from keras.preprocessing.sequence import pad_sequences
from numpy import zeros
from keras.models import Sequential
from keras.layers import Flatten
from keras.layers.core import Dense
from keras.layers.embeddings import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D


################## MODELS DEFINITION ##############################
# function to evaluate a model
# dataframe contains two cols: review, label
# tk is the tokenizer to encode the reviewsç
# clean specifies if it has to clean or not the reviews to be predicted
# cod_mode. 'binary', 'count', 'tfidf' or 'freq'
# return values: list with
# Good reviews, Bad reviews, Good predictions hits, Bad predictions hits,
# Good predictions fails, Bad predictions fails

def evaluate_model(model,dataframe,tk,longest_review):

    f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    positivos = 0   # total 'Good' reviews
    negativos = 0   # total 'Bad' reviews
    a_positivos = 0  # total predicted as 'Good' and labeled as 'Good'
    a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'
    lol, y = dataframe['comentario'], dataframe['valor'].apply( \
```

```
                                    lambda x: 1 if x == 'Good' else 0)

    x_v = tk.texts_to_sequences(lol)
    x = pad_sequences(x_v,maxlen=longest_review,padding='post' \
                      ,truncating='post')
    print("Tensor x.ndim = ",x.ndim, " x.shape = ",x.shape \
                      ," vector.dtype = ",x.dtype)

    tam = len(dataframe)
    for i in list(range(tam)):
        vector = x[i].reshape(1,len(x[i]))
        prediction = model.predict(vector, verbose=0)
        prediction = int(round(prediction[0][0]))
        if (y[i] == 1):
            if (prediction == 1): a_positivos = a_positivos + 1
            else: f_negativos = f_negativos + 1
            positivos = positivos + 1
        else:
            if (prediction == 0): a_negativos = a_negativos + 1
            else: f_positivos = f_positivos + 1
            negativos = negativos + 1

    return [positivos,negativos,a_positivos,a_negativos,f_positivos \
            ,f_negativos]

# print results calculated in evaluate_model
def print_results(results):
    print("Comentarios positivos = ",results[0])
    print("Comentarios negativos = ",results[1])
    print("Aciertos en positivos = ",results[2])
    print("Aciertos en negativos = ",results[3])
    print("Fallos en positivos = ",results[4])
    print("Fallos en negativos = ",results[5])
    print("Porcentaje aciertos = ", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              "%")
    print("Porcentaje fallos = ", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
              "%")
```

```
# define a model with a Convolutional layer -> Maxpool layer
# num_inputs dimension of the input vector to the network
# layers is list. layers.size is number of hidden layers after maxpool
# layer[i] specifies number o neurons of hidden layer i+1
# words_in_vocab number or word in the used vocabulary = rows in
    emb_matrix
# vdim = number of components in embedding vectors = cols in emb_matrix
# numf_conv = number o filters to be applied in convolutional layer
# ksize = kernel size of the convolutional
# psize = pool size of the pooling

def CNN_pooling_multilayer_perceptron(num_inputs,layers,words_in_vocab,
    vdim, \
                                      emb_matrix,numf_conv,ksize,psize):
    assert len(layers) > 0, \
        "Error, el número de capas ocultas debe ser mayor que 0"
    model = Sequential()
    model.add(Embedding(words_in_vocab, vdim, weights=[emb_matrix], \
                  input_length=num_inputs, trainable=False))
    # padding='same' makes a padding frame, so the output will be the
    # shame lenght as input
    model.add(Conv1D(filters=numf_conv, kernel_size=ksize, padding='same'
        , \
                     activation='relu'))
    model.add(MaxPooling1D(pool_size=psize))
    model.add(Flatten())
    for n in layers:
        model.add(Dense(n, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                  metrics=['accuracy'])
    model.summary()
    return model
```

```
# read train file
workdir = '/Users/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python/
    '
filename = workdir+'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'
filename_word2vec_model = workdir+'tourist_reviews_model.bin'
verbose_train = 2 #  Integer. 0, 1, or 2. Verbosity mode. 0 = silent,
                  #1 = progress bar, 2 = one line per epoch
num_epochs = 20  # num of epochs
nfiltersC = 128 # num of filters to apply in Convolution
kernel_s = 3 # kernel size in convolution
pooling_s = 2 # pooling size in maxpool operation

input_size = 800 # sequece size. Cut or pad reviews to this size (in
    words)

# COMO HACER EL MODELO REPRODUCIBLE
# fix random seed for reproducibility
os.environ['PYTHONHASHSEED'] ='0'
np.random.seed(42) # para que numpy tenre aleatorios en un estado bien
    definido
rn.seed(12345)  # para que python parta de un estado bien definido
session_conf = tf.ConfigProto(intra_op_parallelism_threads=1 \
                              , inter_op_parallelism_threads=1)
tf.set_random_seed(1234)
sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
K.set_session(sess)

# https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=
    sharing
# 3000000 words in dictionary
# produce vector 300 floats
model_google_file='/Users/carlos/Desktop/GoogleNews-vectors-negative300.
    bin'

df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
```

```
    -8')


# extract comments and value (converted 1 for 'Good' and 0 for 'Bad')
# into two lists
list_of_reviews, y_train = df_train['comentario'], df_train['valor'].
    apply( \
                                lambda x: 1 if x == 'Good' else 0)
test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)
tokenizer = Tokenizer()
tokenizer.fit_on_texts(list_of_reviews)

# number of words in tokenizer
# we have to add 1 because index 0 is reserved with to encode the padded
# components, so there is one row more than number of words in tokenizer
nwords = len(tokenizer.word_index) + 1
# encode the reviews into integers using tokenizer
encoded_reviews = tokenizer.texts_to_sequences(list_of_reviews)
encoded_test_reviews = tokenizer.texts_to_sequences(test_reviews)
# convert list of reviews, in a list of sentences splitted into words
lor_in_words = [ text_to_word_sequence(review) for review in
    list_of_reviews]

# number of components of vectors
vector_dim = 300

#    print("Using Google pre-trained Word2Vec")
model_emb = word2vec.KeyedVectors.load_word2vec_format( \
                                    model_google_file, binary=True)

# Create a embedding matrix. rows = number of words in vocabulary + 1
# because there will be zeros (in padded components)
# cols = num of components of the vectors used in embedding
# initialize matriz with zeroes
embedding_matrix = zeros((nwords, vector_dim))
for word, i in tokenizer.word_index.items():
    if word in model_emb:
        vector = model_emb[word]
```

```
        embedding_matrix[i] = vector


x_train = pad_sequences(encoded_reviews, maxlen=input_size, \
                                padding='post',truncating='post')
x_test = pad_sequences(encoded_test_reviews, maxlen=input_size, \
                                padding='post',truncating='post')
print("Tensor x_train.ndim =",x_train.ndim, " x_train.shape =", \
          x_train.shape," x_train.dtype =",x_train.dtype)
print("Tensor x_test.ndim =",x_test.ndim, " x_test.shape =", \
          x_test.shape," x_test.dtype =",x_test.dtype)


model_net = CNN_pooling_multilayer_perceptron(input_size,[50],nwords,\
                    vector_dim,embedding_matrix,nfiltersC,kernel_s,
                        pooling_s)


model_net.fit(x_train, y_train, epochs=num_epochs, verbose=verbose_train)
loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
print('Porcentaje de aciertos en el set de test: %f' %(acc*100))
resultados = evaluate_model(model_net,df_evaluation,tokenizer,input_size)
print_results(resultados)
print ("-----------------------------------------------------------------")
print ("-----------------------------------------------------------------")
print ("-----------------------------------------------------------------")

# uncomment next section to print review codification example
"""
# model_net.layers[0] is first layer, embedding for this model
embedded_layer = Model(inputs=model_net.input,
                                outputs=model_net.layers[0].output)
index = 0 # index to use as example
lista = ["The pool was very nice, it always was warm. \
         Food and staff are fantastic. I will recommend this hotel."]
encoded_lista = tokenizer.texts_to_sequences(lista)
encoded_lista2 = pad_sequences(encoded_lista, maxlen=input_size, \
                                padding='post',truncating='post')
comentario_tok = text_to_word_sequence(lista[index])
```

```
x = encoded_lista2[index].reshape(1,input_size)
print("Muestra ",index," de ejemplo original -> token -> secuencia ->
    coded")
embedded_output = embedded_layer.predict(x)
print(lista[index])
print(comentario_tok)
print(encoded_lista2[index])
print(embedded_output)
print("Tamaño del vocabulario = ",nwords)
"""
```

A.14. Apéndice: Word2Vec_CNNv2.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 20 20:01:59 2019

@author: carlos
"""
import os
import random as rn
import numpy as np
from keras import backend as K
import tensorflow as tf
import gensim.models.keyedvectors as word2vec
from keras.preprocessing.text import text_to_word_sequence, Tokenizer
import pandas as pd
from keras.preprocessing.sequence import pad_sequences
from numpy import zeros
from keras.models import Sequential
from keras.layers import Flatten
from keras.layers.core import Dense
from keras.layers.embeddings import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
```

```
################## MODELS DEFINITION ##############################
# function to evaluate a model
# dataframe contains two cols: review, label
# tk is the tokenizer to encode the reviewsç
# clean specifies if it has to clean or not the reviews to be predicted
# cod_mode. 'binary', 'count', 'tfidf' or 'freq'
# return values: list with
# Good reviews, Bad reviews, Good predictions hits, Bad predictions hits,
# Good predictions fails, Bad predictions fails

def evaluate_model(model,dataframe,tk,longest_review):

    f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    positivos = 0   # total 'Good' reviews
    negativos = 0   # total 'Bad' reviews
    a_positivos = 0  # total predicted as 'Good' and labeled as 'Good'
    a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'
    lol, y = dataframe['comentario'], dataframe['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)

    x_v = tk.texts_to_sequences(lol)
    x = pad_sequences(x_v,maxlen=longest_review,padding='post' \
                      ,truncating='post')
    print("Tensor x.ndim = ",x.ndim, " x.shape = ",x.shape \
                      ," vector.dtype = ",x.dtype)

    tam = len(dataframe)
    for i in list(range(tam)):
        vector = x[i].reshape(1,len(x[i]))
        prediction = model.predict(vector, verbose=0)
        prediction = int(round(prediction[0][0]))
        if (y[i] == 1):
            if (prediction == 1): a_positivos = a_positivos + 1
            else: f_negativos = f_negativos + 1
            positivos = positivos + 1
        else:
            if (prediction == 0): a_negativos = a_negativos + 1
            else: f_positivos = f_positivos + 1
```

```
            negativos = negativos + 1

    return [positivos,negativos,a_positivos,a_negativos,f_positivos \
            ,f_negativos]

# print results calculated in evaluate_model
def print_results(results):
    print("Comentarios positivos =",results[0])
    print("Comentarios negativos =",results[1])
    print("Aciertos en positivos =",results[2])
    print("Aciertos en negativos =",results[3])
    print("Fallos en positivos =",results[4])
    print("Fallos en negativos =",results[5])
    print("Porcentaje aciertos =", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              "%")
    print("Porcentaje fallos =", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
              "%")


# define a model with a Convolutional layer -> Maxpool layer
# num_inputs dimension of the input vector to the network
# layers is list. layers.size is number of hidden layers after maxpool
# layer[i] specifies number o neurons of hidden layer i+1
# words_in_vocab number or word in the used vocabulary = rows in
    emb_matrix
# vdim = number of components in embedding vectors = cols in emb_matrix
# numf_conv = number o filters to be applied in convolutional layer
# ksize = kernel size of the convolutional
# psize = pool size of the pooling

def CNN_pooling_multilayer_perceptron(num_inputs,layers,words_in_vocab,
    vdim, \
                                      emb_matrix,numf_conv,ksize,psize):
    assert len(layers) > 0, \
        "Error, el número de capas ocultas debe ser mayor que 0"
    model = Sequential()
    model.add(Embedding(words_in_vocab, vdim, weights=[emb_matrix], \
```

```
                input_length=num_inputs, trainable=False))
    # padding='same' makes a padding frame, so the output will be the
    # shame lenght as input
    model.add(Conv1D(filters=numf_conv, kernel_size=ksize, padding='same'
        , \
                     activation='relu'))
    model.add(MaxPooling1D(pool_size=psize))
    model.add(Flatten())
    for n in layers:
        model.add(Dense(n, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                  metrics=['accuracy'])
    # model.summary()
    return model

# define a model with a 2 Convolutional layer -> Maxpool layer
# num_inputs dimension of the input vector to the network
# layers is list. layers.size is number of hidden layers after maxpool
# layer[i] specifies number o neurons of hidden layer i+1
# words_in_vocab number or word in the used vocabulary = rows in
    emb_matrix
# vdim = number of components in embedding vectors = cols in emb_matrix
# numf_conv = number o filters to be applied in convolutional layer
# ksize = kernel size of the convolutional
# psize = pool size of the pooling

def CNN2_pooling2_multilayer_perceptron(num_inputs,layers,words_in_vocab,
    vdim, \
                                        emb_matrix,numf_conv,ksize,psize):
    assert len(layers) > 0, \
        "Error, el número de capas ocultas debe ser mayor que 0"
    model = Sequential()
    model.add(Embedding(words_in_vocab, vdim, weights=[emb_matrix], \
                  input_length=num_inputs, trainable=False))
    # padding='same' makes a padding frame, so the output will be the
    # shame lenght as input
    model.add(Conv1D(filters=numf_conv, kernel_size=ksize, padding='same'
        , \
```

```
                    activation='relu'))
    model.add(MaxPooling1D(pool_size=psize))
    model.add(Conv1D(filters=numf_conv, kernel_size=ksize, padding='same'
        , \
                    activation='relu'))
    model.add(MaxPooling1D(pool_size=psize))
    model.add(Flatten())
    for n in layers:
        model.add(Dense(n, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                  metrics=['accuracy'])
    # model.summary()
    return model



# read train file
workdir = '/Users/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python/
    '
filename = workdir+'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'
filename_word2vec_model = workdir+'tourist_reviews_model.bin'
verbose_train = 0 #  Integer. 0, 1, or 2. Verbosity mode. 0 = silent,
                  #1 = progress bar, 2 = one line per epoch
num_epochs = 20  # num of epochs
nfiltersC_list = [16,32,64,128] # num of filters to apply in Convolution
kernel_s_list  = [2,3,4,5] # kernel size in convolution
pooling_s = 2 # pooling size in maxpool operation
one_conv = False # True to add only one pair Conv1D - Maxpool layer
                 # False adds two pairs Conv1D - Maxpool layers

input_size = 800 # sequece size. Cut or pad reviews to this size (in
    words)

# COMO HACER EL MODELO REPRODUCIBLE
# fix random seed for reproducibility
os.environ['PYTHONHASHSEED'] ='0'
```

```
np.random.seed(42) # para que numpy tenre aleatorios en un estado bien
    definido
rn.seed(12345)  # para que python parta de un estado bien definido
session_conf = tf.ConfigProto(intra_op_parallelism_threads=1 \
                              , inter_op_parallelism_threads=1)
tf.set_random_seed(1234)
sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
K.set_session(sess)

# https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=
    sharing
# 3000000 words in dictionary
# produce vector 300 floats
model_google_file='/Users/carlos/Desktop/GoogleNews-vectors-negative300.
    bin'

df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')


# extract comments and value (converted 1 for 'Good' and 0 for 'Bad')
# into two lists
list_of_reviews, y_train = df_train['comentario'], df_train['valor'].
    apply( \
                                  lambda x: 1 if x == 'Good' else 0)
test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                  lambda x: 1 if x == 'Good' else 0)
tokenizer = Tokenizer()
tokenizer.fit_on_texts(list_of_reviews)

# number of words in tokenizer
# we have to add 1 because index 0 is reserved with to encode the padded
# components, so there is one row more than number of words in tokenizer
nwords = len(tokenizer.word_index) + 1
# encode the reviews into integers using tokenizer
encoded_reviews = tokenizer.texts_to_sequences(list_of_reviews)
encoded_test_reviews = tokenizer.texts_to_sequences(test_reviews)
```

```
# convert list of reviews, in a list of sentences splitted into words
lor_in_words = [ text_to_word_sequence(review) for review in
    list_of_reviews]

# number of components of vectors
vector_dim = 300

#    print("Using Google pre-trained Word2Vec")
model_emb = word2vec.KeyedVectors.load_word2vec_format( \
                                    model_google_file, binary=True)

# Create a embedding matrix. rows = number of words in vocabulary + 1
# because there will be zeros (in padded components)
# cols = num of components of the vectors used in embedding
# initialize matriz with zeroes
embedding_matrix = zeros((nwords, vector_dim))
for word, i in tokenizer.word_index.items():
    if word in model_emb:
        vector = model_emb[word]
        embedding_matrix[i] = vector


x_train = pad_sequences(encoded_reviews, maxlen=input_size, \
                                    padding='post',truncating='post')
x_test = pad_sequences(encoded_test_reviews, maxlen=input_size, \
                                    padding='post',truncating='post')

print("Tensor x_train.ndim = ",x_train.ndim, " x_train.shape = ", \
                x_train.shape," x_train.dtype = ",x_train.dtype)
print("Tensor x_test.ndim = ",x_test.ndim, " x_test.shape = ", \
                x_test.shape," x_test.dtype = ",x_test.dtype)

for nfiltersC in nfiltersC_list:
    for kernel_s in kernel_s_list:
        if (one_conv):
            model_net = CNN_pooling_multilayer_perceptron(input_size
                ,[50],nwords,\
                        vector_dim,embedding_matrix,nfiltersC,
                            kernel_s,pooling_s)
```

```
        else:
            model_net = CNN2_pooling2_multilayer_perceptron(input_size
                ,[50],nwords,\
                            vector_dim,embedding_matrix,nfiltersC,
                                kernel_s,pooling_s)
        print("Nº filtros aplicados: ",nfiltersC,"Tamano kernel: ",
            kernel_s)
        model_net.summary()
        model_net.fit(x_train, y_train, epochs=num_epochs, verbose=
            verbose_train)
        loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
        print('Porcentaje de aciertos en el set de test: %f' %(acc*100))
        resultados = evaluate_model(model_net,df_evaluation,tokenizer,
            input_size)
        print_results(resultados)
        print ("
            ____________________________________________________________"
            )


# uncomment next section to print review codification example
"""
# model_net.layers[0] is first layer, embedding for this model
embedded_layer = Model(inputs=model_net.input,
                                    outputs=model_net.layers[0].output)
index = 0 # index to use as example
lista = ["The pool was very nice, it always was warm. \
         Food and staff are fantastic. I will recommend this hotel."]
encoded_lista = tokenizer.texts_to_sequences(lista)
encoded_lista2 = pad_sequences(encoded_lista, maxlen=input_size, \
                                  padding='post',truncating='post')
comentario_tok = text_to_word_sequence(lista[index])
x = encoded_lista2[index].reshape(1,input_size)
print("Muestra ",index," de ejemplo original -> token -> secuencia ->
    coded")
embedded_output = embedded_layer.predict(x)
print(lista[index])
print(comentario_tok)
print(encoded_lista2[index])
```

```
print(embedded_output)
print("Tamaño del vocabulario = ",nwords)
"""
```

A.15. Apéndice: Word2Vec_LSTM.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 20 20:01:59 2019

@author: carlos
"""
import os
import random as rn
import numpy as np
from keras import backend as K
import tensorflow as tf
import gensim.models.keyedvectors as word2vec
from keras.preprocessing.text import Tokenizer
import pandas as pd
from keras.preprocessing.sequence import pad_sequences
from numpy import zeros
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers.core import Dense
from keras.layers.embeddings import Embedding


################## MODELS DEFINITION ##############################
# function to evaluate a model
# dataframe contains two cols: review, label
# tk is the tokenizer to encode the reviewsç
# clean specifies if it has to clean or not the reviews to be predicted
# cod_mode. 'binary', 'count', 'tfidf' or 'freq'
# return values: list with
# Good reviews, Bad reviews, Good predictions hits, Bad predictions hits,
# Good predictions fails, Bad predictions fails
```

```
def evaluate_model(model,dataframe,tk,longest_review):

    f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    positivos = 0   # total 'Good' reviews
    negativos = 0   # total 'Bad' reviews
    a_positivos = 0  # total predicted as 'Good' and labeled as 'Good'
    a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'
    lol, y = dataframe['comentario'], dataframe['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)

    x_v = tk.texts_to_sequences(lol)
    # IMPORTANT: first version of this program used padding and trunc
    # post. I was not good for LSTM training (netork does not fit
    # well). I has to change this to pre
    x = pad_sequences(x_v,maxlen=longest_review,padding='pre' \
                      ,truncating='pre')
    print("Tensor␣x.ndim␣=␣",x.ndim, "␣x.shape␣=␣",x.shape \
                      ,"␣vector.dtype␣=␣",x.dtype)

    tam = len(dataframe)
    for i in list(range(tam)):
        vector = x[i].reshape(1,len(x[i]))
        prediction = model.predict(vector, verbose=0)
        prediction = int(round(prediction[0][0]))
        if (y[i] == 1):
            if (prediction == 1): a_positivos = a_positivos + 1
            else: f_negativos = f_negativos + 1
            positivos = positivos + 1
        else:
            if (prediction == 0): a_negativos = a_negativos + 1
            else: f_positivos = f_positivos + 1
            negativos = negativos + 1

    return [positivos,negativos,a_positivos,a_negativos,f_positivos \
            ,f_negativos]

# print results calculated in evaluate_model
```

```
def print_results(results):
    print("Comentarios positivos = ",results[0])
    print("Comentarios negativos = ",results[1])
    print("Aciertos en positivos = ",results[2])
    print("Aciertos en negativos = ",results[3])
    print("Fallos en positivos = ",results[4])
    print("Fallos en negativos = ",results[5])
    print("Porcentaje aciertos = ", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              "%")
    print("Porcentaje fallos = ", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
              "%")



# define a model with a Convolutional layer -> Maxpool layer
# num_inputs dimension of the input vector to the network
# layers is list. layers.size is number of hidden layers after maxpool
# layer[i] specifies number o neurons of hidden layer i+1
# words_in_vocab number or word in the used vocabulary = rows in
    emb_matrix
# vdim = number of components in embedding vectors = cols in emb_matrix
# numf_conv = number o filters to be applied in convolutional layer
# ksize = kernel size of the convolutional
# psize = pool size of the pooling

def LSTM_multilayer_perceptron(num_inputs,layers,words_in_vocab,vdim, \
                                        emb_matrix,ncells):
    assert len(layers) > 0, \
        "Error, el número de capas ocultas debe ser mayor que 0"
    model = Sequential()
    model.add(Embedding(words_in_vocab, vdim, weights=[emb_matrix], \
                  input_length=num_inputs, trainable=False))
    model.add(LSTM(ncells))
    for n in layers:
        model.add(Dense(n, activation='relu'))
```

```
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                  metrics=['accuracy'])
    #model.summary()
    return model



# read train file
workdir = '/Users/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python/
    '
filename = workdir+'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'
filename_word2vec_model = workdir+'tourist_reviews_model.bin'
verbose_train = 1 #  Integer. 0, 1, or 2. Verbosity mode. 0 = silent,
                  #1 = progress bar, 2 = one line per epoch
num_epochs = 30  # num of epochs
nLSTM_cells_list = [30,50,100] # list of num of LSTM cells to aply to
    LSTM layer


input_size = 800 # sequece size. Cut or pad reviews to this size (in
    words)

# COMO HACER EL MODELO REPRODUCIBLE
# fix random seed for reproducibility
os.environ['PYTHONHASHSEED'] ='0'
np.random.seed(42) # para que numpy tenre aleatorios en un estado bien
    definido
rn.seed(12345)  # para que python parta de un estado bien definido
session_conf = tf.ConfigProto(intra_op_parallelism_threads=1 \
                              , inter_op_parallelism_threads=1)
tf.set_random_seed(1234)
sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
K.set_session(sess)

# https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=
    sharing
```

```
# 3000000 words in dictionary
# produce vector 300 floats
model_google_file='/Users/carlos/Desktop/GoogleNews-vectors-negative300.
    bin'

df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')

df_train = df_train.sample(frac=1).reset_index(drop=True)

# extract comments and value (converted 1 for 'Good' and 0 for 'Bad')
# into two lists
list_of_reviews, y_train = df_train['comentario'], df_train['valor'].
    apply( \
                                    lambda x: 1 if x == 'Good' else 0)
test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                    lambda x: 1 if x == 'Good' else 0)
tokenizer = Tokenizer()
tokenizer.fit_on_texts(list_of_reviews)

# number of words in tokenizer
# we have to add 1 because index 0 is reserved with to encode the padded
# components, so there is one row more than number of words in tokenizer
nwords = len(tokenizer.word_index) + 1
# encode the reviews into integers using tokenizer
encoded_reviews = tokenizer.texts_to_sequences(list_of_reviews)
encoded_test_reviews = tokenizer.texts_to_sequences(test_reviews)

# number of components of vectors
vector_dim = 300

#    print("Using Google pre-trained Word2Vec")
model_emb = word2vec.KeyedVectors.load_word2vec_format( \
                                    model_google_file, binary=True)

# Create a embedding matrix. rows = number of words in vocabulary + 1
# because there will be zeros (in padded components)
```

```
# cols = num of components of the vectors used in embedding
# initialize matriz with zeroes
embedding_matrix = zeros((nwords, vector_dim))
for word, i in tokenizer.word_index.items():
    if word in model_emb:
        vector = model_emb[word]
        embedding_matrix[i] = vector

# IMPORTANT: first version of this program used padding and trunc
# post. I was not good for LSTM training (netork does not fit
# well). I has to change this to pre
x_train = pad_sequences(encoded_reviews, maxlen=input_size, \
                                    padding='pre',truncating='pre')
x_test = pad_sequences(encoded_test_reviews, maxlen=input_size, \
                                    padding='pre',truncating='pre')

print("Tensor x_train.ndim =",x_train.ndim, " x_train.shape =", \
                    x_train.shape," x_train.dtype =",x_train.dtype)
print("Tensor x_test.ndim =",x_test.ndim, " x_test.shape =", \
                    x_test.shape," x_test.dtype =",x_test.dtype)


for ncell in nLSTM_cells_list:

    model_net = LSTM_multilayer_perceptron(input_size,[50],nwords,\
                        vector_dim,embedding_matrix,ncell)
    model_net.summary()
    model_net.fit(x_train, y_train, epochs=num_epochs, \
                    verbose=verbose_train)
    loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
    print('Porcentaje de aciertos en el set de test: %f' %(acc*100))
    resultados = evaluate_model(model_net,df_evaluation,tokenizer,
        input_size)
    print_results(resultados)
    print ("-----------------------------------------------------------")


# uncomment next section to print review codification example
"""
```

```
# model_net.layers[0] is first layer, embedding for this model
embedded_layer = Model(inputs=model_net.input,
                                    outputs=model_net.layers[0].output)
index = 0 # index to use as example
lista = ["The pool was very nice, it always was warm. \
         Food and staff are fantastic. I will recommend this hotel."]
encoded_lista = tokenizer.texts_to_sequences(lista)
encoded_lista2 = pad_sequences(encoded_lista, maxlen=input_size, \
                                   padding='post',truncating='post')
comentario_tok = text_to_word_sequence(lista[index])
x = encoded_lista2[index].reshape(1,input_size)
print("Muestra ",index," de ejemplo original -> token -> secuencia ->
    coded")
embedded_output = embedded_layer.predict(x)
print(lista[index])
print(comentario_tok)
print(encoded_lista2[index])
print(embedded_output)
print("Tamaño del vocabulario = ",nwords)
"""
```

A.16. Apéndice: Embedding_CNNv2.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 20 20:01:59 2019

@author: carlos
"""
import os
import random as rn
import numpy as np
from keras import backend as K
import tensorflow as tf
from keras.preprocessing.text import text_to_word_sequence, Tokenizer
import pandas as pd
from keras.preprocessing.sequence import pad_sequences
```

```
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers.core import Dense
from keras.layers.embeddings import Embedding


################# MODELS DEFINITION ##############################
# function to evaluate a model
# dataframe contains two cols: review, label
# tk is the tokenizer to encode the reviewsç
# clean specifies if it has to clean or not the reviews to be predicted
# cod_mode. 'binary', 'count', 'tfidf' or 'freq'
# return values: list with
# Good reviews, Bad reviews, Good predictions hits, Bad predictions hits,
# Good predictions fails, Bad predictions fails

def evaluate_model(model,dataframe,tk,longest_review):

    f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    positivos = 0   # total 'Good' reviews
    negativos = 0   # total 'Bad' reviews
    a_positivos = 0  # total predicted as 'Good' and labeled as 'Good'
    a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'
    lol, y = dataframe['comentario'], dataframe['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)

    x_v = tk.texts_to_sequences(lol)
    # IMPORTANT: first version of this program used padding and trunc
    # post. I was not good for LSTM training (netork does not fit
    # well). I has to change this to pre
    x = pad_sequences(x_v,maxlen=longest_review,padding='pre' \
                      ,truncating='pre')
    print("Tensor x.ndim = ",x.ndim, " x.shape = ",x.shape \
                      ," vector.dtype = ",x.dtype)

    tam = len(dataframe)
    for i in list(range(tam)):
        vector = x[i].reshape(1,len(x[i]))
```

```
        prediction = model.predict(vector, verbose=0)
        prediction = int(round(prediction[0][0]))
        if (y[i] == 1):
            if (prediction == 1): a_positivos = a_positivos + 1
            else: f_negativos = f_negativos + 1
            positivos = positivos + 1
        else:
            if (prediction == 0): a_negativos = a_negativos + 1
            else: f_positivos = f_positivos + 1
            negativos = negativos + 1

    return [positivos,negativos,a_positivos,a_negativos,f_positivos \
            ,f_negativos]

# print results calculated in evaluate_model
def print_results(results):
    print("Comentarios positivos =",results[0])
    print("Comentarios negativos =",results[1])
    print("Aciertos en positivos =",results[2])
    print("Aciertos en negativos =",results[3])
    print("Fallos en positivos =",results[4])
    print("Fallos en negativos =",results[5])
    print("Porcentaje aciertos =", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              "%")
    print("Porcentaje fallos =", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
              "%")




# define a model with a Embedding + LSTM + Dense layers
# num_inputs dimension of the input vector to the network
# layers is list. layers.size is number of hidden layers after maxpool
# layer[i] specifies number o neurons of hidden layer i+1
# words_in_vocab number or word in the used vocabulary = rows in
    emb_matrix
```

```
# vdim = number of components in embedding vectors = cols in emb_matrix
# ncells = LSTM Cells or h dim

def LSTM_multilayer_perceptron(num_inputs,layers,words_in_vocab,vdim,
    ncells):
    assert len(layers) > 0, \
        "Error, el número de capas ocultas debe ser mayor que 0"
    model = Sequential()
    model.add( \
        Embedding(words_in_vocab,vdim,input_length=num_inputs, trainable=
            True))
    model.add(LSTM(ncells))
    for n in layers:
        model.add(Dense(n, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                  metrics=['accuracy'])
    #model.summary()
    return model



# read train file
workdir = '/Users/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python/
    '
filename = workdir+'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'
filename_word2vec_model = workdir+'tourist_reviews_model.bin'
verbose_train = 1 #  Integer. 0, 1, or 2. Verbosity mode. 0 = silent,
                  #1 = progress bar, 2 = one line per epoch
num_epochs = 30  # num of epochs
nLSTM_cells_list = [30,50,100] # list of num of LSTM cells to aply to
    LSTM layer


input_size = 800 # sequece size. Cut or pad reviews to this size (in
    words)
```

```
# COMO HACER EL MODELO REPRODUCIBLE
# fix random seed for reproducibility
os.environ['PYTHONHASHSEED'] ='0'
np.random.seed(42) # para que numpy tenre aleatorios en un estado bien
    definido
rn.seed(12345)  # para que python parta de un estado bien definido
session_conf = tf.ConfigProto(intra_op_parallelism_threads=1 \
                              , inter_op_parallelism_threads=1)
tf.set_random_seed(1234)
sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
K.set_session(sess)


df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')



# extract comments and value (converted 1 for 'Good' and 0 for 'Bad')
# into two lists
list_of_reviews, y_train = df_train['comentario'], df_train['valor'].
    apply( \
                                    lambda x: 1 if x == 'Good' else 0)
test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                    lambda x: 1 if x == 'Good' else 0)
tokenizer = Tokenizer()
tokenizer.fit_on_texts(list_of_reviews)

# number of words in tokenizer
# we have to add 1 because index 0 is reserved with to encode the padded
# components, so there is one row more than number of words in tokenizer
nwords = len(tokenizer.word_index) + 1

# encode the reviews into integers using tokenizer
encoded_reviews = tokenizer.texts_to_sequences(list_of_reviews)
encoded_test_reviews = tokenizer.texts_to_sequences(test_reviews)
```

```
# number of components of vectors
vector_dim = 300


# IMPORTANT: first version of this program used padding and trunc
# post. I was not good for LSTM training (netork does not fit
# well). I has to change this to pre

x_train = pad_sequences(encoded_reviews, maxlen=input_size, \
                                    padding='pre',truncating='pre')
x_test = pad_sequences(encoded_test_reviews, maxlen=input_size, \
                                    padding='pre',truncating='pre')
print("Tensor x_train.ndim =",x_train.ndim, " x_train.shape =", \
                    x_train.shape," x_train.dtype =",x_train.dtype)
print("Tensor x_test.ndim =",x_test.ndim, " x_test.shape =", \
                    x_test.shape," x_test.dtype =",x_test.dtype)


for ncell in nLSTM_cells_list:

    model_net = LSTM_multilayer_perceptron(input_size,[50],nwords,\
                        vector_dim,ncell)
    model_net.summary()
    model_net.fit(x_train, y_train, epochs=num_epochs, \
                    verbose=verbose_train)
    loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
    print('Porcentaje de aciertos en el set de test: %f' %(acc*100))
    resultados = evaluate_model(model_net,df_evaluation,tokenizer,
        input_size)
    print_results(resultados)
    print ("-----------------------------------------------------------")


# uncomment next section to print review codification example
"""
# model_net.layers[0] is first layer, embedding for this model
embedded_layer = Model(inputs=model_net.input,
                                    outputs=model_net.layers[0].output)
index = 0 # index to use as example
```

```
lista = ["The pool was very nice, it always was warm. \
         Food and staff are fantastic. I will recommend this hotel."]
encoded_lista = tokenizer.texts_to_sequences(lista)
encoded_lista2 = pad_sequences(encoded_lista, maxlen=input_size, \
                               padding='post',truncating='post')
comentario_tok = text_to_word_sequence(lista[index])
x = encoded_lista2[index].reshape(1,input_size)
print("Muestra ",index," de ejemplo original -> token -> secuencia ->
    coded")
embedded_output = embedded_layer.predict(x)
print(lista[index])
print(comentario_tok)
print(encoded_lista2[index])
print(embedded_output)
print("Tamaño del vocabulario = ",nwords)
"""
```

A.17. Apéndice: Embedding_CNN_LSTM.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 20 20:01:59 2019

@author: carlos
"""
import os
import random as rn
import numpy as np
from keras import backend as K
import tensorflow as tf
import gensim.models.keyedvectors as word2vec
from keras.preprocessing.text import Tokenizer
import pandas as pd
from keras.preprocessing.sequence import pad_sequences
from numpy import zeros
from keras.models import Sequential
from keras.layers import LSTM, Dropout
```

```
from keras.layers.core import Dense
from keras.layers.embeddings import Embedding
from keras.layers.convolutional import Conv1D, MaxPooling1D



################# MODELS DEFINITION ##############################
# function to evaluate a model
# dataframe contains two cols: review, label
# tk is the tokenizer to encode the reviewsç
# clean specifies if it has to clean or not the reviews to be predicted
# cod_mode. 'binary', 'count', 'tfidf' or 'freq'
# return values: list with
# Good reviews, Bad reviews, Good predictions hits, Bad predictions hits,
# Good predictions fails, Bad predictions fails

def evaluate_model(model,dataframe,tk,longest_review):

    f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    positivos = 0   # total 'Good' reviews
    negativos = 0   # total 'Bad' reviews
    a_positivos = 0  # total predicted as 'Good' and labeled as 'Good'
    a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'
    lol, y = dataframe['comentario'], dataframe['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)

    x_v = tk.texts_to_sequences(lol)
    # IMPORTANT: first version of this program used padding and trunc
    # post. I was not good for LSTM training (netork does not fit
    # well). I has to change this to pre
    x = pad_sequences(x_v,maxlen=longest_review,padding='pre' \
                      ,truncating='pre')
    print("Tensor x.ndim = ",x.ndim, " x.shape = ",x.shape \
                      ," vector.dtype = ",x.dtype)

    tam = len(dataframe)
    for i in list(range(tam)):
        vector = x[i].reshape(1,len(x[i]))
```

```
            prediction = model.predict(vector, verbose=0)
            prediction = int(round(prediction[0][0]))
            if (y[i] == 1):
                if (prediction == 1): a_positivos = a_positivos + 1
                else: f_negativos = f_negativos + 1
                positivos = positivos + 1
            else:
                if (prediction == 0): a_negativos = a_negativos + 1
                else: f_positivos = f_positivos + 1
                negativos = negativos + 1

        return [positivos,negativos,a_positivos,a_negativos,f_positivos \
                ,f_negativos]

# print results calculated in evaluate_model
def print_results(results):
    print("Comentarios positivos =",results[0])
    print("Comentarios negativos =",results[1])
    print("Aciertos en positivos =",results[2])
    print("Aciertos en negativos =",results[3])
    print("Fallos en positivos =",results[4])
    print("Fallos en negativos =",results[5])
    print("Porcentaje aciertos =", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              "%")
    print("Porcentaje fallos =", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
              "%")




# define a model with a Convolutional layer -> Maxpool layer
# num_inputs dimension of the input vector to the network
# layers is list. layers.size is number of hidden layers after maxpool
# layer[i] specifies number o neurons of hidden layer i+1
# words_in_vocab number or word in the used vocabulary = rows in
    emb_matrix
```

```
# vdim = number of components in embedding vectors = cols in emb_matrix
# numf_conv = number o filters to be applied in convolutional layer
# ksize = kernel size of the convolutional
# psize = pool size of the pooling

def CNN_LSTM_multilayer_perceptron(num_inputs,layers,words_in_vocab,vdim,
    \
                                   emb_matrix,ncells,numf_conv,ksize,
                                       psize, \
                                   apply_dropout, other_conv):
    assert len(layers) > 0, \
        "Error, el número de capas ocultas debe ser mayor que 0"
    model = Sequential()
    model.add(Embedding(words_in_vocab, vdim, weights=[emb_matrix], \
                  input_length=num_inputs, trainable=False))
    # padding='same' makes a padding frame, so the output will be the
    # shame lenght as input
    model.add(Conv1D(filters=numf_conv, kernel_size=ksize, padding='same'
        , \
                     activation='relu'))
    # 20% of units droped in training y Conv1D layer
    if apply_dropout:
        model.add(Dropout(0.2))
    model.add(MaxPooling1D(pool_size=psize))
    if other_conv:
        model.add(Conv1D(filters=numf_conv, kernel_size=ksize, padding='
            same', \
                     activation='relu'))
        # 20% of units droped in training y Conv1D layer
        if apply_dropout:
            model.add(Dropout(0.2))
        model.add(MaxPooling1D(pool_size=psize))
    model.add(LSTM(ncells))
    for n in layers:
        model.add(Dense(n, activation='relu'))
        # 20% of units droped in dense layer
        if apply_dropout:
            model.add(Dropout(0.2))
    model.add(Dense(1, activation='sigmoid'))
```

```
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                  metrics=['accuracy'])
    #model.summary()
    return model



# read train file
workdir = '/Users/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python/
    '
filename = workdir+'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'
filename_word2vec_model = workdir+'tourist_reviews_model.bin'
verbose_train = 1 #  Integer. 0, 1, or 2. Verbosity mode. 0 = silent,
                  #1 = progress bar, 2 = one line per epoch
num_epochs = 20  # num of epochs
nLSTM_cells_list = [50] # list of num of LSTM cells to aply to LSTM layer
nfiltersC = 64
kernel_s = 2
pooling_s = 2
applydrop = False   # apply dropout in hide layers

input_size = 800 # sequece size. Cut or pad reviews to this size (in
    words)

# COMO HACER EL MODELO REPRODUCIBLE
# fix random seed for reproducibility
os.environ['PYTHONHASHSEED'] ='0'
np.random.seed(42) # para que numpy tenre aleatorios en un estado bien
    definido
rn.seed(12345)  # para que python parta de un estado bien definido
session_conf = tf.ConfigProto(intra_op_parallelism_threads=1 \
                              , inter_op_parallelism_threads=1)
tf.set_random_seed(1234)
sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
K.set_session(sess)

# https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=
```

```
    sharing
# 3000000 words in dictionary
# produce vector 300 floats
model_google_file='/Users/carlos/Desktop/GoogleNews-vectors-negative300.
    bin'

df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')

df_train = df_train.sample(frac=1).reset_index(drop=True)

# extract comments and value (converted 1 for 'Good' and 0 for 'Bad')
# into two lists
list_of_reviews, y_train = df_train['comentario'], df_train['valor'].
    apply( \
                                    lambda x: 1 if x == 'Good' else 0)
test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                    lambda x: 1 if x == 'Good' else 0)
tokenizer = Tokenizer()
tokenizer.fit_on_texts(list_of_reviews)

# number of words in tokenizer
# we have to add 1 because index 0 is reserved with to encode the padded
# components, so there is one row more than number of words in tokenizer
nwords = len(tokenizer.word_index) + 1
# encode the reviews into integers using tokenizer
encoded_reviews = tokenizer.texts_to_sequences(list_of_reviews)
encoded_test_reviews = tokenizer.texts_to_sequences(test_reviews)

# number of components of vectors
vector_dim = 300

#    print("Using Google pre-trained Word2Vec")
model_emb = word2vec.KeyedVectors.load_word2vec_format( \
                                    model_google_file, binary=True)

# Create a embedding matrix. rows = number of words in vocabulary + 1
```

```
# because there will be zeros (in padded components)
# cols = num of components of the vectors used in embedding
# initialize matriz with zeroes
embedding_matrix = zeros((nwords, vector_dim))
for word, i in tokenizer.word_index.items():
    if word in model_emb:
        vector = model_emb[word]
        embedding_matrix[i] = vector

# IMPORTANT: first version of this program used padding and trunc
# post. I was not good for LSTM training (netork does not fit
# well). I has to change this to pre
x_train = pad_sequences(encoded_reviews, maxlen=input_size, \
                                    padding='pre',truncating='pre')
x_test = pad_sequences(encoded_test_reviews, maxlen=input_size, \
                                    padding='pre',truncating='pre')

print("Tensor x_train.ndim = ",x_train.ndim, " x_train.shape = ", \
                    x_train.shape," x_train.dtype = ",x_train.dtype)
print("Tensor x_test.ndim = ",x_test.ndim, " x_test.shape = ", \
                    x_test.shape," x_test.dtype = ",x_test.dtype)

for extraconv in [False,True]:
   for ncell in nLSTM_cells_list:

    model_net = CNN_LSTM_multilayer_perceptron(input_size,[50],nwords,\
                        vector_dim,embedding_matrix,ncell,nfiltersC,
                            kernel_s, \
                        pooling_s,applydrop,extraconv)
    model_net.summary()
    model_net.fit(x_train, y_train, epochs=num_epochs, \
                    verbose=verbose_train)
    loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
    print('Porcentaje de aciertos en el set de test: %f' %(acc*100))
    resultados = evaluate_model(model_net,df_evaluation,tokenizer,
        input_size)
    print_results(resultados)
    print ("————————————————————————————————————————————————————————")
```

A.18. Apéndice: Ensemble_weightedaverage.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


from keras.preprocessing.text import text_to_word_sequence, Tokenizer
import pandas as pd
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Flatten
from keras.layers.core import Dense
from keras.layers.embeddings import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
import gensim.models.keyedvectors as word2vec
from sklearn.model_selection import train_test_split, KFold
from sklearn.utils import resample
from numpy import mean, zeros
from numpy.linalg import norm
from scipy.optimize import differential_evolution
from itertools import product
import sys


# function to evaluate the loss function of a
# weighted average ensemble
# return 1 - prediction probability (real number between 0 and 1)
# of the weighted mean (dot product between ensemble preds and weights)
# evaluated with a test dataset
# to be minimized for an optimization method

def evaluate_loss_function(weights,models,xtest,ytest):
    # normalize weights between 0 and 1
    # transfor weight vector into a vector with norm 1
    result = norm(weights,1)
    if result == 0.0: # check if weights has all its components to zero
        weights1 = weights
    else:
```

```
        weights1 = weights / result

    totalacc = 0
    for i in range(len(weights1)):
        model = models[i]
        w = weights1[i]
        loss, acc = model.evaluate(xtest, ytest, verbose=0)
        totalacc = totalacc + w * acc
    return 1.0 - totalacc

def search_weights(models,xtest,ytest):
    # bounds for weights
    boundw = [(0.0, 1.0) for _ in range(len(models))]
    # arguments to the loss function
    search_arg = (models,xtest,ytest)
    # return a dict, where x key contains optimal set of weights
    result = differential_evolution(evaluate_loss_function, boundw, \
                                    search_arg, maxiter=1000, tol=1e-7)
    weights = result['x']
    result = norm(weights,1)
    if result == 0.0: # check if weights has all its components to zero
        weights1 = weights
    else:
        weights1 = weights / result
    return(weights1)


def search_weights_brute_force(models, xtest, ytest):
    # values that can take weights
    w = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
    num_models = len(models)
    best_acc = 0
    best_weights = list()
    for p in product(w,repeat=num_models):
        n = norm(p,1)
        if n == 0:
            continue
        p = p / n
        weight_mean = 0
```

```
        i = 0
        for model in models:
            loss, acc = model.evaluate(xtest, ytest, verbose=0)
            weight_mean = weight_mean + p[i] * acc
            i = i + 1
        if weight_mean >= best_acc:
            best_acc = weight_mean
            best_weights = p
    return best_acc,best_weights

# function to evaluate ensemble of models
# models = list of models in ensemble
# vector = input vector to the models
# return values:
# mean_pred = float [0,1] mean of models predictions
# mode_pred = mode of rounded predictions
# list_pred = list of individual predictions of the models

def evaluate_ensemble(models, weights, vector):
    list_pred = list()
    weight_mean = 0;
    i = 0;
    for model in models:
        prediction = model.predict(vector, verbose=0)
        weight_mean = weight_mean + weights[i] * prediction[0][0]
        i = i + 1
        list_pred.append(prediction[0][0])
    mean_pred = mean(list_pred)
    rounded_pred = [round(x) for x in list_pred]
    if (rounded_pred.count(0) >= rounded_pred.count(1)):
        mode_pred = 0
    else:
        mode_pred = 1
    return weight_mean, mean_pred,mode_pred,list_pred
```

```
################## MODELS DEFINITION ##############################
# function to evaluate a ensemble of models
# dataframe contains two cols: review, label
# tk is the tokenizer to encode the reviews
#
# return values: list with
# Good reviews, Bad reviews,
# for mean of predictions in ensemble: Good predictions hits,
# Bad predictions hits, Good predictions fails, Bad predictions fails
# for mode of predictions in ensemble: Good predictions hits,
# Bad predictions hits, Good predictions fails, Bad predictions fails

# list of accuracies of models in ensemble
def evaluate_model(models,weights,dataframe,tk,sequence_size):

    # Counter of predictions in ensemble using mean of models
    mean_f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    mean_f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    mean_a_positivos = 0  # total predicted as 'Good' and labeled as '
        Good'
    mean_a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'

    # Counter of predictions in ensemble using mode of models
    mode_f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    mode_f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    mode_a_positivos = 0  # total predicted as 'Good' and labeled as '
        Good'
    mode_a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'

    # Counter for weight menas predictions in ensemble
    wmean_f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad
        '
    wmean_f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good
        '
    wmean_a_positivos = 0  # total predicted as 'Good' and labeled as '
        Good'
    wmean_a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'
```

```
positivos = 0   # total 'Good' reviews
negativos = 0   # total 'Bad' reviews

lol, y = dataframe['comentario'], dataframe['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)

x_v = tk.texts_to_sequences(lol)
x = pad_sequences(x_v,maxlen=sequence_size,padding='post' \
                  ,truncating='post')
print("Tensor x.ndim = ",x.ndim, " x.shape = ",x.shape \
                  ," vector.dtype = ",x.dtype)

# evaluate each model
lofmodels_acc = list()
for model in models:
    _, acc = model.evaluate(x, y, verbose=0)
    lofmodels_acc.append(acc)

tam = len(dataframe)
for i in list(range(tam)):
    vector = x[i].reshape(1,len(x[i]))
    wmean,mean_ensemb,mode_prediction,lofpred = evaluate_ensemble(
        models, \
                                                    weights,
                                                        vector
                                                        )

    mean_prediction = int(round(mean_ensemb))
    wmean_prediction = int(round(wmean))
    if (y[i] == 1):
        if (mean_prediction == 1): mean_a_positivos =
            mean_a_positivos + 1
        else: mean_f_negativos = mean_f_negativos + 1
        if (mode_prediction == 1): mode_a_positivos =
            mode_a_positivos + 1
        else: mode_f_negativos = mode_f_negativos + 1
        if (wmean_prediction == 1): wmean_a_positivos =
            wmean_a_positivos+1
        else: wmean_f_negativos = wmean_f_negativos + 1
        positivos = positivos + 1
```

```
        else:
            if (mean_prediction == 0): mean_a_negativos =
                mean_a_negativos + 1
            else: mean_f_positivos = mean_f_positivos + 1
            if (mode_prediction == 0): mode_a_negativos =
                mode_a_negativos + 1
            else: mode_f_positivos = mode_f_positivos + 1
            if (wmean_prediction == 0): wmean_a_negativos =
                wmean_a_negativos+1
            else: wmean_f_positivos = wmean_f_positivos + 1
            negativos = negativos + 1

    return [positivos,negativos, mean_a_positivos,mean_a_negativos, \
            mean_f_positivos,mean_f_negativos, mode_a_positivos, \
            mode_a_negativos, mode_f_positivos, mode_f_negativos, \
            wmean_a_positivos,wmean_a_negativos, \
            wmean_f_positivos,wmean_f_negativos],lofmodels_acc

# print results calculated in evaluate_model
def print_results(results):
    print("Comentarios positivos =",results[0])
    print("Comentarios negativos =",results[1])
    print("Resulados usando la media de predicciones en ensemble:")
    print("[media ensemble] Aciertos en positivos =",results[2])
    print("[media ensemble] Aciertos en negativos =",results[3])
    print("[media ensemble] Fallos en positivos =",results[4])
    print("[media ensemble] Fallos en negativos =",results[5])
    print("[media ensemble] Porcentaje aciertos =", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              "%")
    print("[media ensemble] Porcentaje fallos =", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
              "%")
    print("Resulados usando la moda de predicciones en ensemble:")
    print("[moda ensemble] Aciertos en positivos =",results[6])
    print("[moda ensemble] Aciertos en negativos =",results[7])
    print("[moda ensemble] Fallos en positivos =",results[8])
    print("[moda ensemble] Fallos en negativos =",results[9])
    print("[moda ensemble] Porcentaje aciertos =", \
```

```
            round((results[6]+results[7])*100.0/(results[0]+results[1]),2),
                "%")
        print("[moda ensemble] Porcentaje fallos =", \
            round((results[8]+results[9])*100.0/(results[0]+results[1]),2),
                "%")
        print("Resulados usando la media ponderada de predicciones en
            ensemble:")
        print("[w-mean ensemble] Aciertos en positivos =",results[10])
        print("[w-mean ensemble] Aciertos en negativos =",results[11])
        print("[w-mean ensemble] Fallos en positivos =",results[12])
        print("[w-mean ensemble] Fallos en negativos =",results[13])
        print("[w-eman ensemble] Porcentaje aciertos =", \
            round((results[10]+results[11])*100.0/(results[0]+results[1])
                ,2),"%")
        print("[moda ensemble] Porcentaje fallos =", \
            round((results[12]+results[13])*100.0/(results[0]+results[1])
                ,2),"%")


# define a multilayer perceptron network with an embedding layer
# that has to be trained together with dense layers
# relu for activation function in hidden layers and sigmoid in output
# optimizer adam and binary_crossentropy as loss function
# output layer only have 1 neuron
# num_inputs dimension of the input vector to the network
# layers is list. layers.size is number of hidden layers.
# layer[i] specifies number o neurons of hidden layer i+1
# words_in_vocab number or word in the used vocabulary = rows in
    emb_matrix
# vdim = number of components in embedding vectors = cols in emb_matrix
def embedding_multilayer_perceptron(num_inputs,layers,words_in_vocab,vdim
    ):
    assert len(layers) > 0, \
        "Error, el número de capas ocultas debe ser mayor que 0"
    model = Sequential()
    # add and embedding layer with training
    model.add( \
        Embedding(words_in_vocab,vdim,input_length=num_inputs, trainable=
            True))
```

```
    model.add(Flatten())
    for n in layers:
        model.add(Dense(n, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                  metrics=['accuracy'])
    #model.summary()
    return model

# define a model with a 2 Convolutional layer -> Maxpool layer
# num_inputs dimension of the input vector to the network
# layers is list. layers.size is number of hidden layers after maxpool
# layer[i] specifies number o neurons of hidden layer i+1
# words_in_vocab number or word in the used vocabulary = rows in
    emb_matrix
# vdim = number of components in embedding vectors = cols in emb_matrix
# numf_conv = number o filters to be applied in convolutional layer
# ksize = kernel size of the convolutional
# psize = pool size of the pooling
# emb_goo usar embedding de google
# load = True if you have to load saved weights. file is filename of
    weights
def CNN2_pooling2_multilayer_perceptron(num_inputs,layers,words_in_vocab,
    vdim, \
                        emb_matrix,numf_conv,ksize,psize,emb_goo):
    assert len(layers) > 0, \
        "Error, el número de capas ocultas debe ser mayor que 0"
    model = Sequential()
    if emb_goo:
        model.add(Embedding(words_in_vocab, vdim, weights=[emb_matrix], \
                  input_length=num_inputs, trainable=False))
    else:
        # add and embedding layer with training
        model.add( \
        Embedding(words_in_vocab,vdim,input_length=num_inputs, trainable=
            True))
    # padding='same' makes a padding frame, so the output will be the
    # shame lenght as input
    model.add(Conv1D(filters=numf_conv, kernel_size=ksize, padding='same'
```

```
        , \
                        activation='relu'))
    model.add(MaxPooling1D(pool_size=psize))
    model.add(Conv1D(filters=numf_conv, kernel_size=ksize, padding='same'
        , \
                        activation='relu'))
    model.add(MaxPooling1D(pool_size=psize))
    model.add(Flatten())
    for n in layers:
        model.add(Dense(n, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                  metrics=['accuracy'])
    # model.summary()
    return model




# read train file
#workdir = '/Users/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python
    /'
workdir = '/scratch/cmartin/ensemble/'
filename = workdir+'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'


verbose_train = 0 #  Integer. 0, 1, or 2. Verbosity mode. 0 = silent,
                  #1 = progress bar, 2 = one line per epoch
num_epochs = 10  # num of epochs


ensemble_type = "randomsplit"  #use "kfold" for k-fold
                        #use "randomsplit" for random split
                        #use "bagging" for bagging
```

```
#MODEL_TYPE = 'MLP'
#USE_GOOGLE_EMBEDDING = False
MODEL_TYPE = 'CNN2'

if (MODEL_TYPE == 'CNN2'):
    USE_GOOGLE_EMBEDDING = True  # Use google embedding with CNN"
    nfiltersC = 64 # num of filters to apply in Convolution
    kernel_s = 2 # kernel size in convolution
    pooling_s = 2 # pooling size in maxpool operation
    one_conv = False # True to add only one pair Conv1D - Maxpool layer
                # False adds two pairs Conv1D - Maxpool layers

output_file = workdir+"Ensemble_"+ensemble_type+"_"+MODEL_TYPE+"_salida.
    txt"
sys.stdout = open(output_file, 'w')

# read train data set as a data frame with three columns:
# "index" "comentario" "valor"
df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')
# shuffle data frames
df_train = df_train.sample(frac=1).reset_index(drop=True)
df_test = df_test.sample(frac=1).reset_index(drop=True)
df_evaluation = df_evaluation.sample(frac=1).reset_index(drop=True)

# extract comments and value (converted 1 for 'Good' and 0 for 'Bad')
# into two lists
list_of_reviews, y_train = df_train['comentario'], df_train['valor'].
    apply( \
                                    lambda x: 1 if x == 'Good' else 0)
test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                    lambda x: 1 if x == 'Good' else 0)


# generate a unique list of reviews and  y_train to generate random
    splits
list_of_reviews = list_of_reviews.append(test_reviews)
```

```
y_train = y_train.append(y_test)
list_of_reviews.reset_index(inplace=True, drop=True)
y_train.reset_index(inplace=True, drop=True)



tokenizer = Tokenizer()
tokenizer.fit_on_texts(list_of_reviews)
# number of words in tokenizer
# it is important to add 1 to calculate the total number of words,
    because
# this parameter is used in the embedding layer to build the embedding
    matrix
# and it has to take in account that the index 0 (used as padding) needs
# an extra row in the embedding matrix
nwords = len(tokenizer.word_index) + 1
print("Tamaño del vocabulario + 1 =",nwords)
# encode the reviews into integers using tokenizer
encoded_reviews = tokenizer.texts_to_sequences(list_of_reviews)
encoded_test_reviews = tokenizer.texts_to_sequences(test_reviews)
# conver list of reviews, in a list of sentences splitted into words
lor_in_words = [ text_to_word_sequence(review) for review in
    list_of_reviews]

# number of components of vectors
vector_dim = 300
# calculate longest review, if you want to use it as input size of
    network
longest_review = max(len(review.split()) for review in list_of_reviews)


embedding_matrix = zeros((nwords, vector_dim))
if USE_GOOGLE_EMBEDDING:
    print("Using Google pre-trained Word2Vec")
    # https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?
        usp=sharing
    # 3000000 words in dictionary
    # produce vector 300 floats
    #model_google_file='/Users/carlos/Desktop/GoogleNews-vectors-
```

```
        negative300.bin'
    model_google_file = workdir+'GoogleNews-vectors-negative300.bin'
    model_emb = word2vec.KeyedVectors.load_word2vec_format( \
                                        model_google_file, binary=True)

    # Create a embedding matrix. rows = number of words in vocabulary + 1
    # because there will be zeros (in padded components)
    # cols = num of components of the vectors used in embedding
    # initialize matriz with zeroes

    for word, i in tokenizer.word_index.items():
        if word in model_emb:
            vector = model_emb[word]
            embedding_matrix[i] = vector


# define input size to 800 words for experiments
input_size = 800

# padd and truncate sequences to the specified size
x_train = pad_sequences(encoded_reviews, maxlen=input_size, \
                                    padding='post',truncating='post')

print("Tensor x_train.ndim =",x_train.ndim, "x_train.shape =", \
      x_train.shape,"x_train.dtype =",x_train.dtype)


# number of models to fit with the splits

list_of_acc = list()
list_of_models = list()
scores, members = list(), list()
print("Preparando entrenamiento del ensemble... con modelos ",MODEL_TYPE)
num_splits = 10

if (ensemble_type == "randomsplit"):
    print ("Usando random split con ",num_splits," particiones diferentes
        ")
    for i in range(num_splits):
```

```
    # split data train 10% for test
        x_train2, x_test, y_train2, y_test = train_test_split(x_train,
            y_train, \
                                                              test_size
                                                                  =0.10)
        if (MODEL_TYPE == 'MLP'):
            model_net = embedding_multilayer_perceptron(input_size,[50],
                \
                                                        nwords,
                                                            vector_dim)
        elif (MODEL_TYPE == 'CNN2'):
            model_net = CNN2_pooling2_multilayer_perceptron(input_size
                ,[50], \
                            nwords, vector_dim,embedding_matrix,nfiltersC
                                , \
                            kernel_s,pooling_s, USE_GOOGLE_EMBEDDING)
        print("entrenando␣modelo␣",i+1,"/",num_splits)
        model_net.fit(x_train2, y_train2, epochs=num_epochs, verbose=
            verbose_train)
        loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
        list_of_acc.append(acc)
        list_of_models.append(model_net)

elif (ensemble_type == "kfold"):
    # implementation kfold cross validation
    k = 10
    kfold = KFold(k, True, 1)
    # cross validation estimation of performance
    print ("Usando␣kfold␣cross␣validation␣con␣", k, "␣particiones")
    i = 1
    for part_train, part_test in kfold.split(x_train):
        x_train2, y_train2 = x_train[part_train], y_train[part_train]
        x_test, y_test = x_train[part_test], y_train[part_test]
        if (MODEL_TYPE == 'MLP'):
            model_net = embedding_multilayer_perceptron(input_size,[50],
                \
                                                        nwords,
                                                            vector_dim)
        elif (MODEL_TYPE == 'CNN2'):
```

```
            model_net = CNN2_pooling2_multilayer_perceptron(input_size
                ,[50], \
                            nwords, vector_dim,embedding_matrix,nfiltersC
                                , \
                            kernel_s,pooling_s, USE_GOOGLE_EMBEDDING)
        print("entrenando modelo ",i,"/",k)
        i = i +1
        model_net.fit(x_train2, y_train2, epochs=num_epochs, verbose=
            verbose_train)
        loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
        list_of_acc.append(acc)
        list_of_models.append(model_net)

elif (ensemble_type == "bagging"):
    # implementation of bootstrap agregation or bagging
    sample_percent = 80 # 80% data used as training with resampling
    training_size = round(len(x_train)* 80 / 100)
    print ("Usando bootstrap agregation or bagging con un ", \
           sample_percent,"% de muestras con remplazamiento para entrenar
               ", \
           "y ",num_splits," bootstraps diferentes")
    for i in range(num_splits):
        # valuate model
        indexes = [j for j in range(len(x_train))]
        indexes_train = resample(indexes, replace=True,n_samples=
            training_size)
        indexes_test = [k for k in indexes if k not in indexes_train]

        x_train2, y_train2 = x_train[indexes_train], y_train[
            indexes_train]
        x_test, y_test = x_train[indexes_test], y_train[indexes_test]
        if (MODEL_TYPE == 'MLP'):
            model_net = embedding_multilayer_perceptron(input_size,[50],
                \
                                                        nwords,
                                                            vector_dim)
        elif (MODEL_TYPE == 'CNN2'):
            model_net = CNN2_pooling2_multilayer_perceptron(input_size
                ,[50], \
```

```
                            nwords, vector_dim,embedding_matrix,nfiltersC
                                , \
                            kernel_s,pooling_s, USE_GOOGLE_EMBEDDING)
        print("entrenando modelo ",i+1,"/",num_splits)
        model_net.fit(x_train2, y_train2, epochs=num_epochs, verbose=
            verbose_train)
        loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
        list_of_acc.append(acc)
        list_of_models.append(model_net)

print("Finalizado entrenamiento. Calculando pesos para la media ponderada
    ...")
# get a 10% for tests the weights
x_train2, x_test, y_train2, y_test = train_test_split(x_train, y_train, \
                                                      test_size
                                                          =0.10)
pesos = search_weights(list_of_models,x_test,y_test)
prec,pesos = search_weights_brute_force(list_of_models,x_test,y_test)
print("La lista de pesos que consiguió la precisión ", prec,"es:")
print(pesos)
print("Calculando resultados")
result,lac = evaluate_model(list_of_models,pesos, df_evaluation,tokenizer
    , \
                            input_size)
print_results(result)
print ("-----------------------------------------------------------------")
print("Lista de precisión de los modelos con test subconjunto:",
    list_of_acc)
print ("-----------------------------------------------------------------")
print("Lista de precisión de los modelos con test independiente:",lac)
```

A.19. Apéndice: Ensemble_loaded_models_v2.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


import pandas as pd
```

```
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model
from numpy import mean
from numpy.linalg import norm
from scipy.optimize import differential_evolution
import sys
import pickle
from itertools import product


# function to evaluate the loss function of a
# weighted average ensemble
# return 1 - prediction probability (real number between 0 and 1)
# of the weighted mean (dot product between ensemble preds and weights)
# evaluated with a test dataset
# to be minimized for an optimization method

def evaluate_loss_function(weights,models,tokenizers,padtypes,
    sequence_size, \
                           reviews,ytest):
    # normalize weights between 0 and 1
    # transfor weight vector into a vector with norm 1
    result = norm(weights,1)
    if result == 0.0: # check if weights has all its components to zero
        weights1 = weights
    else:
        weights1 = weights / result

    totalacc = 0
    for i in range(len(weights1)):
        model = models[i]
        w = weights1[i]
        tk = tokenizers[i]
        padtype = padtypes[i]
        xtest_v =tk.texts_to_sequences(reviews)
        xtest = pad_sequences(xtest_v,maxlen=sequence_size,padding=
            padtype \
                      ,truncating=padtype)
        loss, acc = model.evaluate(xtest, ytest, verbose=0)
```

```
        totalacc = totalacc + w * acc
    return 1.0 - totalacc

def search_weights(models,tokenizers,padtypes,sequence_size, reviews,
    ytest):
    # bounds for weights
    boundw = [(0.0, 1.0) for _ in range(len(models))]
    # arguments to the loss function in adition to weights
    search_arg = (models,tokenizers,padtypes, sequence_size, reviews,
        ytest)
    # return a dict, where x key contains optimal set of weights
    result = differential_evolution(evaluate_loss_function, boundw, \
                                    search_arg, maxiter=1000, tol=1e-7)
    weights = result['x']
    result = norm(weights,1)
    if result == 0.0: # check if weights has all its components to zero
        weights1 = weights
    else:
        weights1 = weights / result
    return(weights1)


#def search_weights_brute_force(models, xtest, ytest):
def search_weights_brute_force(models,tokenizers,padtypes,sequence_size,
    \
                               reviews,ytest):
    # calculate xtest for every model
    xtest = list()
    for i in range(len(models)):
        model = models[i]
        tk = tokenizers[i]
        padtype = padtypes[i]
        x_v = tk.texts_to_sequences(reviews)
        x = pad_sequences(x_v,maxlen=sequence_size,padding=padtype \
                    ,truncating=padtype)
        xtest.append(x)
    # values that can take weights
    w = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
    num_models = len(models)
```

```
    best_acc = 0
    best_weights = list()
    for p in product(w,repeat=num_models):
        n = norm(p,1)
        if n == 0:
            continue
        p = p / n
        weight_mean = 0
        i = 0
        for model in models:
            loss, acc = model.evaluate(xtest[i], ytest, verbose=0)
            weight_mean = weight_mean + p[i] * acc
            i = i + 1
        if weight_mean >= best_acc:
            best_acc = weight_mean
            best_weights = p
    return best_acc,best_weights



# function to evaluate ensemble of models
# models = list of models in ensemble
# tokenizers contains list of tokenikers. tokenizers[i] is the
# the tokenizer used with models[i]
# padtypes list of pad types used in models.
# review is text to be evaluated. It has to be coded using correspondig
# tokenizer and padding
# return values:
# weight_mean weighted_mean
# mean_pred = float [0,1] mean of models predictions
# mode_pred = mode of rounded predictions
# list_pred = list of individual predictions of the models

def evaluate_ensemble(models, tokenizers, padtypes, \
                      weights, review, sequence_size):
    list_pred = list()
    weight_mean = 0;
    for i in range(len(models)):
```

```
        model = models[i]
        tk = tokenizers[i]
        padtype = padtypes[i]
        x_v = tk.texts_to_sequences([review])
        x = pad_sequences(x_v,maxlen=sequence_size,padding=padtype \
                    ,truncating=padtype)
        vector = x[0].reshape(1,len(x[0]))
        prediction = model.predict(vector, verbose=0)
        weight_mean = weight_mean + weights[i] * prediction[0][0]
        i = i + 1
        list_pred.append(prediction[0][0])
    mean_pred = mean(list_pred)
    rounded_pred = [round(x) for x in list_pred]
    if (rounded_pred.count(0) >= rounded_pred.count(1)):
        mode_pred = 0
    else:
        mode_pred = 1
    return weight_mean, mean_pred,mode_pred,list_pred




################## MODELS DEFINITION ##############################
# function to evaluate a ensemble of models
# dataframe contains two cols: review, label
# models contains list of models to evaluate
# tokenizers contains list of tokenikers. tokenizers[i] is the
# the tokenizer used with models[i]
# padtypes list of pad types used in models.
# return values: list with
# Good reviews, Bad reviews,
# for mean of predictions in ensemble: Good predictions hits,
# Bad predictions hits, Good predictions fails, Bad predictions fails
# for mode of predictions in ensemble: Good predictions hits,
# Bad predictions hits, Good predictions fails, Bad predictions fails

# list of accuracies of models in ensemble
```

```
def evaluate_model(models,tokenizers, padtypes, weights,dataframe, \
                   sequence_size):

    # Counter of predictions in ensemble using mean of models
    mean_f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    mean_f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    mean_a_positivos = 0  # total predicted as 'Good' and labeled as '
        Good'
    mean_a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'

    # Counter of predictions in ensemble using mode of models
    mode_f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    mode_f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    mode_a_positivos = 0  # total predicted as 'Good' and labeled as '
        Good'
    mode_a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'

    # Counter for weight menas predictions in ensemble
    wmean_f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad
        '
    wmean_f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good
        '
    wmean_a_positivos = 0  # total predicted as 'Good' and labeled as '
        Good'
    wmean_a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'

    positivos = 0   # total 'Good' reviews
    negativos = 0   # total 'Bad' reviews

    lol, y = dataframe['comentario'], dataframe['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)

    # evaluate each model
    lofmodels_acc = list()
    for i in range(len(models)):
        model = models[i]
        tk = tokenizers[i]
        padtype = padtypes[i]
        x_v = tk.texts_to_sequences(lol)
```

```
        x = pad_sequences(x_v,maxlen=sequence_size,padding=padtype \
                        ,truncating=padtype)
        _, acc = model.evaluate(x, y, verbose=0)
        lofmodels_acc.append(acc)

    tam = len(dataframe)
    for i in list(range(tam)):
        wmean,mean_ensemb,mode_prediction,lofpred = evaluate_ensemble( \
                        models, tokenizers, padtypes, weights,lol[i], \
                            sequence_size)
        mean_prediction = int(round(mean_ensemb))
        wmean_prediction = int(round(wmean))
        if (y[i] == 1):
            if (mean_prediction == 1): mean_a_positivos =
                mean_a_positivos + 1
            else: mean_f_negativos = mean_f_negativos + 1
            if (mode_prediction == 1): mode_a_positivos =
                mode_a_positivos + 1
            else: mode_f_negativos = mode_f_negativos + 1
            if (wmean_prediction == 1): wmean_a_positivos =
                wmean_a_positivos+1
            else: wmean_f_negativos = wmean_f_negativos + 1
            positivos = positivos + 1
        else:
            if (mean_prediction == 0): mean_a_negativos =
                mean_a_negativos + 1
            else: mean_f_positivos = mean_f_positivos + 1
            if (mode_prediction == 0): mode_a_negativos =
                mode_a_negativos + 1
            else: mode_f_positivos = mode_f_positivos + 1
            if (wmean_prediction == 0): wmean_a_negativos =
                wmean_a_negativos+1
            else: wmean_f_positivos = wmean_f_positivos + 1
            negativos = negativos + 1

    return [positivos,negativos, mean_a_positivos,mean_a_negativos, \
            mean_f_positivos,mean_f_negativos, mode_a_positivos, \
            mode_a_negativos, mode_f_positivos, mode_f_negativos, \
            wmean_a_positivos,wmean_a_negativos, \
```

```
            wmean_f_positivos,wmean_f_negativos],lofmodels_acc

# print results calculated in evaluate_model
def print_results(results):
    print("Comentarios positivos = ",results[0])
    print("Comentarios negativos = ",results[1])
    print("Resulados usando la media de predicciones en ensemble: ")
    print("[media ensemble] Aciertos en positivos = ",results[2])
    print("[media ensemble] Aciertos en negativos = ",results[3])
    print("[media ensemble] Fallos en positivos = ",results[4])
    print("[media ensemble] Fallos en negativos = ",results[5])
    print("[media ensemble] Porcentaje aciertos = ", \
        round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
            "%")
    print("[media ensemble] Porcentaje fallos = ", \
        round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
            "%")
    print("Resulados usando la moda de predicciones en ensemble: ")
    print("[moda ensemble] Aciertos en positivos = ",results[6])
    print("[moda ensemble] Aciertos en negativos = ",results[7])
    print("[moda ensemble] Fallos en positivos = ",results[8])
    print("[moda ensemble] Fallos en negativos = ",results[9])
    print("[moda ensemble] Porcentaje aciertos = ", \
        round((results[6]+results[7])*100.0/(results[0]+results[1]),2),
            "%")
    print("[moda ensemble] Porcentaje fallos = ", \
        round((results[8]+results[9])*100.0/(results[0]+results[1]),2),
            "%")
    print("Resulados usando la media ponderada de predicciones en
        ensemble: ")
    print("[w-mean ensemble] Aciertos en positivos = ",results[10])
    print("[w-mean ensemble] Aciertos en negativos = ",results[11])
    print("[w-mean ensemble] Fallos en positivos = ",results[12])
    print("[w-mean ensemble] Fallos en negativos = ",results[13])
    print("[w-eman ensemble] Porcentaje aciertos = ", \
        round((results[10]+results[11])*100.0/(results[0]+results[1])
            ,2),"%")
    print("[moda ensemble] Porcentaje fallos = ", \
        round((results[12]+results[13])*100.0/(results[0]+results[1])
```

```
                ,2),"%")


# read train file
#workdir = '/Users/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python
    /'
#workdir = '/scratch/cmartin/ensemble/'
workdir = '/scratch/cmartin/ensembles/'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'
ensemble_type = "Loaded_Models"  #use "kfold" for k-fold
                        #use "randomsplit" for random split
                        #use "bagging" for bagging



output_file = workdir+"Ensemble_"+ensemble_type+"_salida.txt"
sys.stdout = open(output_file, 'w')

df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')
# shuffle data frames
df_evaluation = df_evaluation.sample(frac=1).reset_index(drop=True)

df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)


# define input size to 800 words for experiments
input_size = 800
list_of_models = list()
list_of_tokenizers = list()

# loading files with models and tokenizers
#model_files = ['CNN2_W2VGoogleEmb_only_train_dataset_yelp_model.h5', \
#               'CNN2_W2VGoogleEmb_only_train_dataset_amazon_model.h5']

#tokenizer_files=['CNN2_W2VGoogleEmb_only_train_dataset_yelp_tokenizer.
```

```
    pickle',\
#             'CNN2_W2VGoogleEmb_only_train_dataset_amazon_tokenizer.
    pickle']

model_files = ['CNN2_W2VGoogleEmb_only_train_dataset_yelp_model.h5',
               'CNN2_W2VGoogleEmb_only_train_dataset_amazon_model.h5',
               'CNN2_EmbLayer_filters_64_kernelsize_2_model.h5',
               'CNN2_GoogleEmb_filters_64_kernelsize_2_model.h5',
               'CNN_GoogleEmb_filters_64_kernelsize_2_model.h5',
               'LSTM_Emblayer_ncells_50_model.h5',
               'LSTM_GoogleEmb_ncells_50_model.h5']

tokenizer_files=['CNN2_W2VGoogleEmb_only_train_dataset_yelp_tokenizer.
    pickle',
                 'CNN2_W2VGoogleEmb_only_train_dataset_amazon_tokenizer.
                     pickle',
                 'CNN2_EmbLayertokenizer.pickle',
                 'CNN2_GoogleEmb_tokenizer.pickle',
                 'CNN_GoogleEmb_tokenizer.pickle',
                 'LSTM_Emblayer_tokenizer.pickle',
                 'LSTM_GoogleEmb_tokenizer.pickle',]

list_of_padtypes =  ['post',
                     'post',
                     'post',
                     'post',
                     'post',
                     'pre',
                     'pre']

for i in range(len(model_files)):
        tk_file = workdir+tokenizer_files[i]
        with open(tk_file, 'rb') as handle:
            tokenizer = pickle.load(handle)
        list_of_tokenizers.append(tokenizer)
        model_f = workdir+model_files[i]
        print("Cargando␣modelo:␣",model_files[i])
        model = load_model(model_f)
        list_of_models.append(model)
```

```
#print("Calculando pesos para la media ponderada ...")
#pesos = search_weights(list_of_models,list_of_tokenizers,
    list_of_padtypes, \
#                    input_size, test_reviews,y_test)

# Pesos asignados manualmente
#print("Pesos asignados de forma manual ...")
#pesos = [0.3, 0.15, 0.15, 0.1, 0.1, 0.1, 0.1]
#print("La lista de pesos es: ")
#print(pesos)

# Pesos calculados por fuerza bruta
acc,pesos = search_weights_brute_force(list_of_models,list_of_tokenizers,
     \
                            list_of_padtypes,input_size, test_reviews,
                                y_test)
print("Pesos calculados por fuerza bruta")
print("Los pesos que han obtenido una precisión de",acc, " fueron:")
print(pesos)
print("Calculando resultados")

result,lac = evaluate_model(list_of_models,list_of_tokenizers, \
                            list_of_padtypes,pesos,df_evaluation,
                                input_size)
print_results(result)

print ("--------------------------------------------------------------------")
print("Lista de precisión de los modelos con test independiente: ",lac)
```

A.20. Apéndice: Stacking_ensemble.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


import pandas as pd
```

```
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model
from keras.models import Sequential
from keras.layers.core import Dense
import numpy as np
import sys
import pickle




# function to evaluate ensemble of models
# models = list of models in ensemble
# tokenizers contains list of tokenikers. tokenizers[i] is the
# the tokenizer used with models[i]
# padtypes list of pad types used in models.
# review is text to be evaluated. It has to be coded using correspondig
# tokenizer and padding
# return values:
# list_pred = list of individual predictions of the models

def evaluate_ensemble(models, tokenizers, padtypes, review, sequence_size
    ):
    list_pred = list()
    for i in range(len(models)):
        model = models[i]
        tk = tokenizers[i]
        padtype = padtypes[i]
        x_v = tk.texts_to_sequences([review])
        x = pad_sequences(x_v,maxlen=sequence_size,padding=padtype \
                    ,truncating=padtype)
        vector = x[0].reshape(1,len(x[0]))
        prediction = model.predict(vector, verbose=0)
        list_pred.append(prediction[0][0])
    return list_pred

def MLP(num_neurons,num_inputs):
    model = Sequential()
```

```
    model.add(Dense(num_neurons,input_shape=(num_inputs,),activation='
        relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                  metrics=['accuracy'])
    model.summary()
    return model
################## MODELS DEFINITION ##############################
# function to evaluate a model

# tk is the tokenizer to encode the reviewsç
# clean specifies if it has to clean or not the reviews to be predicted
# cod_mode. 'binary', 'count', 'tfidf' or 'freq'
# return values: list with
# Good reviews, Bad reviews, Good predictions hits, Bad predictions hits,
# Good predictions fails, Bad predictions fails

def evaluate_model(model,reviews,y,models,tokenizers, padtypes, \
                   sequence_size):
    f_positivos = 0 # total predicted as 'Good' and labeled as 'Bad'
    f_negativos = 0 # total predicted as 'Bad' and labeled as 'Good'
    positivos = 0   # total 'Good' reviews
    negativos = 0   # total 'Bad' reviews
    a_positivos = 0  # total predicted as 'Good' and labeled as 'Good'
    a_negativos = 0 # total predicted as 'Bad' and labeled as 'Bad'

    tam = len(reviews)
    for i in list(range(tam)):
        x = np.array(evaluate_ensemble(models,tokenizers,padtypes,reviews
            [i], \
                                       sequence_size))
        vector = x.reshape(1,len(x))
        prediction = model.predict(vector, verbose=0)
        prediction = int(round(prediction[0][0]))
        if (y[i] == 1):
            if (prediction == 1): a_positivos = a_positivos + 1
            else: f_negativos = f_negativos + 1
            positivos = positivos + 1
        else:
```

```
            if (prediction == 0): a_negativos = a_negativos + 1
            else: f_positivos = f_positivos + 1
            negativos = negativos + 1

    return [positivos,negativos,a_positivos,a_negativos,f_positivos \
            ,f_negativos]

# print results calculated in evaluate_model
def print_results(results):
    print("Comentarios positivos =",results[0])
    print("Comentarios negativos =",results[1])
    print("Aciertos en positivos =",results[2])
    print("Aciertos en negativos =",results[3])
    print("Fallos en positivos =",results[4])
    print("Fallos en negativos =",results[5])
    print("Porcentaje aciertos =", \
          round((results[2]+results[3])*100.0/(results[0]+results[1]),2),
              "%")
    print("Porcentaje fallos =", \
          round((results[4]+results[5])*100.0/(results[0]+results[1]),2),
              "%")



# read train file
#workdir = '/Users/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python
    /'
#workdir = '/scratch/cmartin/ensemble/'
workdir = '/home/cmartin/ensembles/'
filename = workdir + 'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'
ensemble_type = "separated_stacking"



output_file = workdir+"Ensemble_"+ensemble_type+"_mod_con_transfer_salida
    .txt"
```

```
sys.stdout = open(output_file, 'w')

df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
# shuffle data frames
df_evaluation = df_evaluation.sample(frac=1).reset_index(drop=True)
df_test = df_test.sample(frac=1).reset_index(drop=True)
df_train = df_train.sample(frac=1).reset_index(drop=True)

train_reviews, y_train = df_train['comentario'], df_train['valor'].apply(
    \
                                lambda x: 1 if x == 'Good' else 0)

test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)

eval_reviews, y_eval = df_evaluation['comentario'], df_evaluation['valor'
    ].apply( \
                                lambda x: 1 if x == 'Good' else 0)

verbose_train = 0 #  Integer. 0, 1, or 2. Verbosity mode. 0 = silent,
                  #1 = progress bar, 2 = one line per epoch

num_epochs = 100  # num of epochs

# define input size to 800 words for experiments
input_size = 800
list_of_models = list()
list_of_tokenizers = list()



# loading files with models and tokenizers

model_files = ['CNN2_W2VGoogleEmb_only_train_dataset_yelp_model.h5',
               'CNN2_W2VGoogleEmb_only_train_dataset_amazon_model.h5',
               'CNN2_EmbLayer_filters_64_kernelsize_2_model.h5',
```

```
                'CNN2_GoogleEmb_filters_64_kernelsize_2_model.h5',
                'CNN_GoogleEmb_filters_64_kernelsize_2_model.h5',
                'LSTM_Emblayer_ncells_50_model.h5',
                'LSTM_GoogleEmb_ncells_50_model.h5']

tokenizer_files=['CNN2_W2VGoogleEmb_only_train_dataset_yelp_tokenizer.
    pickle',
                 'CNN2_W2VGoogleEmb_only_train_dataset_amazon_tokenizer.
                     pickle',
                 'CNN2_EmbLayertokenizer.pickle',
                 'CNN2_GoogleEmb_tokenizer.pickle',
                 'CNN_GoogleEmb_tokenizer.pickle',
                 'LSTM_Emblayer_tokenizer.pickle',
                 'LSTM_GoogleEmb_tokenizer.pickle',]
list_of_padtypes =  ['post',
                     'post',
                     'post',
                     'post',
                     'post',
                     'pre',
                     'pre']

# load all models and tokenizers
for i in range(len(model_files)):
        tk_file = workdir+tokenizer_files[i]
        with open(tk_file, 'rb') as handle:
            tokenizer = pickle.load(handle)
        list_of_tokenizers.append(tokenizer)
        model_f = workdir+model_files[i]
        print("Cargando modelo:",model_files[i])
        model = load_model(model_f)
        list_of_models.append(model)


print("Generando training set para el learner ...")
# input set for meta learner netwrok
x_train_learner = list()
for review in train_reviews:
    preds = evaluate_ensemble(list_of_models, list_of_tokenizers, \
```

```
                            list_of_padtypes, review, input_size)
    x_train_learner.append(preds)

print("Generando␣test␣set␣para␣el␣learner␣...")
x_test_learner = list()
for review in test_reviews:
    preds = evaluate_ensemble(list_of_models, list_of_tokenizers, \
                            list_of_padtypes, review, input_size)
    x_test_learner.append(preds)


x_train = np.array(x_train_learner)
x_test = np.array(x_test_learner)



print("Tensor␣x_train.ndim␣=␣",x_train.ndim, "␣x_train.shape␣=␣", \
          x_train.shape,"␣x_train.dtype␣=␣",x_train.dtype)
print("Tensor␣x_test.ndim␣=␣",x_test.ndim, "␣x_test.shape␣=␣", \
          x_test.shape,"␣x_test.dtype␣=␣",x_test.dtype)
# MLP. input predictions of models
model_net = MLP(30,len(list_of_models))
model_net.fit(x_train, y_train, epochs=num_epochs, verbose=verbose_train)

loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
print('Porcentaje␣de␣aciertos␣en␣el␣set␣de␣test:␣%f' %(acc*100))

resultados = evaluate_model(model_net,eval_reviews,y_eval, list_of_models
    , \
                            list_of_tokenizers, list_of_padtypes,
                                input_size)
print_results(resultados)
print ("————————————————————————————————————————————————————————————")
```

A.21. Apéndice: Integrated_Stacking_ensemble_v2.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


import pandas as pd
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model, Model
from keras.layers.core import Dense
from keras.layers.merge import concatenate
from keras.utils import plot_model
import sys
import pickle



# read train file
#workdir = '/Users/carlos/Dropbox/docencia/tesis/CAPITULOS/codigos_python
    /'
workdir = '/scratch/cmartin/ensembles/'
#workdir = '/home/cmartin/ensembles/'
#workdir = '/Users/carlos/ensembles/'
#workdir = '/home/carlos/ensembles/'
filename = workdir + 'train_file.csv'
filename_test = workdir+'test_file.csv'
filename_evaluation = workdir+'evaluation_file.csv'
ensemble_type = "integrated_stacking_retrained"

neurons = 20 # number of neurons of Multi Layer Perceptron
train_layers = True # set to False if you don't want to train loaded
    models

output_file = workdir+"Ensemble_"+ensemble_type+"_mod_full_salida.txt"
graph_file =  workdir+"Ensemble_"+ensemble_type+"_mod_full_graph.png"
ensemble_name = workdir+"Ensemble_"+ensemble_type+"_mod_full.h5"
ensemble_json = workdir+"Ensemble_"+ensemble_type+"_mod_full.json"
ensemble_weights=workdir+"Ensemble_"+ensemble_type+"_full_weights.h5"

sys.stdout = open(output_file, 'w')
```

```
df_evaluation = pd.read_csv(filename_evaluation, sep='\t', encoding='utf
    -8')
df_test = pd.read_csv(filename_test, sep='\t', encoding='utf-8')
df_train = pd.read_csv(filename, sep='\t', encoding='utf-8')
# shuffle data frames
df_evaluation = df_evaluation.sample(frac=1).reset_index(drop=True)
df_test = df_test.sample(frac=1).reset_index(drop=True)
df_train = df_train.sample(frac=1).reset_index(drop=True)

train_reviews, y_train = df_train['comentario'], df_train['valor'].apply(
    \
                                lambda x: 1 if x == 'Good' else 0)

test_reviews, y_test = df_test['comentario'], df_test['valor'].apply( \
                                lambda x: 1 if x == 'Good' else 0)

eval_reviews, y_eval = df_evaluation['comentario'], df_evaluation['valor'
    ].apply( \
                                lambda x: 1 if x == 'Good' else 0)

verbose_train = 2
 #  Integer. 0, 1, or 2. Verbosity mode. 0 = silent,
                #1 = progress bar, 2 = one line per epoch

num_epochs = 10  # num of epochs

# define input size to 800 words for experiments
input_size = 800


list_of_models = list()
list_of_tokenizers = list()



# loading files with models and tokenizers

model_files = ['CNN2_W2VGoogleEmb_only_train_dataset_yelp_model_v2.h5',
               'CNN2_W2VGoogleEmb_only_train_dataset_amazon_model_v2.h5',
```

```
                'CNN2_EmbLayer_filters_64_kernelsize_2_model.h5',
                'CNN2_GoogleEmb_filters_64_kernelsize_2_model.h5',
                'CNN_GoogleEmb_filters_64_kernelsize_2_model.h5',
                'LSTM_Emblayer_ncells_50_model_v2.h5',
                'LSTM_GoogleEmb_ncells_50_model.h5']

tokenizer_files=['CNN2_W2VGoogleEmb_only_train_dataset_yelp_tokenizer.
    pickle',
                 'CNN2_W2VGoogleEmb_only_train_dataset_amazon_tokenizer.
                     pickle',
                 'CNN2_EmbLayertokenizer.pickle',
                 'CNN2_GoogleEmb_tokenizer.pickle',
                 'CNN_GoogleEmb_tokenizer.pickle',
                 'LSTM_Emblayer_tokenizer.pickle',
                 'LSTM_GoogleEmb_tokenizer.pickle',]
list_of_padtypes =  ['post',
                     'post',
                     'post',
                     'post',
                     'post',
                     'pre',
                     'pre']


# load all models and tokenizers
for i in range(len(model_files)):
        tk_file = workdir+tokenizer_files[i]
        with open(tk_file, 'rb') as handle:
            tokenizer = pickle.load(handle)
        list_of_tokenizers.append(tokenizer)
        model_f = workdir+model_files[i]
        print("Cargando␣modelo:␣",model_files[i])
        model = load_model(model_f)

        # conver all layers in model into no trainables
        # also we have to change the name of the layers because
        # they have to be unique in the ensemble, so we will add
        # the number of the model in the ensemble
        for layer in model.layers:
```

```
            if not train_layers:
                layer.trainable = False
            layer.name = 'model_' + str(i+1) + '_' + layer.name
        model.compile(loss='binary_crossentropy', optimizer='adam', \
                metrics=['accuracy'])
        model.summary()
        list_of_models.append(model)


ensemble_input = [model.input for model in list_of_models]
ensemble_output = [model.output for model in list_of_models]
mlp_input = concatenate(ensemble_output)
mlp_hidden_layer = Dense(neurons, activation='relu')(mlp_input)
mlp_output_layer = Dense(1,activation='sigmoid')(mlp_hidden_layer)
model_net = Model(inputs=ensemble_input, outputs=mlp_output_layer)
plot_model(model_net, show_shapes=True, to_file=graph_file)
model_net.compile(loss='binary_crossentropy', optimizer='adam', \
                metrics=['accuracy'])


x_train = list()
x_test = list()
x_eval = list()
for i in range(len(list_of_models)):
   cr1 = list_of_tokenizers[i].texts_to_sequences(train_reviews)
   cr2 = list_of_tokenizers[i].texts_to_sequences(test_reviews)
   cr3 = list_of_tokenizers[i].texts_to_sequences(eval_reviews)
   x_tr = pad_sequences(cr1, maxlen=input_size, \
               padding=list_of_padtypes[i],truncating=list_of_padtypes[i
                   ])
   x_te = pad_sequences(cr2, maxlen=input_size, \
               padding=list_of_padtypes[i],truncating=list_of_padtypes[i
                   ])
   x_ev = pad_sequences(cr3, maxlen=input_size, \
               padding=list_of_padtypes[i],truncating=list_of_padtypes[i
                   ])
   x_train.append(x_tr)
   x_test.append(x_te)
   x_eval.append(x_ev)
```

```
model_net.fit(x_train,y_train, epochs=num_epochs, verbose=verbose_train)
loss, acc = model_net.evaluate(x_test, y_test, verbose=0)
print('Porcentaje de aciertos en el set de test: %f' %(acc*100))
model_net.save(ensemble_name)
model_json = model.to_json()
with open(ensemble_json, "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights(ensemble_weights)

print("Saved model to disk")
loss, acc = model_net.evaluate(x_eval, y_eval, verbose=0)
print('Porcentaje de aciertos en el set de evaluacion: %f' %(acc*100))
```

Bibliografía

Las siguientes referencias bibliográficas se presentan en orden alfabético por autor. Las referencias con más de un autor aparecen ordenadas en base al primero de los mismos.

M. Bradley y P. Lang. Affective norms for english words (anew): Instruction manual and affective ratings. *University of Florida: The Center for Research in Psychophysiology*, 1999.

J. Brownlee. Ensemble learning methods for deep learning neural networks, n.d. URL `https://machinelearningmastery.com/ensemble-methods-for-deep-learning-neural-networks/`.

J. Broß. A distant supervision method for product aspect extraction from customer reviews. págs. 339–346, 09 2013. DOI: 10.1109/ICSC.2013.65.

M. Cajamarca. Inteligencia artificial, aprendizaje automático y aprendizaje profundo., 2019. URL `https://planetachatbot.com/inteligencia-artificial-aprendizaje-automAtico-y-aprendizaje-profundo-862ca9`

A. Cantino y K. Maxwell. Selector gadget extension.

Cascading Style Sheets. CSS. Cascading Style Sheets., n.d. URL https://www.w3schools.com/css/.

D. C. Cire
csan, U. Meier, J. Masci, L. M. Gambardella y J. Schmidhuber. Flexible, high performance convolutional neural networks for image classification. En *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two*, págs. 1237–1242, 2011.

Colah. Understanding lstm networks, 2015. URL http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

C. Cortes y V. Vapnik. Support-vector networks. *Machine Learning*, 20(3): 273–297, Sep 1995. ISSN 1573-0565. DOI: 10.1023/A:1022627411411. URL https://doi.org/10.1023/A:1022627411411.

L. de Vries, S. Gensler y P. S. Leeflang. Popularity of Brand Posts on Brand Fan Pages: An Investigation of the Effects of Social Media Marketing. *Journal of Interactive Marketing*, 26(2):83–91, 5 2012. ISSN 10949968. DOI: 10.1016/j.intmar.2012.01.003. URL https://www.sciencedirect.com/science/article/pii/S1094996812000060http://linkinghub.elsevier.com/retrieve/pii/S1094996812000060.

T. G. Dipanjan Sarkar, Raghav Bali. *Hands-On Transfer Learning with Python*. Packt Publishing Ltd, 2018. ISBN ISBN 978-1-78883-130-7.

E. M.-B. L. Federico Alberto Pozzi, Elisabetta Fersini. *Sentiment Analisys in Social Networks*. Morgan Kaufmann, 2016. ISBN 9780128044124.

K. Fukushima. Neocognitron: a self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

M. G. Gallarza, I. G. Saura y H. C. García. Destination image: Towards a conceptual framework. *Annals of Tourism Research*, 29(1):56 – 78, 2002. ISSN 0160-7383. DOI: https://doi.org/10.1016/S0160-7383(01)00031-7. URL http://www.sciencedirect.com/science/article/pii/S0160738301000317.

K. G. M. George Kyriakides. *Hands-On Ensemble Learning with Python*. Packt Publishing Ltd, 2019.

X. Glorot, A. Bordes e Y. Bengio. Deep sparse rectifier neural networks. En *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, eds. G. Gordon, D. Dunson y M. Dudík, vol. 15 de *Proceedings of Machine Learning Research*, págs. 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL http://proceedings.mlr.press/v15/glorot11a.html.

Google word2vec Project. Google Word2Vec re-trained embedding, n.d. URL https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing.

T. Hennig-Thurau, K. P. Gwinner, G. Walsh y D. D. Gremler. Electronic word-of-mouth via consumer-opinion platforms: What motivates consumers to articulate themselves on the Internet? *Journal of Interactive Marketing*, 18(1):38–52, 1 2004. ISSN 10949968. DOI: 10.1002/dir.10073. URL https://www.sciencedirect.com/science/article/pii/S1094996804700961http://linkinghub.elsevier.com/retrieve/pii/S1094996804700961.

S. Hochreiter y J. Schmidhuber. Long short-term memory. *Neural Compu-*

tation, 9(8):1735–1780, Nov 1997. ISSN 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

S. Hochreiter, Y. Bengio, P. Frasconi y J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. En *A Field Guide to Dynamical Recurrent Neural Networks*, eds. S. C. Kremer y J. F. Kolen. IEEE Press, 2001.

IMDB. Imdb film reviews, n.d. URL `https://www.kaggle.com/utathya/imdb-review-dataset`.

A. M. Kaplan y M. Haenlein. Users of the world, unite! The challenges and opportunities of Social Media. *Business Horizons*, 53(1): 59–68, 1 2010. ISSN 00076813. DOI: 10.1016/j.bushor.2009.09.003. URL `https://www.sciencedirect.com/science/article/pii/S0007681309001232http://linkinghub.elsevier.com/retrieve/pii/S0007681309001232`.

Keras API. Keras python library, n.d. URL `https://keras.io`.

Y. Kim. Convolutional neural networks for sentence classification. En *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, págs. 1746–1751, Doha, Qatar, oct. 2014. Association for Computational Linguistics. DOI: 10.3115/v1/D14-1181. URL `https://www.aclweb.org/anthology/D14-1181`.

S. Kobayashi. Contextual augmentation: Data augmentation by words with paradigmatic relations. En *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, págs. 452–457, New Orleans, Louisiana, jun. 2018. Association for Computational

Linguistics. DOI: 10.18653/v1/N18-2072. URL https://www.aclweb.org/anthology/N18-2072.

A. Kowalczyk. Syncfusion, 2017.

P. F. Lazarsfeld y R. K. Merton. Friendship as social process: a substantive and methodological analysis. 1964.

Y. LeCun, L. Bottou, Y. Bengio y P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

B. Liu. *Sentiment Analysis and Opinion Mining*. Morgan & Claypool Publishers, 2012. ISBN 1608458849.

H. Mahmood. The softmax function, simplified., 2018. URL https://towardsdatascience.com/softmax-function-simplified-714068bf8156.

E. Marine-Roig. Los "travel blogsçomo objeto de estudio de la imagen percibida de un destino [travel blogs as objects of study of the perceived image of a destination]. 10 2010.

C. A. Martín, J. M. Torres, R. M. Aguilar y S. Diaz. Using deep learning to predict sentiments: Case study in tourism. *Complexity*, 2018:9, 2018. URL https://doi.org/10.1155/2018/7408431.

J. McAuley y J. Leskovec. Hidden factors and hidden topics: Understanding rating dimensions with review text. En *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, págs. 165–172, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2409-0. DOI: 10.

1145/2507157.2507163. URL http://doi.acm.org/10.1145/2507157.2507163.

C. McCormick. Word2vec tutorial - the skip-gram model., 2016. URL http://www.mccormickml.com.

T. Mikolov, I. Sutskever, K. Chen, G. Corrado y J. Dean. Distributed representations of words and phrases and their compositionality. En *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, págs. 3111–3119, USA, 2013. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999792.2999959.

C. A. Miranda, R. C. Rodriguez y R. Zagal-Flores. Arquitectura web para análisis de sentimientos en facebook con enfoque semántico. *Research in Computing Science*, 75:59–69, 2014.

MonkeyLearn Inc. Hotel review analysis. Monkey Learn., n.d. URL https://github.com/monkeylearn/hotel-review-analysis.

A. Newell. Perceptrons. an introduction to computational geometry. marvin minsky and seymour papert. m.i.t. press, cambridge, mass., 1969. vi + 258 pp., illus. cloth, 12;*paper*,4.95. *Science*, 165(3895):780–782, 1969. ISSN 0036-8075. DOI: 10.1126/science.165.3895.780. URL https://science.sciencemag.org/content/165/3895/780.

NLTK Project. NLTK Python Library, n.d. URL https://www.nltk.org.

Numpy Developers. Numpy Python Package, n.d. URL https://www.numpy.org/.

C. Nwankpa, W. Ijomah, A. Gachagan y S. Marshall. Activation functions: Comparison of trends in practice and research for deep learning. 11 2018.

B. Pan, T. Maclaurin y J. Crotts. Travel blogs and their implications for destination marketing. *Journal of Travel Research*, 46, 08 2007. DOI: 10.1177/0047287507302378.

S. J. Pan y Q. Yang. A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on*, 22(10):1345–1359, 2010.

Y. Park y D. Njite. Relationship between destination image and tourists' future behavior: Observations from jeju island, korea. *Asia Pacific Journal of Tourism Research*, 15(1):1–20, 2010. DOI: 10.1080/10941660903510024. URL `https://doi.org/10.1080/10941660903510024`.

J. Pennington, R. Socher y C. D. Manning. Glove: Global vectors for word representation. En *Empirical Methods in Natural Language Processing (EMNLP)*, págs. 1532–1543, 2014. URL `http://www.aclweb.org/anthology/D14-1162`.

D. Peppers y M. Rogers, Ph.D. *Managing Customer Relationships: A Strategic Framework: Third Edition*. 09 2016. DOI: 10.1002/9781119239833.

Radim Rehurek. Gensim Python Library, n.d. URL `https://radimrehurek.com/gensim/`.

A. Ratner, S. H. Bach, H. R. Ehrenberg, J. A. Fries, S. Wu y C. Ré. Snorkel: Rapid training data creation with weak supervision. *CoRR*, abs/1711.10160, 2017. URL `http://arxiv.org/abs/1711.10160`.

SciKitProject. Scikit-learn, n.d. URL https://scikit-learn.org.

SciPy. Scientific computing tools for python, n.d. URL https://www.scipy.org/.

Scrapinghub. Scrapy collaborative framework, n.d. URL https://scrapy.org/.

G. Sidorov, S. Miranda-Jiménez, F. Viveros-Jiménez, A. Gelbukh, N. Castro-Sánchez, F. Velásquez, I. Díaz-Rangel, S. Suárez-Guerra, A. Treviño y J. Gordon. Empirical study of machine learning based approach for opinion mining in tweets. En *Advances in Artificial Intelligence*, eds. I. Batyrshin y M. González Mendoza, págs. 1–14, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37807-2.

Snorkel. Snorkel metal, n.d. URL https://github.com/HazyResearch/metal#snorkel-metal.

M. D. Sotiriadis y C. van Zyl. Electronic word-of-mouth and online reviews in tourism services: the use of twitter by tourists. *Electronic Commerce Research*, 13(1):103–124, 3 2013. ISSN 1389-5753. DOI: 10.1007/s10660-013-9108-1. URL http://link.springer.com/10.1007/s10660-013-9108-1.

C. Souza. Kernel functions for machine learning applications, n.d. URL http://crsouza.com/2010/03/17/kernel-functions-for-machine-learning-applications/.

Statista. Statista, n.d. URL https://es.statista.com/.

R. Storn y K. Price. Differential evolution - a simple and efficient adaptive

scheme for global optimization over continuous spaces. *International Computer Science Institute-Publications TR*, 1995.

C. Strapparava y A. Valitutti. Wordnet-affect: an affective extension of wordnet. En *Proceedings of LREC*, vol. 4, págs. 1083–1086, 2004.

W. C. V.-S. K. Su P, Li G. Using distant supervision to augment manually annotated data for relation extraction. 2019. DOI: 10.1371/journal.pone.0216913.

J. Suttles y N. Ide. Distant supervision for emotion classification with discrete binary values. En *Computational Linguistics and Intelligent Text Processing*, ed. A. Gelbukh, págs. 121–136, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37256-8.

Term Frequency-Inverse Document Frequency. term frequency-inverse document frequency, n.d. URL `http://www.tfidf.com/`.

J. Torres. *Deep Learning: Introducción práctica con Keras (primera parte)*. 2018. ISBN 978-0-244-07895-9.

L. Weng. Learning word embedding., 2015. URL `https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html`.

B. Widrow y M. A. Lehr. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9): 1415–1442, Sep. 1990. ISSN 0018-9219. DOI: 10.1109/5.58323.

J. M. Wiebe, R. F. Bruce y T. P. O'Hara. Development and use of a gold-standard data set for subjectivity classifications, 1999.

D. H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241 – 259, 1992. ISSN 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(05)80023-1. URL http://www.sciencedirect.com/science/article/pii/S0893608005800231.

WTTC. World travel and toursim council, 2019. URL https://www.wttc.org/economic-impact/.

XPATH Expressions. XPATH. Path expression for XML., n.d. URL https://www.w3schools.com/xml/xml_xpath.asp.

YelpDataset. Yelp dataset, n.d. URL https://www.yelp.com/dataset.

Zhang y Tsm. Analysis of blogs and microblogs : a case study of chinese bloggers sharing their hong kong travel experiences, Jan 1970. URL http://hdl.handle.net/10397/24193.

X. Zhang. Amazon products reviews, n.d. URL https://www.kaggle.com/bittlingmayer/amazonreviews.

X. Zhang, J. Zhao e Y. LeCun. Character-level convolutional networks for text classification, 2015.