

**Universidad
Rey Juan Carlos**

Escuela Técnica Superior
de Ingeniería Informática

**Grado en Ingeniería del Software
Grado en Matemáticas**

Curso 2020-2021

Trabajo Fin de Grado

**PREDICCIÓN AUTOMÁTICA DE LA AFINIDAD
HACIA UNA PELÍCULA A PARTIR DE LA SINOPSIS**

**Autor: Jaime Morillo Leal
Tutor: Alfredo Cuesta Infante**

Agradecimientos

A la universidad, a los docentes y en especial a mi tutor por su ayuda y paciencia.

A toda mi familia, sobre todo a mis padres y hermana por apoyarme siempre y confiar en mi.

A mis amigos de toda la vida por estar a mi lado y a los nuevos que he ganado durante este grado.

Resumen

La Inteligencia Artificial es la revolución más importante de la tecnología en los últimos años, todo el mundo habla de ella y pocos la comprenden. Se encuentra al comprar por internet, cuando haces una búsqueda en Google, cuando le preguntas a tu asistente, cuando traduces..., en infinidad de tareas de tu día a día. Este gran desarrollo se debe en parte a una de sus ramas que más ha crecido, el aprendizaje automático, que permite que los ordenadores aprendan por su cuenta a dar resultados, sin necesidad de un programador que escriba las órdenes.

Dentro del aprendizaje automático nos encontramos con el aprendizaje profundo y las redes neuronales, que se emplean para extraer conocimiento de datos que no se encuentran habitualmente en forma de tablas, como son las imágenes o los textos. Gracias a ello podemos llegar a automatizar tareas que haría habitualmente un humano, como detectar objetos en imágenes o separar correos electrónicos.

En este trabajo de fin de grado se ha desarrollado una aplicación que da al usuario un porcentaje de afinidad a una cierta película dada, siendo 100% el máximo para una película que deberías ver y 0% para evitarla. Todo gracias a procesar la sinopsis de la película mediante redes neuronales y clasificarla. También estamos logrando la implementación de una técnica y metodología extrapolable a otros problemas, como son el análisis de sentimiento, la clasificación de temas o la detección de “fake news”.

Durante la memoria se detallarán todas las fases del desarrollo de la aplicación, partiendo de la planificación inicial, pasando por la explicación de conceptos, implementación y experimentos, hasta terminar con las conclusiones finales.

Palabras clave:

- Procesamiento del lenguaje natural (*Natural language processing*, NLP).
- Redes convolucionales y recurrentes.
- Stemming.
- Word Embedding.
- Sistema recomendador.
- Análisis de sentimiento.

Índice de contenidos

Índice de figuras	X
Índice de tablas	XI
1. Introducción	1
1.1. Contexto	1
1.2. Objetivos	2
1.3. Trabajos relacionados	3
1.3.1. Análisis de sentimiento y clasificación	4
1.3.2. Extracción de texto	4
1.3.3. Transformación de texto	5
2. Planificación	7
2.1. Requisitos	7
2.2. Estudio de alternativas	8
2.3. Metodología de desarrollo	8
2.3.1. Historias de usuario	9
2.3.2. Tablero Scrum	11
2.3.3. Pruebas	13
2.3.4. Diseño de la solución	14
3. Fundamentos	16
3.1. NLP	16
3.1.1. Bolsa de palabras	16
3.1.2. Codificación One-Hot	17
3.1.3. Codificación basada en índices	18
3.1.4. Word Embedding	19
3.2. Machine learning	21
3.2.1. Aprendizaje supervisado - clasificación	21
3.2.2. Redes neuronales	21
4. Descripción del problema	27
4.1. Datos	27

4.2. Técnicas aplicadas	28
4.3. Diseño	32
4.3.1. Diseño de la aplicación web	32
4.3.2. Diseño de la red neuronal	33
4.4. Implementación	35
4.5. Pruebas	37
5. Experimentos y resultados	39
5.1. Métricas	39
5.2. Análisis exploratorio	41
5.3. Red prototipo	43
5.4. Modificación ResNet	45
5.5. Modificación GRU Bidireccional	46
5.6. Modificación GRU + GRU	47
5.7. Resultados finales	48
6. Conclusiones	50
6.1. Logros	50
6.2. Trabajos futuros	51
Bibliografía	51
Apéndices	55
A. Códigos	57
A.1. Redes neuronales	57
A.2. Pruebas	63

Índice de figuras

1.1.	Análisis de sentimiento e intención (original)	4
1.2.	Ejemplo NER (original)	5
1.3.	Usos de GPT-3 (original)	6
2.1.	Tablero Trello durante el primer sprint	13
2.2.	Vista de la aplicación web	15
3.1.	Word embedding [1]	19
3.2.	Red neuronal (original)	22
3.3.	Actualización de la red (original)	22
3.4.	Red neuronal convolucional. [2]	23
3.5.	Convolución en textos	24
3.6.	Reducción en textos	24
3.7.	LSTM y GRU [3]	25
3.8.	Unidad Residual [4]	26
4.1.	Diagrama de la aplicación	32
4.2.	Estructura de la red neuronal	33
4.3.	Ejemplo de codificación del texto	34
5.1.	Matriz de confusión (original)	40
5.2.	Curva ROC	41
5.3.	Histograma	41
5.4.	Gráfico de densidad	41
5.5.	Wordcloud completo	42
5.6.	Wordcloud dislike	42
5.7.	Wordcloud like	42
5.8.	Condensación de textos	43
5.9.	Entrenamiento de la red original	43
5.10.	Matrices de confusión red prototipo	44
5.11.	Curva ROC red prototipo	44
5.12.	Matrices de confusión red ResNet	46
5.13.	Curva ROC red ResNet	46
5.14.	Matrices de confusión red GRU Bidireccional	47

5.15. Curva ROC red GRU Bidireccional.	47
5.16. Matrices de confusión red GRU + GRU.	48
5.17. Curva ROC red GRU + GRU.	48

Índice de tablas

3.1.	Bag of words encoding.	17
3.2.	One-hot encoding de palabras.	18
3.3.	One-hot encoding de documentos.	18
3.4.	Index-Based encoding.	19
4.1.	Métodos.	35
4.3.	Clases de test.	38
5.1.	Resultados sinopsis dataset (Mé: Métrica, Ex: Experimento). . . .	49
5.2.	Resultados IMDb dataset (Mé: Métrica, Ex: Experimento). . . .	49

1

Introducción

1.1. Contexto

Este proyecto nace en la idea de priorizar el contenido audiovisual que consumimos, de manera única y personalizada para cada usuario. Con el auge de las plataformas de vídeo en streaming la cantidad de películas y series que tenemos a nuestro alcance se ha disparado. Netflix, líderes del mercado en la actualidad, cuenta con 32.600 horas de vídeo según el periódico The Times [5], lo que vendría a ser el equivalente a 4 años de visionado ininterrumpido. También, la propia Netflix generó en 2020 más de 2.000 millones de dólares en beneficios [6], por lo que tener su contenido correctamente priorizado es esencial para mantener a sus usuarios.

Las empresas de distribución cuentan con sus algoritmos de recomendación que tratan de encontrar el contenido afín al usuario mediante el uso de múltiples factores que les permiten determinarlo [7], lo que vemos es que se encuentran limitados únicamente a lo visualizado dentro de su propia plataforma y no consultan al usuario por nada visto en el exterior de su ecosistema, como en el cine, la televisión o en la competencia. Por otro lado también podemos observar cierto sesgo en estos algoritmos; tienen preferencia por mostrar contenido producido por la propia casa, sus películas o series cobran más importancia a la hora de mostrarse al usuario, así como aquello publicado más recientemente o que está siendo más popular entre el resto de usuarios.

En este trabajo de fin de grado vamos a desarrollar una aplicación que usando las sinopsis de las películas, y mediante una red neuronal, sea capaz de ofrecer un

nivel de afinidad a las mismas para el usuario que en ese momento esté usando la aplicación, mostrará un valor entre 0 % y 100 %, siendo 100 % el máximo que se puede recomendar una película.

Los sistemas recomendadores en la actualidad funcionan mediante filtrado colaborativo. Aquí se plantea una alternativa que no dependa de terceros usuarios, y que se base únicamente en el contenido de los visto por el usuario anteriormente. El emplear los resúmenes de las cintas para dar la probabilidad de recomendación se debe a limitaciones de escalabilidad que existen en la actualidad, en cuanto al rendimiento, almacenamiento y complejidad del problema. Se podrían obtener más características útiles usando el guión de la película completa o incluso el propio contenido audiovisual del largometraje, lo que resulta bastante inviable a día de hoy.

Esta manera de utilizar las redes neuronales se podría extrapolar a otros ámbitos fuera del cine como la literatura, la industria de la moda o la alimentación, entre otros. Prácticamente en toda actividad donde se quiera dar una recomendación totalmente personalizada.

1.2. Objetivos

El objetivo del proyecto es el desarrollo, implementación y testing de una aplicación que cubra todas las etapas de nuestra red neuronal, desde el etiquetado previo de datos, pasando por el entrenamiento y acabando con la visualización de resultados al aplicarla sobre datos nunca vistos. La red debe permitir la obtención de un porcentaje de afinidad a una película basándose en la sinopsis.

Los objetivos técnicos generales son los siguientes:

1. Codificación y procesamiento previo del conjunto de sinopsis para que sea entendido por las redes neuronales, convirtiendo palabras en valores numéricos.
2. Creación de un modelo de NLP (Procesamiento del Lenguaje Natural), basado en redes neuronales, que resuelva un problema de clasificación binaria usando los textos codificados como entrada.
3. Desarrollo de una aplicación web que contenga:
 - La gestión del sistema de etiquetado de datos que permita un marcado previo de películas afines y no afines para el usuario.
 - Un entorno de visualización para localizar nuevas películas con su respectivo porcentaje de afinidad.

4. Tener una respuesta rápida y sin demora de la aplicación para que el usuario no tenga grandes tiempos de espera.
5. Implementación de un sistema de actualización para incorporar siempre las películas más recientes que se encuentran en el cine.

Además de los objetivos técnicos hay que detallar los objetivos académicos que se pretenden lograr mediante el trabajo. Un trabajo de fin de grado debe aportar nuevos conocimientos al alumno.

Los objetivos académicos que se lograrán con la realización de este trabajo son:

1. Aprendizaje de técnicas de procesamiento del lenguaje natural, como redes neuronales recurrentes o redes convolucionales.
2. Aprendizaje de técnicas de preprocesamiento de textos como stemming o lematization.
3. Aprendizaje de técnicas de codificación de textos como *Bag of words* (Bolsa de palabras), *One hot encoding* (Codificación en caliente) o *Word embeddings*.
4. Aprendizaje de conceptos de machine learning y deep learning como métricas, funciones de optimización, funciones de perdida o separación de muestras.
5. Aprendizaje de nuevas librerías para Python como Pandas, Scikit-learn, Keras, NLTK, Flask.
6. Aplicación de metodologías de Ingeniería del Software para el desarrollo, como identificación de requisitos previos, testing, seguimiento y evolución del proceso.

1.3. Trabajos relacionados

El NLP es uno de los campos del aprendizaje automático con más auge de los últimos años, los modelos no dejan de evolucionar y cada día aparecen redes neuronales con más parámetros que la anterior. Existen multitud de aplicaciones y en esta sección repasaremos algunas de ellas.

1.3.1. Análisis de sentimiento y clasificación

El análisis de sentimiento es una de las primeras aplicaciones que surgieron del NLP. Conocer la opinión del público en relación a un tema es muy importante para empresas y organizaciones, las ayuda a percibir el sentimiento hacia su marca, su producto o su servicio.

El análisis de sentimiento consiste en clasificar los textos entre positivos, negativos o neutros en función del mensaje que se esté transmitiendo. Hay palabras que indican un sentimiento negativo, como los adjetivos despectivos, y palabras que indican sentimiento positivo, como los adjetivos apreciativos. El conjunto de estas palabras dentro de la oración nos determina su sentimiento final, pero no solo depende de ello, hay que considerar también el contexto en el que se encuentra o incluso la ironía. Los avances más recientes en el aprendizaje profundo han logrado una considerable mejora en este aspecto. Uno de los primeros proyectos en aparecer fue ELMo [8], que permitió una representación profunda de las palabras y su contexto.

Para analizar el mensaje al completo también es muy importante considerar su intencionalidad. Esto se logra mediante la clasificación del mensaje en diferentes grupos, por ejemplo, si se trata de una pregunta, una queja, una sugerencia, etc. Se puede entrenar una red neuronal para que haga cualquier tipo de clasificación de manera automática.

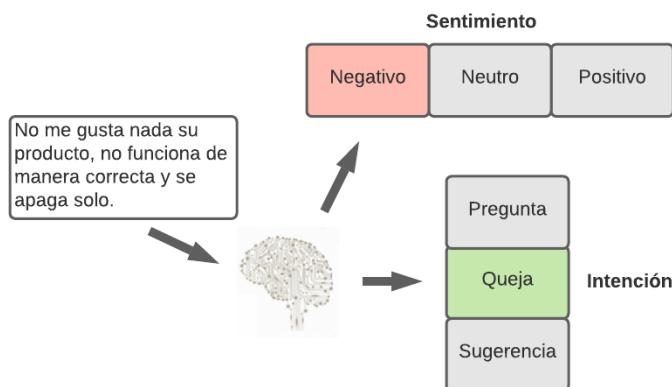


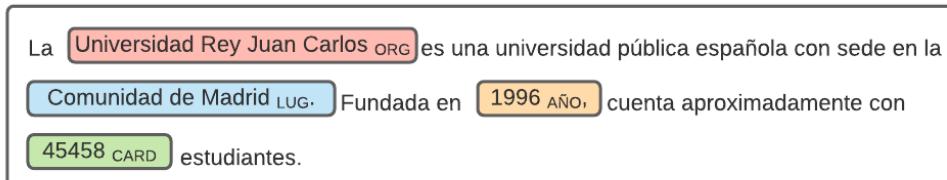
Figura 1.1: Análisis de sentimiento e intención (original).

1.3.2. Extracción de texto

La extracción de texto nos permite detectar de manera automática información específica, como nombres, compañías, lugares, fechas, etc. Se conoce también como reconocimiento de entidades nombradas (*Named entity recognition*, NER). Puede extraerse cualquier tipo de palabra clave predefinida con anterioridad.

Una de las aplicaciones más útiles del NER es el procesamiento de mensajes sobre los que haya que ejecutar una acción. Por ejemplo, tenemos que devolver un importe a un cliente a una determinada tarjeta de crédito que nos indica, gracias al NER se pueden extraer los datos del cliente y su número de tarjeta del mensaje y efectuar el pago, todo de manera automática. Las entidades extraídas también nos permiten enriquecer un posible análisis de sentimiento, como mostrar aquellas palabras sobre las que se está dando la visión negativa (o positiva).

Para lograr la extracción de entidades, una red neuronal necesita asignar a cada palabra de un texto una etiqueta con la clase a la que corresponde (nombre, lugar, fecha, cardinal...). Las redes recurrentes y en especial las LSTMs, que veremos más adelante, nos permiten ejecutar esta tarea.



La Universidad Rey Juan Carlos ORG es una universidad pública española con sede en la Comunidad de Madrid LUG. Fundada en 1996 AÑO, cuenta aproximadamente con 45458 CARD estudiantes.

Figura 1.2: Ejemplo NER (original).

1.3.3. Transformación de texto

En los últimos años ha surgido una nueva arquitectura de red neuronal que ha permitido dar un salto cualitativo al NLP, los *transformers* [9]. Fueron desarrollados para resolver el problema de transformación de secuencias (*sequence transduction* [10] o *neural machine translation*), que consiste en cualquier tarea que transforme una secuencia de entrada en una secuencia de salida. Incluye el reconocimiento de voz, la transformación de texto a voz, la traducción a otro idioma, etc.

Su funcionamiento se basa en combinar las redes convolucionales con mecanismos de atención, que permiten a la red fijarse únicamente en ciertas palabras en específico, por ejemplo, un sustantivo que se está traduciendo en ese momento junto a sus adjetivos. Esto se logra mediante etapas de codificación y decodificación, concretamente un *transformer* tiene 6 codificadores (*encoders*) y 6 decodificadores (*decoders*). Explicar el detalle de esta arquitectura no es materia de este trabajo.

Los dos modelos más conocidos en la actualidad construidos a partir de *transformers* son **BERT** [11], desarrollado por Google e incorporado en su propio motor de búsqueda y en su traductor, y **GPT-3** [12], desarrollado por OpenAI.

Estos modelos se están empleando en multitud tareas en la actualidad y se pueden agrupar entre las siguientes:

- Asistentes virtuales y chatbots: se utilizan los modelos tanto a la hora de reconocer la voz, como posteriormente para generar una respuesta inteligente al usuario, tenemos ejemplos conocidos como Siri o Alexa.
- Traducción: el modelo transforma una secuencia de texto en un idioma en otra secuencia de un idioma diferente (DeepL o Google Traductor).
- Resumen de textos: el modelo transforma el texto original en un texto más corto que resume todo el contenido.
- Generación de textos: dado un texto inicial al modelo, este es capaz de continuar su generación acorde al tema introducido.

GPT-3, el modelo más grande a día de hoy con 175 millones de parámetros, se está aplicando en proyectos específicos, algunos de los más destacados son:

- Generadores de código ([Debuild](#)) y de modelos de aprendizaje automático a partir de una descripción dada.
- Asistentes para la escritura, ofreciendo propuestas de cómo terminar cada frase ([Compose AI](#)) o generando la oración completa a partir de una idea o concepto ([OthersideAI](#)).
- Responder a preguntas, cuya respuesta se encuentra dentro de documentos introducidos al modelo previamente ([OpenAI](#)).
- Generadores de noticias falsas, como un artículo publicado en [The Guardian](#) escrito por GPT-3 en 2020 [13].

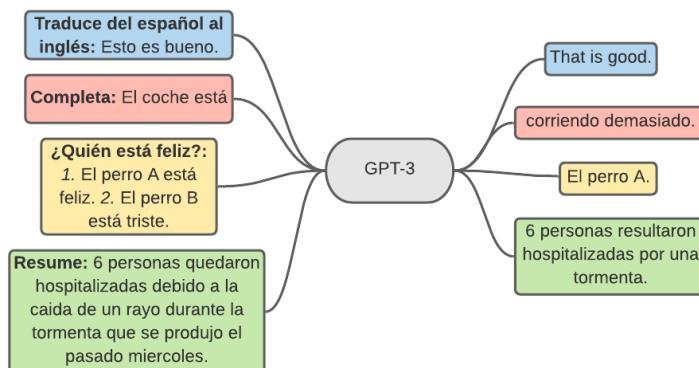


Figura 1.3: Usos de GPT-3 (original).

2

Planificación

En todo proyecto de Software la planificación siempre es el punto de partida del mismo. Tener una buena planificación nos facilitará el desarrollo y nos servirá de pilar para cubrir todos los objetivos.

Durante este apartado trataremos los requisitos, la metodología de desarrollo empleada y las alternativas consideradas.

2.1. Requisitos

Requisitos funcionales:

- **RF1:** El programa tiene que ser capaz de dar al usuario un nivel de afinidad a una película en forma de porcentaje (0-100 %).
- **RF2:** El programa tiene que usar la sinopsis de la película para dar el valor de afinidad.
- **RF3:** El programa tiene que permitir al usuario marcar películas afines y no afines.
- **RF4:** El programa tiene que mostrar como salida el porcentaje de afinidad a la película junto a los datos de la misma (cartel y sinopsis).

Requisitos no funcionales:

- **RNF1:** La aplicación debe dar una respuesta rápida al usuario sin que existan tiempos de espera amplios.
- **RNF2:** La aplicación debe tener la capacidad de actualizarse para incorporar siempre la cartelera de películas actual y poder añadir nuevas funciones o cambios en el diseño.

2.2. Estudio de alternativas

Partiendo del listado de requisitos es necesario elegir entre las opciones existentes para llevar a cabo la implementación.

Python Se ha elegido este lenguaje, frente a R o Scala, por su sencillez, la cantidad de librerías disponibles para el tratamiento de textos (NLTK) e implementación de redes neuronales (Keras y Tensorflow), documentación y por su amplia demanda en la actualidad.

Keras Se ha optado por esta librería al ofrecer una API de alto nivel que corre sobre Tensorflow, la otra alternativa considerada. Ofrece buenos rendimientos y además su código es conciso y fácil de leer permitiendo un rápido prototipado.

Flask Se ha seleccionado este framework para construir la aplicación web sencilla que nos permita cubrir el resto de requisitos (marcado de películas y salida de datos). La otra alternativa es Django, pero Flask es más ligero y cuenta con una menor curva de aprendizaje. Es recomendable su uso en aplicaciones web pequeñas como la nuestra.

2.3. Metodología de desarrollo

En la ingeniería del software existen principalmente dos formas o enfoques de desarrollo:

- **Desarrollo en cascada:** Esta metodología ordena de manera rigurosa las etapas del desarrollo, no se puede continuar con la siguiente fase hasta haber terminado la anterior. Es una metodología tradicional y rígida que consta normalmente de cinco fases (requisitos, diseño, implementación, verificación y mantenimiento).

- **Desarrollo ágil:** es un enfoque basado en el desarrollo iterativo e incremental, el proyecto va avanzando y adaptándose a las necesidades cambiantes del cliente. Es ejecutado por equipos multidisciplinarios que comparten ideas para la toma de decisiones. Al final de cada etapa del ciclo de vida, el objetivo no es agregar toda la funcionalidad recogida en los requisitos iniciales, si no ir agregando valor a un software mínimo que funciona sin errores.

Se ha elegido un desarrollo ágil para nuestra aplicación, concretamente un marco de trabajo **Scrum** [14], que basa la calidad del proyecto en los equipos que intervienen frente a los procesos, solapando las fases de desarrollo, en lugar de realizar una a continuación de la otra.

En nuestro proyecto, contamos únicamente con una persona, por lo tanto tendrá que desempeñar todas las tareas básicas de cada rol de Scrum. Se ha elegido este marco de trabajo, pese a no disponer de un equipo completo, ya que es más riguroso con la calidad del producto software que se entrega, da una mayor prioridad al testing y es más flexible ante los posibles cambios.

En Scrum existen tres roles principales: el *Scrum Master*, que facilita la aplicación del marco de trabajo y gestiona los cambios, el *Product Owner*, representante de los *stakeholders*, interesados en el proyecto (internos y externos), y el *Team* (equipo) que desarrolla.

Cada interacción del desarrollo se denomina *sprint*, un periodo de tiempo que varía normalmente entre una y cuatro semanas. Al final de cada *sprint*, incluido el primero, se consigue tener un producto mínimo viable (MVP).

Los requisitos del cliente se registran en el *product backlog*, una lista de características expresadas en forma de *historias de usuario* (HU), contadas desde la perspectiva del usuario o el equipo. Siempre han de contener un *como* (persona), un *quiero* (qué se quiere tener) y un *para* (para qué se quiere tener).

El *product owner* se presenta en el *sprint planning meeting* (reunión al inicio del sprint) con el *product backlog* priorizado. El equipo selecciona los items que pueden completar en el sprint y los mueve al *sprint backlog* correspondiente, ampliando cada historia de usuario en tareas. En nuestro caso, el único miembro del equipo se encarga de todo. También son importantes los *daily meetings*, reuniones diarias durante el sprint para mantener al equipo actualizado sobre las tareas.

2.3.1. Historias de usuario

Tras elegir el marco Scrum, necesitamos detallar cual es la división de los requisitos iniciales en historias de usuario. Cada HU se escribe siguiendo el **como**, el **quiero** y el **para** visto antes. Partiendo de los requisitos, nuestras HU quedan:

2.3. Metodología de desarrollo

REQ	Historia de usuario	P	T	O
RF1	HU1 COMO usuario QUIERO que la aplicación aprenda de mis gustos PARA obtener el resultado esperado.	A	5	8º
	HU2 COMO equipo QUIERO construir una red neuronal convolucional PARA aplicarla.	A	5	5º
	HU3 COMO equipo QUIERO construir una red neuronal recurrente PARA aplicarla.	A	5	6º
	HU4 COMO equipo QUIERO unir la red convolucional y la red recurrente PARA construir un modelo final.	A	5	7º
RF2	HU5 COMO equipo QUIERO disponer de un método de testing PARA conocer el rendimiento de mi modelo.	M	2	11º
	HU6 COMO equipo QUIERO disponer de un dataset de sinopsis con etiquetas PARA entrenar el modelo.	A	3	2º
	HU7 COMO equipo QUIERO dividir los datos PARA tener muestra de entrenamiento y test.	A	1	3º
RF3	HU8 COMO equipo QUIERO codificar las sinopsis de manera adecuada PARA poder entrenar el modelo.	A	3	4º
	HU9 COMO usuario QUIERO poder marcar las películas que me gustan y no PARA compartirlo con la aplicación.	B	3	13º
RF4	HU10 COMO usuario QUIERO poder ver mi afinidad con las películas en forma de % PARA decidir si verlas o no.	M	2	9º
	HU11 COMO usuario QUIERO poder ver el cartel y la sinopsis de las película PARA decidir si verla o no.	B	2	12º
RNF1	HU12 COMO equipo QUIERO tener un modelo de tamaño suficiente PARA dar una respuesta en un tiempo razonable.	M	1	10º
	HU13 COMO equipo QUIERO que mi repositorio de código esté conectado con Travis CI PARA tener integración continua y hacer validaciones antes del despliegue.	B	2	15º
	HU14 COMO equipo QUIERO que repositorio de código esté conectado con Sonarcloud PARA conocer la calidad del código.	B	1	16º
RNF2	HU15 COMO equipo QUIERO tener mi código en un repositorio de Github PARA poder añadir implementaciones futuras y tener un seguimiento.	A	1	1º
	HU16 COMO equipo QUIERO que mi aplicación se conecte via API PARA obtener las películas recientes.	B	2	14º

A cada HU le hemos asignado un valor de prioridad (**P**): baja (B), media (M) o alta (A). Y una estimación en forma de puntos de historia (**T**), los cuales se basan en la serie de fibonacci (1, 1, 2, 3, 5, ...) que dan una idea del tamaño y el esfuerzo para ejecutarla.

Además existen HU que dependen de otras, esto es muy importante a la hora de asignar el orden de ejecución (**O**), aparte de tener en cuenta los puntos y la prioridad. Por ejemplo, para poder ejecutar la HU7 es necesario previamente realizar la HU6.

2.3.2. Tablero Scrum

Para llevar el control de las historias de usuario y de las tareas se emplea un tablero (*Scrum Board*) que debe ser visible por el equipo en todo momento para que pueda conocer su estado, existen variantes tanto físicas como digitales. El tablero debe dividirse en cuatro columnas:

- **PBI (Product Backlog Items)**: Se colocan las historias del *product backlog* definidas y priorizadas por el *product owner*.
- **To do**: Aparecen las historias y tareas por hacer, que el equipo ha acordado que son viables de completar durante el sprint.
- **Doing**: Debe contener las tareas ya asignadas a miembros del equipo y que están en proceso de realización.
- **Done**: Ha de contener las tareas ya completadas en su totalidad.

Las tareas se van moviendo entre columnas (de izquierda a derecha) a medida que avanza el progreso de las mismas.

Para la construcción de nuestro tablero digital hemos elegido la herramienta online [Trello](#), frente a otras que requieren instalación o dibujar un tablero físico.

Ahora nos encargaremos de desglosar las historias de usuario en tareas y dar detalle de la implementación. Hemos seleccionado las cinco primeras historias de usuario para el primer sprint y se han creado las siguientes tareas:

- HU15: COMO equipo QUIERO tener mi código en un repositorio de Github PARA poder añadir implementaciones futuras y tener un seguimiento.
 - Tarea 15.1: Crear el repositorio en Github.
 - Tarea 15.2: Crear README con la descripción del proyecto.
 - Tarea 15.3: Crear estructura de carpetas del proyecto.

- HU6: COMO equipo QUIERO disponer de un dataset de sinopsis con etiquetas PARA entrenar el modelo.
 - Tarea 6.1: Encontrar una base de datos de películas en español.
 - Tarea 6.2: Descargar una cantidad suficiente de sinopsis (> 4000).
 - Tarea 6.3: Etiquetar películas según mis gustos (1000 que me gustan, 1000 que no me gustan aprox.).
- HU7: COMO equipo QUIERO dividir los datos PARA tener muestra de entrenamiento y test.
 - Tarea 7.1: Separar el dataset original en dos muestras (train 70% y test 30%).
 - Tarea 7.2: Separar de tal manera que la distribución de las etiquetas sea igual en ambas muestras.
 - Tarea 7.3: Separar de tal manera que la distribución de las longitudes de las sinopsis sea similar en ambas muestras.
- HU8: COMO equipo QUIERO codificar las sinopsis de manera adecuada PARA poder entrenar el modelo.
 - Tarea 8.1: Construir codificación basada en índices.
 - Tarea 8.2: Añadir un padding a la codificación para que tengan igual longitud.
- HU2: COMO equipo QUIERO construir una red neuronal convolucional PARA aplicarla.
 - Tarea 2.1: La red debe tener una capa Embedding.
 - Tarea 2.2: La red debe tener al menos una capa Conv1D.
 - Tarea 2.3: La red debe tener al menos una capa MaxPooling1D.
 - Tarea 2.4: La red debe tener una capa Dense.
 - Tarea 2.5: La red debe tener una capa Dense de 1 neurona de salida con función de activación sigmoide.

Esta metodología de expansión la hemos ido aplicando al resto de historias de usuario a lo largo de cada sprint, para acabar construyendo el tablero que podemos ver en la figura 2.1. Los colores de las HU representan las prioridades asignadas (rojo - alta, amarillo - media y verde - baja).

La duración de nuestros sprints no está delimitada debido a no disponer de fechas de entrega, cada sprint terminará con la construcción de un modelo viable que mejore las características y estructura del anterior o incorpore funcionalidad visual nueva a la aplicación (MVP).

Capítulo 2. Planificación

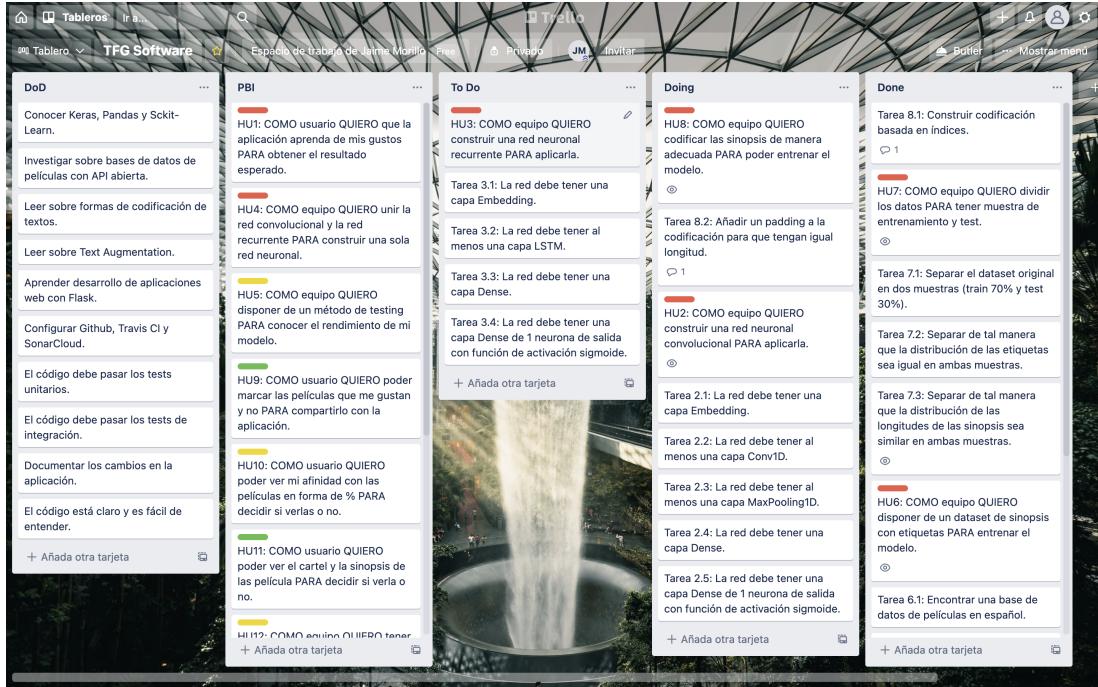


Figura 2.1: Tablero Trello durante el primer sprint.

2.3.3. Pruebas

La actividad del testing en una metodología en cascada se encuentra entre las últimas etapas del desarrollo, por lo que se le acaba asignando menos tiempo y recursos. El testing sin embargo es fundamental para asegurar la calidad del software que se está desarrollando, en el Scrum cobra mucha más importancia. Las pruebas (unitarias, de integración, de estrés...) se tienen que pasar para cada una de las historias de usuario en algún punto de su realización.

Existen tres formas principales de testing:

- TDD (Test driven development): es una práctica de desarrollo enfocada en cómo escribir el código y cómo debería funcionar. El desarrollador es el que escribe los tests y se ocupan solo a nivel unitario o de una porción pequeña de la aplicación en desarrollo, por ejemplo una historia de usuario.
- BDD (Behaviour driven development): es una metodología de trabajo con un enfoque de equipo que hace hincapié en por qué debes escribir ese código y cómo se debería comportar. El usuario final, junto al resto del equipo, escriben los tests que se van a ocupar de las pruebas sobre la integración de las diferentes unidades, estas se encuentran en un idioma de negocio.

- ATDD (Acceptance test driven development): es una metodología de trabajo en la que todo el equipo y usuarios finales analizan conjuntamente los criterios de aceptación, antes de que comience el desarrollo. Las pruebas de aceptación se hacen antes de desarrollar y se automatizan, la ventaja de esto es que ayuda a asegurar que todos los miembros del proyecto entienden qué hay que hacer y saben que están haciendo lo correcto [15].

La práctica elegida para nuestro proyecto es TDD, es la más recomendada cuando se necesitan hacer pruebas de manera unitaria y se quiere un *feedback* rápido del código. Las otras dos metodologías requieren implicación por parte de los equipos y usuarios finales, y en nuestro caso contamos únicamente con un desarrollador.

Para ejecutar de manera correcta la práctica TDD, vamos a definir un test unitario, a partir de cada historia de usuario, que tendrá que pasar el código y las funciones que escribamos.

2.3.4. Diseño de la solución

Tras el estudio de alternativas y explicación de la metodología a seguir, detallaremos el uso de las herramientas para el diseño y construcción de la solución final.

Para el desarrollo del modelo y la experimentación se han empleado notebooks en el entorno de [Google Colab](#).

El resto del proyecto, que es la parte interactiva, se trata de una aplicación web que contiene el modelo en producción, permite el etiquetado de datos, muestra la afinidad al usuario junto a los datos de la película y reentrena el modelo cuando dispone de suficientes etiquetas nuevas. La web se ha desarrollado en un entorno local siguiendo un esquema de clases y scripts.

En la figura 2.2 vemos el aspecto al lanzar la aplicación, donde el número que aparece en color verde o rojo es la afinidad o recomendación a esa película. El usuario puede marcar si le gusta o no una película mediante los botones y pasar entre ellas, además se le permite ampliar una vista detallada de las mismas.

El conjunto de los test se encuentran integrados dentro del módulo de test, se puede instanciar desde el interior de los notebooks o ejecutar de manera independiente.

Todo el código se encuentra en su repositorio de [Github](#) correspondiente, donde también se detallan las instrucciones para lanzar la aplicación.

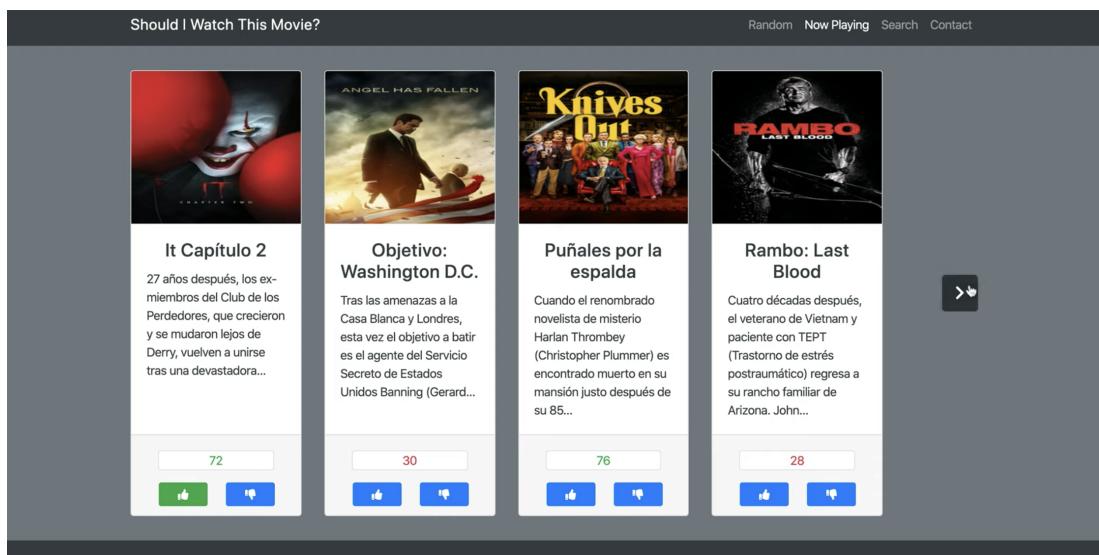


Figura 2.2: Vista de la aplicación web.

3

Fundamentos

En esta sección se describen los fundamentos necesarios para la construcción y comprensión del proyecto. Todos los conceptos explicados son la base de la implementación y experimentos.

3.1. NLP

Las redes neuronales necesitan recibir como entrada un vector de características numérico y nosotros partimos de un conjunto de textos que es necesario transformar. Dentro del procesamiento del lenguaje natural existen varios mecanismos de codificación para lograrlo.

3.1.1. Bolsa de palabras

El término procede del inglés *Bag of words* (*BoW*) [16], es un método utilizado para representar los documentos (textos) sin tener en cuenta el orden en el que aparecen las palabras dentro del mismo. El texto es representado por la bolsa o multiconjunto de sus palabras, que indica su multiplicidad de aparición.

Podemos ver su construcción con un ejemplo sencillo. Partimos de un conjunto de documentos o textos denominado *corpus*:

- (a) Este es el primer documento.
- (b) Este documento es el segundo documento.
- (c) Este es el tercero.

Generamos el diccionario o vocabulario único de las palabras que forman el corpus [*documento, el, es, este, primer, segundo, tercero*], cada una de estas palabras será una columna en el espacio de vectores. De esta manera cada texto se puede convertir en un vector que contiene la frecuencia de aparición de cada una de las palabras del diccionario. En la tabla 3.1 podemos ver cual es el resultado de aplicar *BOW* sobre el ejemplo.

	documento	el	es	este	primer	segundo	tercero
(a)	1	1	1	1	1	0	0
(b)	2	1	1	1	0	1	0
(c)	0	1	1	1	0	0	1

Tabla 3.1: Bag of words encoding.

El ejemplo i-ésimo de entrada de la red quedaría representado por $X^{(i)} \in [x_1, \dots, x_N]$ con $x_i \in \mathbb{N}$ y donde N es el tamaño total del diccionario. Por lo tanto, el conjunto total de M documentos será una matriz $X \in \mathbb{N}^{M \times N}$.

Esta codificación (*encoding*) es fácil de generar y de interpretar, pero tiene dos desventajas principales; el tamaño de los vectores crece rápidamente al aumentar el tamaño del diccionario y no tiene en cuenta el orden de las palabras dentro de la oración, importante para considerar las estructuras gramaticales.

3.1.2. Codificación One-Hot

Esta codificación es capaz de retener el orden de las palabras, el cual puede ser explotado por algunas redes neuronales como las recurrentes. Los vectores resultantes de esta codificación están formados exclusivamente por 0s y 1s.

El texto es representado por un vector que indica la presencia o ausencia de las palabras, manteniendo el orden secuencial de las mismas. Cada documento es representado por un tensor, consistiendo cada uno de ellos en una secuencia de varios vectores de 0s y 1s, lo que da lugar a una representación muy dispersa del corpus.

Veamos un ejemplo empleando el corpus anterior. Partimos del diccionario de palabras y asignamos a cada una un vector que contiene 0s en todas las posiciones menos un 1 en la posición del índice que ocupa la palabra, como se ve en la tabla 3.2.

	índice	vector
documento	0	[1, 0, 0, 0, 0, 0, 0]
el	1	[0, 1, 0, 0, 0, 0, 0]
es	2	[0, 0, 1, 0, 0, 0, 0]
este	3	[0, 0, 0, 1, 0, 0, 0]
primer	4	[0, 0, 0, 0, 1, 0, 0]
segundo	5	[0, 0, 0, 0, 0, 1, 0]
tercero	6	[0, 0, 0, 0, 0, 0, 1]

Tabla 3.2: One-hot encoding de palabras.

En la tabla 3.3 se observa de manera secuencial (de arriba a abajo) el resultado final de aplicar la codificación anterior a cada palabra del texto (b).

	documento	el	es	este	primer	segundo	tercero
este	0	0	0	1	0	0	0
documento	1	0	0	0	0	0	0
es	0	0	1	0	0	0	0
el	0	1	0	0	0	0	0
segundo	0	0	0	0	0	1	0
documento	1	0	0	0	0	0	0

Tabla 3.3: One-hot encoding de documentos.

Para asegurar que las matrices asociadas a cada documento tienen siempre el mismo número de filas, es necesario llenar con vectores de ceros, aquellos textos de menor longitud, tantas filas como sean necesarias para alcanzar la longitud del documento más largo.

Por lo tanto, el ejemplo i -ésimo quedaría representado por la matriz dispersa $X^{(i)} \in [0, 1]^{K \times N}$ donde $K = \max(\text{long}(X^{(i)}))$ y N es el tamaño total del diccionario. El conjunto de M documentos será un tensor de tres dimensiones $X \in [0, 1]^{M \times K \times N}$.

La principal desventaja de esta codificación es la escalabilidad, el tamaño del vector de cada palabra sigue creciendo en relación al del corpus. Además estamos introduciendo una dimensión extra a la representación del texto y al acabar con matrices tan dispersas computacionalmente es costoso de tratar.

3.1.3. Codificación basada en índices

Este método de codificación es el más intuitivo y tiene en cuenta el orden de las palabras. Consiste en asignar a cada palabra el índice que ocupa en el

diccionario, de tal manera que cada documento pasa a ser un vector de índices. El resultado sobre nuestro ejemplo lo vemos en la tabla 3.4.

	Index-Based encoding
Este es el primer documento.	[3, 2, 1, 4, 0]
Este documento es el segundo documento.	[3, 0, 2, 1, 5, 0]
Este es el tercero.	[3, 2, 1, 6]

Tabla 3.4: Index-Based encoding.

Para asegurar que todos los vectores tengan el mismo tamaño se reserva el índice 0 y se completan hasta que sean de igual longitud (*padding*). El padding de 0s se puede aplicar al principio o al final del vector.

El ejemplo i -ésimo quedaría representado por el vector denso $X^{(i)} \in [0, n]^K$ donde $K = \max(\text{long}(X^{(i)}))$ y n es la longitud del diccionario. El conjunto de M documentos será una matriz $X \in [0, n]^{M \times K}$.

El inconveniente de esta codificación es que introduce una distancia numérica entre los textos y las propias palabras que no es real.

3.1.4. Word Embedding

Word embedding es una técnica del procesamiento del lenguaje natural que permite mapear el significado semántico de las palabras en un espacio geométrico. Esto se consigue mediante la asociación de cada palabra a un vector de números reales. De tal manera que la distancia entre dos vectores cualquiera capture la relación semántica entre las dos palabras asociadas, como vemos en la figura 3.1 (la distancia entre hombre-mujer y rey-reina es la misma).

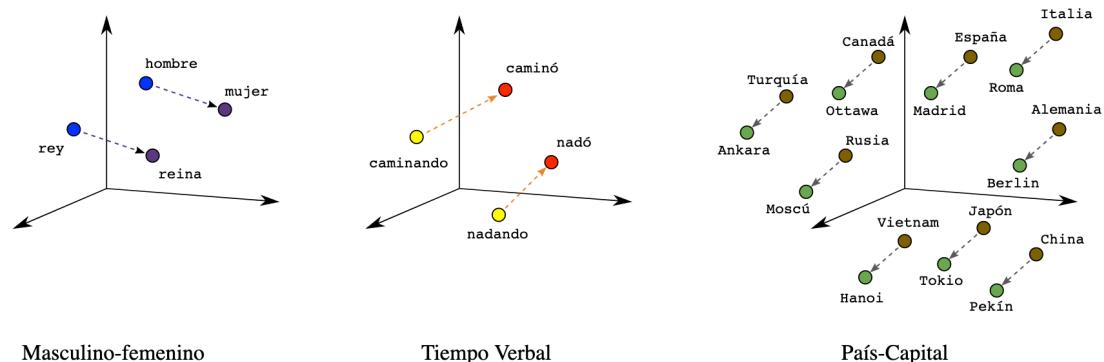


Figura 3.1: Word embedding [1].

El espacio geométrico formado por estos vectores se denomina *embedding space* y conceptualmente implica el encaje matemático¹ de un espacio inicial en el que tenemos una dimensión por cada una de las palabras a un espacio vectorial continuo de menos dimensiones. Cada una de las dimensiones del espacio representa una cualidad de la palabra, como podría ser el género, el número, etc.

Para generar esta representación se pueden emplear las propias redes neuronales (técnica conocida como Word2Vec [17]) o la reducción de dimensionalidad de matrices de co-ocurrencia de palabras (GloVe [18]).

En la práctica, proyectaremos cada palabra en un espacio vectorial continuo, mediante una capa en la red neuronal específica para esta tarea (capa de embedding), que aprende a asociar la representación vectorial óptima de cada palabra para completar la tarea general. Se realiza la proyección de la matriz one-hot dispersa en un espacio latente continuo y denso mediante una matriz de embedding de tamaño ($n \times m$) donde n es el tamaño del diccionario y m son las dimensiones del nuevo espacio.

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}_A \times \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \\ m & n & o \\ p & q & r \\ s & t & u \end{pmatrix}_M = \begin{pmatrix} j & k & l \\ a & b & c \\ g & h & i \\ d & e & f \\ p & q & r \\ d & e & f \end{pmatrix}_Z \quad (3.1)$$

En la ecuación 3.1 vemos un ejemplo, donde A es el documento (b) codificado de la forma one-hot, M es la matriz de embedding con los pesos ($a - u$) a calcular por la red y Z es la codificación resultante con menos dimensiones.

Mediante la codificación word embedding, el ejemplo i -ésimo quedaría representado por la matriz densa $X^{(i)} \in \mathbb{R}^{K \times V}$ donde $K = \max(\text{long}(X^{(i)}))$ y V se obtiene del tamaño de la matriz embedding ($U \times V$) en la que U es el tamaño del diccionario y V se fija antes del entrenamiento. El conjunto de N documentos será un tensor de tres dimensiones $X \in \mathbb{R}^{N \times K \times V}$.

El problema de los embeddings es que son fijos y no comprenden el significado de las palabras según el contexto en el que se encuentran. Se están desarrollando nuevos mecanismos sin esta limitación (*contextualized word embeddings* [19]).

¹ Un encaje o inmersión (embedding) es una instancia de una estructura matemática contenida dentro de otra. Por ejemplo, un subgrupo es el embedding de un grupo.

3.2. Machine learning

3.2.1. Aprendizaje supervisado - clasificación

El problema que planteamos se enmarca dentro del aprendizaje supervisado, concretamente se trata de una clasificación binaria. En los problemas de clasificación y en general en el aprendizaje automático necesitamos que se den una serie de puntos para tratarlos:

- Cada una de las entradas de nuestro modelo tiene que estar representada como un vector de características numérico y además han de ser del mismo tamaño (visto en la sección 3.1).
- Todas las muestras que usaremos para el entrenamiento han de estar etiquetadas, en un problema de clasificación son la clase o clases a las que pertenecen.
- La muestra total ha de dividirse en al menos dos particiones:
 - El conjunto de entrenamiento o train que usaremos para entrenar nuestro modelo, aproximadamente el 80 % del total.
 - El conjunto de test que usaremos para probar el modelo, el 20 % restante. Es recomendable dejar un 10 % para validación si es posible.

3.2.2. Redes neuronales

Se presentan brevemente los diferentes tipos de estructuras de redes neuronales involucrados en este trabajo.

Redes neuronales multicapa Las redes neuronales son la columna vertebral de muchos de los modelos actuales de aprendizaje automático, su funcionamiento se basa en imitar vagamente el cerebro humano para reconocer patrones.

El bloque básico de construcción se llama neurona o perceptrón. Recibe una serie de entradas, las multiplica por un peso w_i , suma los resultados y aplica una función de activación. Las dos funciones más usadas son la ReLU o Rectificador y la sigmoide.

Una capa o *layer* es un conjunto de neuronas que reciben una entrada de las neuronas anteriores y proporcionan una salida. Las funciones de activación de las neuronas de una misma capa han de ser iguales. En la figura 3.2 vemos una pequeña red neuronal.

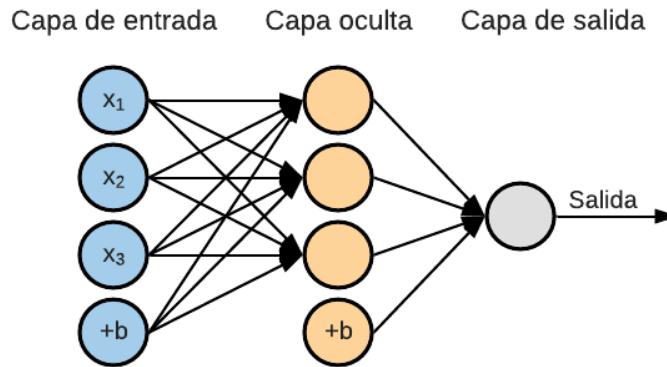


Figura 3.2: Red neuronal (original).

Cuando las neuronas de cada capa están conectadas a todas las neuronas de la capa anterior, la red recibe el nombre de perceptrón multicapa o red neuronal densamente conexa (*Fully Connected Neural Network*, FCNN). El número de capas ocultas es variable en función de la complejidad del problema, cuando tenemos más de 3 o 4 capas ocultas se la considera “red profunda” [20].

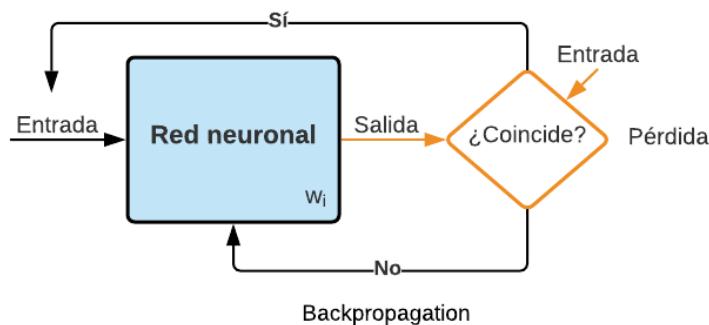


Figura 3.3: Actualización de la red (original).

En la figura 3.3 vemos el proceso general de aprendizaje y actualización del modelo. Cuando se da una **entrada** a la red neuronal siempre devuelve una **salida** gracias al mecanismo de propagación hacia delante (*feedforward*). En el primer intento normalmente no se obtiene la salida correcta, y por eso, cada entrada viene acompañada de su **etiqueta**, para reflejar qué salida debería haber adivinado la red y calcular la diferencia (**perdida**). Si la salida es correcta, se mantienen los pesos actuales y se sigue con la siguiente entrada. Sin embargo, si no coincide con la etiqueta, se modifican los pesos mediante **backpropagation**, que consiste en “inspeccionar” cada conexión para comprobar cómo se comportaría la salida tras un cambio en el peso w_i de la neurona.

Redes neuronales convolucionales Las redes convolucionales (CNN) son una de las variantes de redes neuronales usadas ampliamente en el campo de la visión artificial. Reciben este nombre porque sus capas ocultas están compuestas principalmente de capas de convolución (*convolutional layers*) y capas de reducción (*pooling layers*), en las cuales, se aplica la operación de convolución y de reducción (*pooling*). En la figura 3.4, tenemos una CNN compuesta de dos bloques de convolución y pooling, que realizan una extracción automática de características, y un cabezal para la tarea principal (*fully connected*).

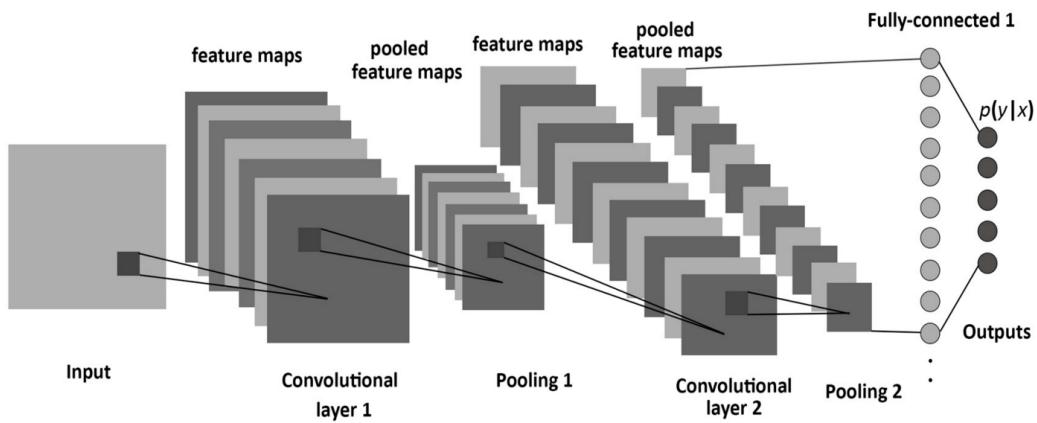


Figura 3.4: Red neuronal convolucional. [2]

Convolución La convolución opera sobre dos matrices, una normalmente es una imagen, aunque en nuestro caso es un texto codificado mediante word embedding, la otra matriz la denominamos núcleo o kernel y actúa como filtro sobre la primera matriz produciendo como salida una nueva.

En las imágenes, es fácil visualizar un núcleo como si se desplazara por toda la imagen cambiando el valor de cada píxel en el proceso. Esto permite extraer un mapa de características de la misma, por ejemplo, las esquinas o los bordes verticales.

En el caso de los textos, un filtro no puede abarcar una fila de manera parcial. La totalidad de fila es la representación de una palabra y por tanto debe tratarse al completo (figura 3.5), si no, estaríamos aplicando la convolución solo a ciertas cualidades de la palabra. Una palabra se entiende en su conjunto por todas las características que la componen.

Dado un texto codificado en una matriz de tamaño $N \times M$, el kernel ha de tener tamaño $Z \times M$, siendo necesario definir únicamente las filas, de ahí que la capa se denomine capa de convolución 1D. En el caso general se denomina capa de convolución 2D (la usada con imágenes).

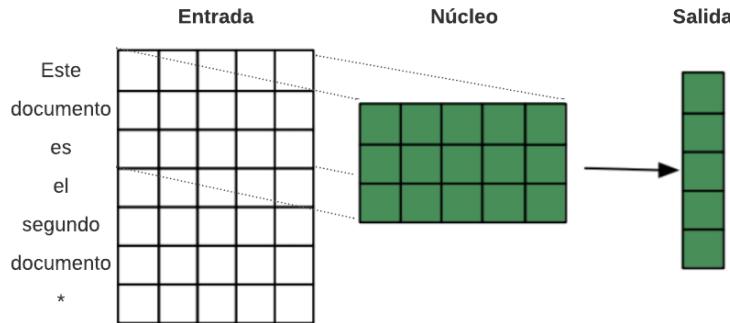


Figura 3.5: Convolución en textos.

Una convolución sobre un texto permite extraer las características de los grupos de palabras próximas. En la figura 3.5 vemos una convolución que extrae la información de las combinaciones de tres palabras.

Reducción La reducción o pooling es un proceso de discretización basado en la muestra. Su objetivo es reducir la dimensionalidad de la entrada agrupando las características extraídas previamente por la convolución.

Hay dos tipos principales de pooling: reducción máxima (max pooling) y reducción media (average pooling). La reducción máxima recoge el valor máximo de la región y la reducción media el valor medio.

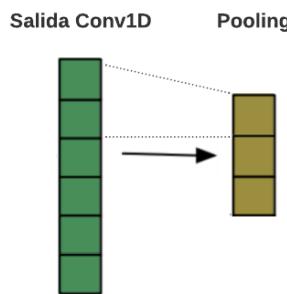


Figura 3.6: Reducción en textos.

Para un texto, la reducción se aplica sobre cada vector extraído previamente por cada uno de los filtros de convolución 1D (figura 3.6). Únicamente se define el número de filas que ocupa, ya que el número de columnas siempre ha de ser uno, denominándose así reducción 1D. El caso general se conoce como reducción 2D (para imágenes). En ambos casos el desplazamiento del filtro (*strides*) suele ser igual a su tamaño.

Redes neuronales recurrentes Las redes neuronales recurrentes (RNN) son una variante muy utilizada en el campo del procesamiento del lenguaje natural (NLP). En las redes neuronales vistas se da la suposición de que dos entradas sucesivas en la red son independientes entre sí. Esta suposición no la podemos considerar en muchos escenarios reales, como a la hora de predecir el precio de la luz o la siguiente palabra en una secuencia.

Las RNN se llaman recurrentes porque realizan la misma tarea para cada elemento de una secuencia. Contienen una serie de bucles internos que permiten persistir la información de lo procesado hasta el momento. Pueden considerarse como tener múltiples copias de la misma red, cada una de las cuales pasa un mensaje a su sucesora. En la práctica, únicamente logran recordar información de instancias recientes.

Redes LSTM Las LSTMs (*Long short-term memory*) fueron propuestas por Hochreiter & Schmidhuber en 1997 [21] con el objetivo de evitar el problema de la dependencia a largo plazo.

Todas las redes neuronales recurrentes tienen forma de cadena, compuesta de módulos de red neuronal repetidos. En las RNN estándar, este módulo se compone de una sola capa con la función de activación tangente hiperbólica (\tanh).

En las LSTM el módulo de repetición tiene cuatro capas, que se conectan como se puede ver en la figura 3.7. Cada módulo se denomina celda LSTM y el número de celdas de la cadena siempre es igual al tamaño de la secuencia de entrada. En nuestro caso habrá tantas celdas como palabras en el texto, cada celda procesa una nueva palabra junto a la información recordada de las anteriores.

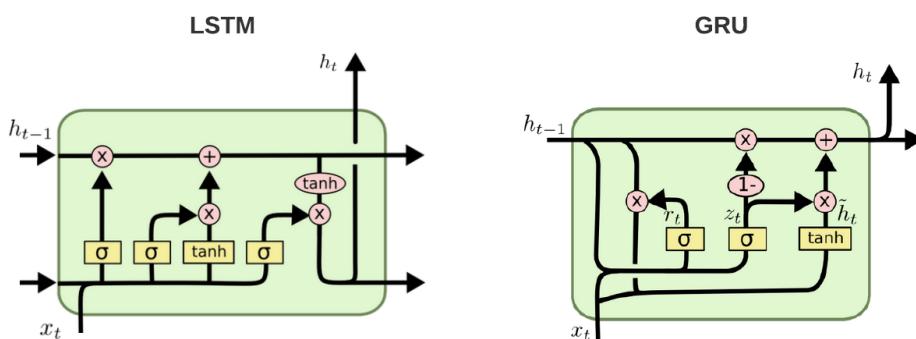


Figura 3.7: LSTM y GRU [3].

A la linea superior que atraviesa la celda se la denomina estado de la celda y tiene la capacidad de transportar la información entre las celdas con interacciones lineales menores. Una LSTM puede añadir o eliminar información del estado de

la celda gracias a una estructura de tres puertas que dejan pasar o no los datos. Estas puertas se componen de una sigmoide y una operación de multiplicación por elementos. La capa sigmoide devuelve números entre cero y uno, que describen la cantidad que hay que dejar pasar de cada componente.

Redes GRU Las redes GRU (*Gated Recurrent Unit*) son una variante de las LSTM que se diferencian por el tipo de celda (figura 3.7). Estas contienen únicamente dos puertas y computacionalmente son mucho más eficientes que las primeras.

Redes neuronales residuales (ResNet) Son un tipo de redes neuronales basadas en construcciones observadas en las células piramidales de la corteza cerebral. Consiguen emular a dichas células mediante conexiones de salto (*skip connections*) que se saltan ciertas capas específicas de la red. Los modelos típicos de ResNet implementan saltos sobre dos o tres capas de convolución que contienen funciones ReLU y capas de normalización (*batch normalization*).

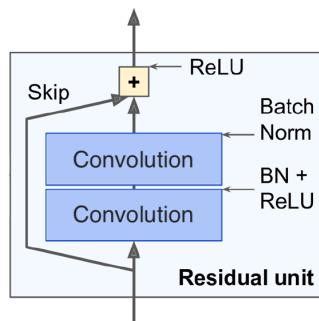


Figura 3.8: Unidad Residual [4].

La unidad básica se conoce como unidad residual (*residual unit*). Se puede observar en la figura 3.8, donde la salida procesada tras las dos convoluciones se concatena con la entrada (*skip*) mediante la operación suma. Gracias a las unidades residuales se logra que las capas posteriores de la red comiencen a aprender antes acelerando el entrenamiento.

Los tipos de redes descritos en esta sección serán los bloques fundamentales para construir los sistemas que se explicarán y se probarán en los capítulos siguientes.

4

Descripción del problema

En este capítulo se busca describir el problema, detallando los datos empleados y la solución final en profundidad. Vamos a explicar el diseño de la arquitectura, la implementación y las pruebas de nuestro proyecto. Nos enfocaremos principalmente en la descripción de las redes neuronales y en las etapas llevadas a cabo para su construcción.

4.1. Datos

Los datos iniciales son el conjunto de sinopsis que usaremos para obtener la predicción de afinidad o recomendación. Se han extraído los resúmenes en español vía API de la base de datos de [TMDB](#) (The Movie DataBase). Tras hacer frente a las limitaciones de la API, en total se han acabado recogiendo cerca de 15000 películas sobre las cuales se han asignado etiquetas (0 y 1) de manera manual, hasta completar suficientes para el entrenamiento (al menos 1000 de cada clase).

La entrada final de la red neuronal es el resultado de aplicar las técnicas de procesamiento, que describiremos en la siguiente sección, sobre los datos recopilados y correctamente etiquetados.

4.2. Técnicas aplicadas

El objetivo de este conjunto de técnicas es el de condensar la información y reducir el vocabulario de los textos, de manera que sigan manteniendo su significado inicial. Con esto conseguiremos acelerar el entrenamiento de la red neuronal, ya que la entrada es únicamente la información esencial.

Normalización Consiste en eliminar todas las tildes y llevar cada una de letras a minúscula, de tal manera que no pueda haber diferencias entre la palabra con tilde o sin ella, o si está en mayúsculas o minúsculas. Se eliminan también todos los signos de puntuación, caracteres especiales y espacios dobles que puedan existir.

Palabras vacías Palabras vacías o *stop words* [22] es el nombre que reciben las palabras que carecen de significado, como artículos, pronombres o preposiciones; las cuales nos interesa filtrar y eliminar antes del tratamiento de nuestras sinopsis. Son ruido para el modelo.

No existe una lista definitiva de *stop words* del español, cada problema puede tener la suya propia. Lo más frecuente es utilizar un [listado genérico](#), que contiene aquellas que son consideradas palabras vacías en la mayoría de ocasiones. Se comparan las palabras del listado con las palabras de cada sinopsis, si coincide se elimina y en caso contrario se mantiene.

Stemming La forma en la que aparecen las palabras puede penalizar su frecuencia de aparición, de tal manera que palabras de la misma familia semántica sean consideradas como totalmente diferentes.

El stemming consiste en llevar cada palabra a su raíz, esta raíz no tiene por qué pertenecer al diccionario, y se logra tras quitar los morfemas (prefijos y sufijos) de la palabra derivada.

Existen diferentes algoritmos, el más conocido es el de Porter [23], únicamente disponible para inglés. Nosotros hemos escogido una mejora de dicho algoritmo conocido como SnowballStemmer [24] disponible para español y que expondremos a continuación. Hemos usado la librería NLTK donde se encuentran ambas implementaciones.

Algoritmo de stemming para el español

Consideramos “vocales” los siguientes caracteres:

a e i o u á é í ó ú ü

R1 es la región después de la primera letra consonante precedida por una vocal, o la región vacía al final de la palabra si no hay tal consonante.

R2 es la región después de la primera consonante precedida por una vocal dentro de R1, o la región vacía al final de la palabra si no existe tal consonante.

f a b u l o s o	
<----->	R1
<---->	R2

RV se define como sigue:

Si la segunda letra es una consonante, RV es la región después de la siguiente vocal, o si las dos primeras letras son vocales, RV es la región después de la siguiente consonante, y en caso contrario (caso consonante-vocal) RV es la región después de la tercera letra. RV sería el final de la palabra si no se encuentran estas posiciones.

Por ejemplo,

m a c h o o l i v a t r a b a j o á u r e o	
... 	

Paso 0: Verbos pronominales

Buscar el sufijo más largo entre los siguientes

me se sel a sel o sel as sel os la le lo las les	
los nos	

y borrarlo si aparece después de uno de los siguientes

- (a) **miéndo ándo ár ér ír**
- (b) **mando iendo ar er ir**
- (c) **yendo** siguiendo a **u**

en RV. En el caso (c), **yendo** debe estar en RV, pero la **u** precedente puede estar fuera.

En el caso (a), la eliminación es seguida de borrar también el acento (por ejemplo, haciéndola → haciendo).

Paso 1: Eliminar sufijos estándar.

Buscar el más largo entre los siguientes sufijos

anza anz as ico ica icos icas ismo ismos able	
ables ible ibles ista istas oso osa osos osas	
amiento amien tos imiento imien tos	

y borrarlo si está en R2

**adora ador acción adoras adores acciones ante antes
ancia ancias**

y borrarlo si está en R2 y si está precedido por **ic**, borrarlo si todo está en R2

logía logías

y remplazarlo por **log** si está en R2

ución ucciones

y remplazarlo con **u** si está en R2

encia encias

y remplazarlo con **ente** si está en R2

amente

y eliminarlo si está en R1, si está precedido por **iv**, eliminarlo si todo está R2 (y si además está precedido por **at**, borrarlo si está en R2), en otro caso, si está precedido de **os**, **ic** o **ad**, eliminarlo si está en R2

mente

y borrarlo si está en R2, si está precedido por **ante**, **able** o **ible**, eliminarlo si está en R2

idad idades

y borrarlo si está en R2, si está precedido por **abil**, **ic** o **iv**, eliminarlo si está en R2

iva ivo ivas ivos

y borrarlo si está en R2, si está precedido por **at**, eliminarlo si está en R2

Paso 2a: Sufijos de verbos que comienzan con **y**.

Hacer si ninguna terminación fue eliminada en el **Paso 1**.

Buscar el más largo entre los siguientes sufijos en RV, y si se encuentra, eliminarlo si está precedido por **u**, no es necesario que **u** se encuentre en RV.

**ya ye yan yen yeron yendo yo yó yas yes yais
yamos**

Paso 2b: Otros sufijos de verbos.

Hacer el **Paso 2b** si se hizo el **Paso 2a**, pero falló al borrar un sufijo.

Buscar el sufijo más largo entre los siguientes en RV

en es éis emos

y eliminarlo, y si está precedido por **gu** eliminar la **u** (**gu** no necesita encontrarse en RV)

*arían arías arán arás aráis aría aréis aríamos
aremos ará aré erían erías erán erás eráis ería
eréis eríamos eremos erá eré irían irías irán
irás iráis iría iréis iríamos iremos irá iré
aba ada ida ía ara iera ad ed id ase iese
aste iste an aban ían aran ieran asen iesen
aron ieron ado ido ando iendo ió ar er ir as
abas adas idas ías aras ieras ases ieses ís áis
abais íais arais ierais aseis ieseis asteis
isteis ados idos amos ábamos íamos imos áramos
iéramos iésemos ásemos*

y eliminarlo

Paso 3: sufijos residuales. Siempre hacer este paso.

Buscar el sufijo más largo entre los siguientes en RV

os a o á í ó

y borrarlo si está en RV

e é

y borrarlo si está en RV, y si está precedido por **gu** con la **u** en RV
eliminar la **u**.

Lemmatization Es la alternativa al stemming y mediante esta técnica se lleva cada palabra a su lema, es la forma básica que aparece en el diccionario y representa a las demás palabras flexionadas (palabras derivadas mediante morfemas). Por ejemplo, “decir” sería el lema de “dije” pero también de “diré” o “dijéramos”.

Se basa en reglas lingüísticas por lo que no existe un algoritmo eficiente para aplicarla, necesitas tener el listado que mapea cada palabra con su correspondiente lema. Hemos usado la librería spacy para llevar a cabo dicha transformación.

4.3. Diseño

En esta sección se explica, en primer lugar, la arquitectura general de la aplicación en la que interviene la parte web, y a continuación, se detalla en profundidad la arquitectura y estructura de la red neuronal.

4.3.1. Diseño de la aplicación web

Primero veamos el esquema con toda la estructura de la aplicación (Figura 4.1). Las líneas sólidas indican el posible flujo e intercambio de la información entre componentes, la linea punteada la utilización de la funcionalidad del módulo y la linea gruesa la extracción de datos del origen.

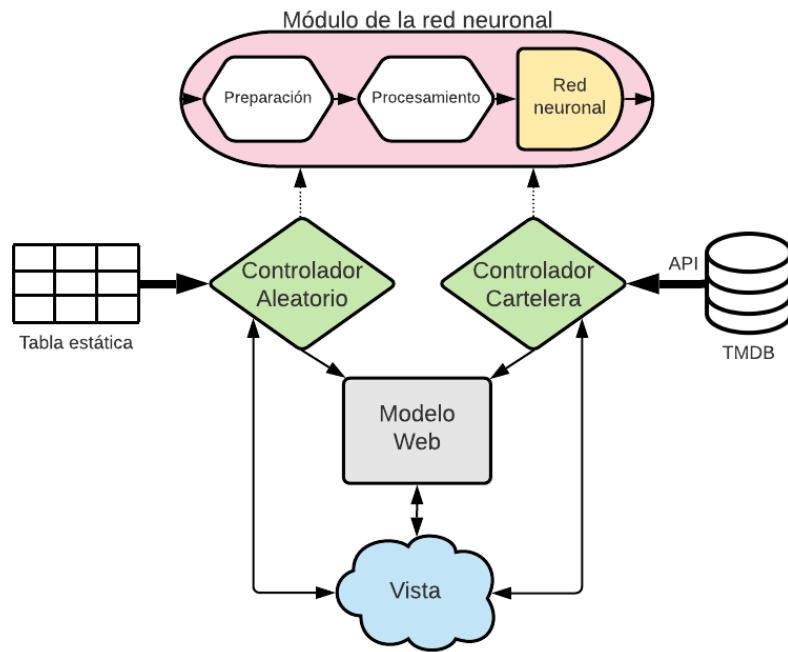


Figura 4.1: Diagrama de la aplicación.

La arquitectura está basada en el patrón de diseño modelo - vista - controlador. En nuestro caso, la diferencia es que existen dos controladores, el primero (**controlador aleatorio**) carga datos en el modelo web desde una tabla estática almacenada en formato “csv”, que contiene películas para etiquetar. El segundo controlador (**controlador cartelera**) obtiene las películas actuales en cartelera mediante una llamada a la API de TMDB (The Movie Database).

Los datos extraídos por ambos controladores han de pasar por el **módulo de la red neuronal** para obtener las predicciones de afinidad, consta de tres partes:

módulo de **preparación** (ejecuta la limpieza y selección de los datos), módulo de **procesamiento** (codifica los textos), módulo de la **red neuronal** (carga la red neuronal guardada y realiza las predicciones).

El **modelo web** se encarga tener actualizada la información a pintar en la **vista**. Si se detectan cambios en la vista, por ejemplo, si el usuario ha marcado una etiqueta, el controlador actualizará la información en el modelo web. En el caso de llegar a cuatro películas etiquetadas, el controlador utilizará el módulo de la red neuronal para reentrenarla.

4.3.2. Diseño de la red neuronal

En este apartado expondremos las partes que conforman la estructura de nuestra red neuronal. En la figura 4.2 podemos ver que la red consta de cuatro partes diferenciadas: el embedding, la red convolucional, la red recurrente y la red multicapa que origina la salida y posterior cálculo de la perdida. En la sección de implementación se dan los detalles de cómo construirla.

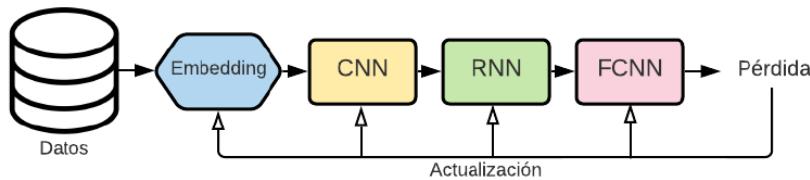


Figura 4.2: Estructura de la red neuronal.

Entrada y embedding En la figura 4.3 se ve como partimos de un vector de palabras de diferentes longitudes (n), resultado de aplicar las técnicas descritas sobre las sinopsis. En el paso (1) se llevan los vectores de palabras a vectores numéricos mediante la codificación basada en índices, se añade un padding de 0s al inicio para igualar las longitudes de los vectores a 90. Los datos de entrada de nuestra red serán vectores de números naturales $[x_1, x_2, \dots, x_{90}]$.

En la primera capa de la red (sin contar la de entrada), capa de embedding, se produce la transformación de los índices de cada palabra en su representación en forma de vector de números reales de longitud 64 (paso 2). Esto se realiza mediante la indexación en la matriz embedding, que contiene los pesos que calculará la red. Finalmente nuestro texto será una matriz $x \in \mathbb{R}^{90 \times 64}$.

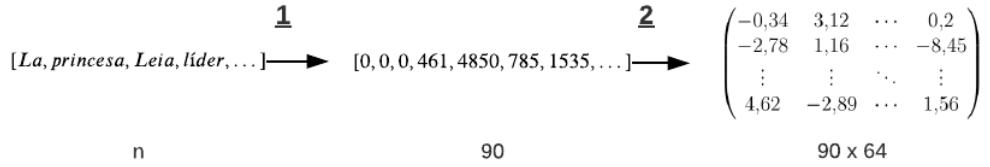


Figura 4.3: Ejemplo de codificación del texto.

Red convolucional Inmediatamente después de la capa de embedding se encuentra una estructura de red convolucional. Esta compuesta de dos módulos de capa convolucional 1D y maxpooling 1D, el primer módulo recibe el texto en forma de matriz de tamaño 90×64 , desde el embedding, y el segundo módulo devuelve otra matriz de tamaño 45×32 .

Red recurrente A continuación de la red convolucional nos encontramos con una red recurrente, en este caso está formada por una capa bidireccional. Consiste en dos LSTM (con dimensión de salida 16) concatenadas, en las que una de las LSTM procesa la secuencia en una dirección y la otra en el sentido contrario. Por lo tanto el resultado devuelto es un vector de tamaño 32.

Red multicapa y función de pérdida El último elemento de nuestra red es una red multicapa (FNN), que recibe un vector de tamaño 32, y está compuesta de tres capas densas, la última de ellas con una sola neurona y una función de activación sigmoide que devuelve valores entre 0 y 1.

Con esta probabilidad obtenida en la salida, calcularemos la función de pérdida que permite actualizar los pesos de la red, midiendo la diferencia entre el dato devuelto y la etiqueta real. En nuestro caso la función de pérdida utilizada es la entropía cruzada binaria (*binary cross-entropy*), cuya fórmula es:

$$H_p(q) = \frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i)) \quad (4.1)$$

Donde y_i es la etiqueta (1 para las películas buenas y 0 para las malas) y $p(y_i)$ es la probabilidad predicha de que la película sea buena. Se tienen en cuenta las N películas que entran como batch a la red.

4.4. Implementación

Una vez vista la arquitectura de la solución, nos centraremos en detallar la implementación de la red neuronal. Todo el código referenciado durante esta sección se encuentra en el apéndice A.

El entrenamiento y construcción del modelo se ha llevado a cabo desde un notebook que aplica una serie de funciones que se encuentran el script `methods.py`. Los métodos (tabla 4.1) se han extraído del notebook principal para facilitar su testing y tener estructurada cada HU a desarrollar.

Métodos	Descripción
read_data	Lee los datos.
preprocessing	Procesa el texto.
normalize	Normaliza el texto.
delete_stop_words	Elimina stop words.
stem_sentence	Realiza el stemming.
split_train_test	Separa la muestra.
tokenize	Codifica el texto.
my_model	Red neuronal.
train_model	Entrena la red.
plot_history	Dibuja las etapas.
plot_metrics	Imprime las métricas.
get_roc_curve	Dibuja la curva ROC.
plot_confusion_matrix	Dibuja la matriz de confusión.

Tabla 4.1: Métodos.

Las librerías principales que se han utilizado han sido Pandas, Tensorflow, Keras, Sklearn y NLTK (código A.2).

Para la lectura del dataset junto a sus etiquetas, se ha implementado una función (código A.1) que lee un “csv”, asigna los nombres de columnas necesarios y filtra los datos para que no existan sinopsis muy cortas en el conjunto.

Con el objetivo de aplicar la condensación de información, para reducir la longitud de los textos y la variabilidad de palabras, hemos implementado un método de preprocessing (código A.6), que consta a su vez de tres métodos: el de normalización (código A.3) que normaliza los textos eliminando acentos y signos de puntuación y llevándolo a minúsculas; el método de eliminación de palabras vacías (código A.4); y el método de stemming (código A.5), que ejecuta el algoritmo descrito en el apartado 4.2 sobre cada palabra de cada texto.

Sobre los textos preparados ejecutaremos la separación entre la muestra de entrenamiento y de test/validación (código A.7), al no tener disponer de dema-

siado volumen de sinopsis se ha optado porque el dataset de test y validación sea el mismo, tomando un 30 % de todo el dataset.

Con las muestras de train y test procederemos a la codificación del texto. La codificación se genera únicamente a partir de los datos de train, que es la única información que debe conocer el modelo. Se instancia un tokenizador y se entrena con las palabras del train, el tokenizador convierte los textos en secuencias según el índice de las palabras, de la manera que explicamos en el diseño de la red, añadiendo al inicio de la secuencia un padding de 0s. Se puede ver en el código [A.8](#).

Ahora trataremos la implementación de nuestra red neuronal (código [A.9](#)). Los textos de entrada (vectores de longitud 90) son mapeados en matrices de 90x64 gracias a la capa embedding. A partir de aquí dividiremos la red en cuatro bloques.

El primer bloque consta de:

- Una convolución 1D, con un kernel de tamaño 3x64 y *strides* 1, 32 filtros y *padding* “same”. Devuelve una matriz de tamaño 90x32.
- Una capa de normalización.
- Un maxpooling 1D, con un filtro de tamaño 2x1 y *strides* 2. Al aplicarse la reducción a las 32 columnas se devuelve una matriz de tamaño 45x32.

El segundo bloque consta de:

- Una convolución 1D, con un kernel de tamaño 3x32 y *strides* 1, 32 filtros y *padding* “same”. Devuelve una matriz de tamaño 45x32.
- Una capa de normalización.
- Un maxpooling 1D, con un filtro de tamaño 2x1 y *strides* 2. Al aplicarse la reducción a las 32 columnas se devuelve una matriz de tamaño 22x32.

El tercer bloque consta de:

- Una capa bidireccional.
- Dos LSTMs concatenadas, cada una procesa la secuencia de tamaño 22 con 32 características en un sentido. Cada LSTM tiene 16 unidades (dimensión del estado de la celda). Devuelve un vector de tamaño 32 ya que únicamente nos interesa el último estado procesado.
- Una capa de normalización.

El cuarto bloque consta de:

- Una capa densa de 16 neuronas, una capa de normalización y una capa de abandono de 0.5 (elimina el 50 % de las conexiones para evitar sobreajuste)
- Una capa densa de 8 neuronas, una capa de normalización y una capa de abandono de 0.5 (elimina el 50 % de las conexiones para evitar sobreajuste)
- Una capa densa de salida de 1 neurona, con función de activación sigmoide.

La función de pérdida es la entropía cruzada binaria (*binary cross-entropy*), el optimizador elegido para entrenar la red es *Adam*, una variante del algoritmo de descenso del gradiente que permite tener una tasa de aprendizaje adaptativa (*learning rate*).

Para entrenar nuestra red, se ha creado un método (código A.10) al que es necesario pasarle el número de interacciones de entrenamiento (épocas) y el tamaño del lote (*batch size*), así como los datos de entrenamiento y validación. Además se ha definido una parada anticipada (*early stopping*), que nos permite detener el entrenamiento cuando la pérdida deja de disminuir durante n épocas definidas.

Todas estas funciones explicadas hasta ahora son llamadas desde el [notebook](#) (código A.11) que ejecuta todo el proceso, desde la lectura de datos hasta finalizar el entrenamiento de la red con su posterior validación. En él se definen los parámetros necesarios: longitud de la secuencia (90), tamaño del embedding (64), learning rate (0.001), épocas (60) y batch size (4).

4.5. Pruebas

Para desempeñar la metodología de trabajo TDD se han implementado una serie de test unitarios que prueban las funciones vistas anteriormente, estos test se escribieron antes que la propia implementación de los métodos, con esto nos aseguramos que cuando nuestros métodos pasen los test estamos cumpliendo con todos y cada uno de los requisitos definidos.

Los test se encuentran dentro del script [test.py](#), en el que se definen una serie de clases, tantas como métodos a probar. Al ejecutarlo o instanciarlo veríamos si los métodos cumplen o no las condiciones necesarias. Se pasan un total de 13 test de manera automática en cada compilación del proyecto.

Las pruebas se han definido usando la librería de test unitarios `unittest`, adaptándonos a su forma de uso y estilo. Algunos test se ejecutan sobre los datos originales y otros sobre una muestra de datos de testing.

En total existen cinco clases determinadas que contienen el total de los test. Se puede ver una descripción de cada clase en la tabla 4.3.

Clases	Descripción
TestDataset	Prueba que el dataset de partida cumple las características necesarias.
TestPreprocessing	Prueba que se aplican las técnicas de preparación de manera correcta.
TestSplit	Prueba que la división (train / test) mantiene todas las proporciones.
Test_tokenize	Prueba la codificación basada en índices y el tamaño del vocabulario.
TestModel	Prueba la red neuronal (estructura, entrenamiento y predicciones).

Tabla 4.3: Clases de test.

5

Experimentos y resultados

En esta sección expondremos un análisis y visualización previa de los datos que justificarán algunas de las decisiones tomadas. Después se mostrarán un conjunto de experimentos aplicando la red prototipo vista y variantes de la misma sobre dos datasets diferentes, el de sinopsis y el IMDb dataset, que contiene 50000 reviews de películas (positivas y negativas).

5.1. Métricas

Primero vamos a introducir las métricas que se usarán durante los experimentos, sirven para evaluar el rendimiento de cada uno de los modelos.

Matriz de confusión Es una matriz que permite visualizar el rendimiento de un modelo en el aprendizaje supervisado (figura 5.1). Las filas contienen los valores reales de la clase y las columnas los valores predichos.

Accuracy (exactitud) Mide el porcentaje de casos que el modelo ha acertado.

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

		VALORES PREDICCIÓN	
		0	1
VALORES REALES	0	Verdaderos Negativos	Falsos Positivos
	1	Falsos Negativos	Verdaderos Positivos
		True Negative (TN)	False Positive (FP)
		False Negative (FN)	True Positive (TP)

Figura 5.1: Matriz de confusión (original).

Precision (precisión) Mide la calidad del modelo en relación a los falsos positivos.

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

Recall (exhaustividad) Informa sobre la cantidad de positivos que el modelo es capaz de identificar.

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

Se conoce como tasa de verdaderos positivos (*True Positive Rate*, TPR). Además se puede calcular la tasa de falsos positivos (*False Positive Rate*, FPR), dada por la fórmula $\frac{FP}{FP+TN}$.

F1 Es una métrica que combina la precision y el recall, nos permite valorar el rendimiento en ambas.

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Curva ROC Cada valor de TPR y FPR puede obtenerse para diferentes umbrales θ , que son los puntos de corte aplicados sobre la probabilidad predicha para asignar la etiqueta, normalmente $\theta = 0,5$. Al bajar este valor a $\theta < 0,5$ un mayor número de elementos serán clasificados como positivos, aumentando los valores de TP y FP mientras que los otros disminuyen.

Al pintar el TPR frente al FPR, para todos los valores $\theta \in [0, 1]$, obtenemos la curva ROC (*Receiver operating characteristic*), curva azul de la figura 5.2. Cuanto más separada esté del eje mejor será el modelo. También se puede calcular el área bajo la curva (AUC), un modelo perfecto tendrá un AUC próximo a 1 y un modelo equivalente a tirar una moneda al aire tendrá un AUC de 0,5. El punto de la curva más próximo a (0,1) nos permite obtener el θ óptimo con el mayor

TPR para el menor FPR posible (maximiza TPR - FPR).

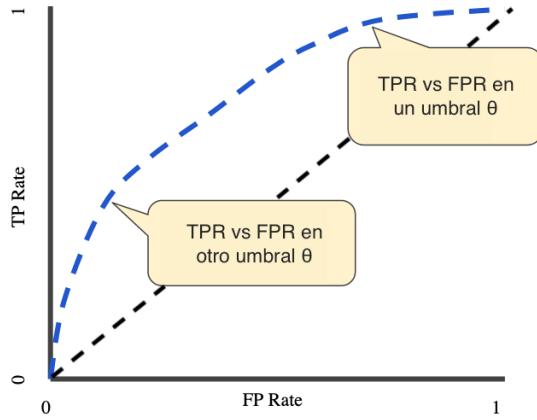


Figura 5.2: Curva ROC.

5.2. Análisis exploratorio

Nuestros datos iniciales constan de 2043 sinopsis de películas, de entre las cuales el reparto de películas marcadas como buenas (*like*) es el 45,2% y malas (*dislike*) el 54,8%. Con esto intuimos que nuestro modelo puede tener una ligera tendencia a marcar más películas como malas.

Al analizar la distribución de las longitudes de las sinopsis, se ve que la media se sitúa entorno a 80 palabras, incluyendo mayor cantidad de sinopsis cortas que de largas, como se observa en el histograma 5.3, además en la comparación del gráfico de densidad por etiquetas (figura 5.4) no se aprecian grandes diferencias.

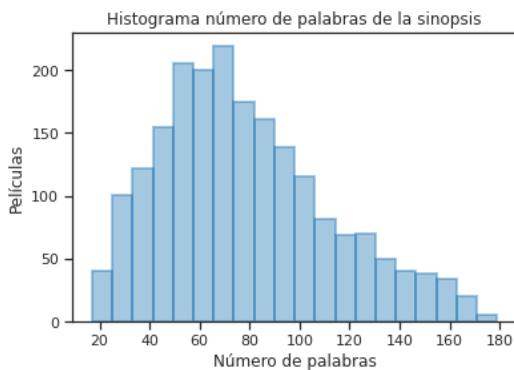


Figura 5.3: Histograma.

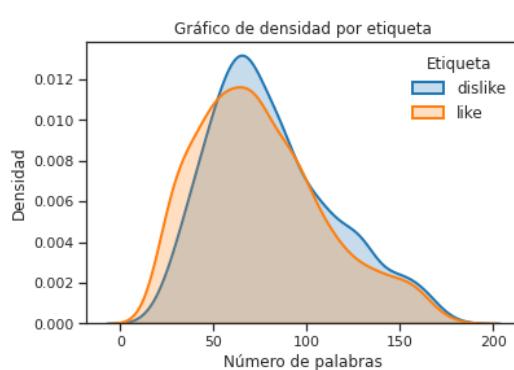


Figura 5.4: Gráfico de densidad.

Como primera toma de contacto y para ver que contienen nuestros textos, se ha dibujado una nube de palabras (word cloud). En la nube (figura 5.5) se

diferencian en grande aquellas palabras que tienen una mayor “frecuencia” o importancia en el conjunto de las sinopsis. Separando por etiquetas, vemos que las películas marcadas como malas (figura 5.7) son las relacionadas con policía, amor, asuntos personales, etc. Mientras que en la figura 5.6 observamos que las películas buenas son las que hablan de la tierra, el futuro, la fuerza o viajes.



Figura 5.5: Wordcloud completo.



Figura 5.6: Wordcloud dislike.



Figura 5.7: Wordcloud like.

Veamos una comparativa entre las distintas alternativas para la preparación y la condensación de información (figura 5.8). Con cada una de ellas conseguimos tener en cada iteración un vocabulario menor, comenzando por 28558 palabras diferentes y alcanzando las 11530, la tercera parte. Las técnicas de lemmatization y stemming están aplicadas después de las dos primeras (normalización + stop words), la que consigue mejores resultados es el stemming, incluso aplicando ambas no se consigue un resultado mucho mejor (lemmatization + stemming).

Se trató de hacer la separación de datos en las 3 muestras deseadas (train, test y validación) pero no mantenían densidades similares de distribución (en general

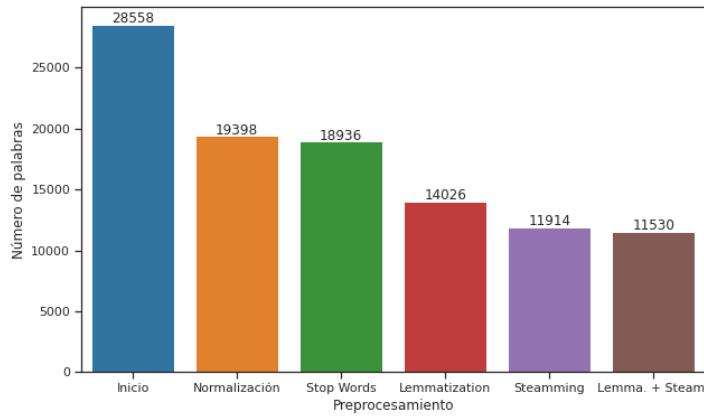


Figura 5.8: Condensación de textos.

y tras separar por etiquetas), la validación y el test resultante no eran muestras representativas del train. Lo mismo ocurría con un test menor al 30 % del dataset original [25]. Por lo tanto, los resultados de los experimentos, que se expondrán a continuación, se muestran sobre los mismos datos usados para la validación.

5.3. Red prototipo

El primer experimento se realizó aplicando la red vista en la sección de implementación (4.4). Primero expondremos los resultados obtenidos sobre nuestro dataset de sinopsis original y, a continuación, se mostrará también el rendimiento obtenido de aplicar la misma red sobre el [IMDb dataset](#), en el que realizamos un análisis de sentimiento.

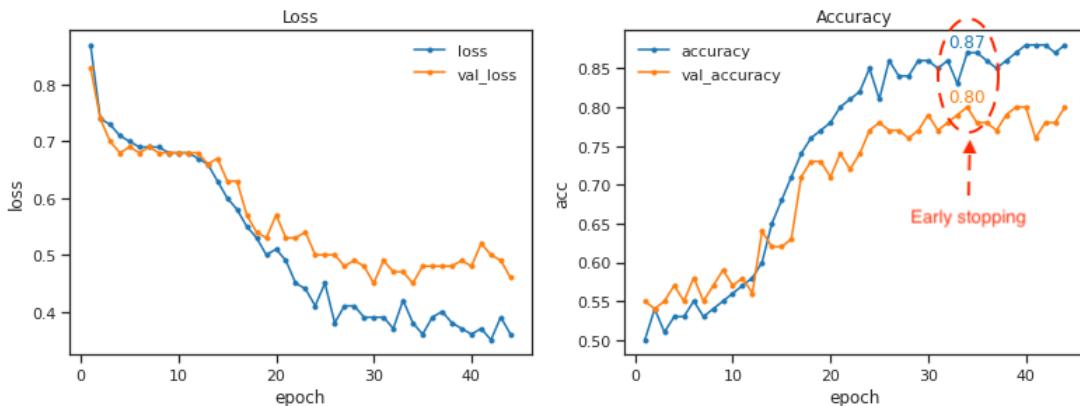


Figura 5.9: Entrenamiento de la red original.

El modelo se entrenó durante 44 épocas, parando en la 34 debido al *early stopping* (10 épocas de espera), la gráfica de entrenamiento con la pérdida y la exactitud (*accuracy*), para el train y validación, la encontramos en la figura 5.9. En el punto óptimo, el accuracy de la validación fue del 80 %. Al tratarse de un problema “complejo” de resolver, ya que una película te pude gustar (o no) por factores diferentes a la sinopsis (actores, dirección, montaje...), es difícil determinar si el accuracy obtenido es bueno o no. Aproximarse al 100 % de accuracy puede no ser factible para nuestro problema.

El resto de métricas se pueden ver en la tabla 5.1. Se han obtenido partiendo de la matriz de confusión de la figura 5.10 (izquierda), calculada teniendo en cuenta que una probabilidad predicha igual o mayor a 0,5 equivale a la etiqueta 1 y el resto a la etiqueta 0. Keras evalúa el modelo usando dicho umbral de corte ($\theta = 0,5$) y por lo tanto es el que logra el mayor accuracy.

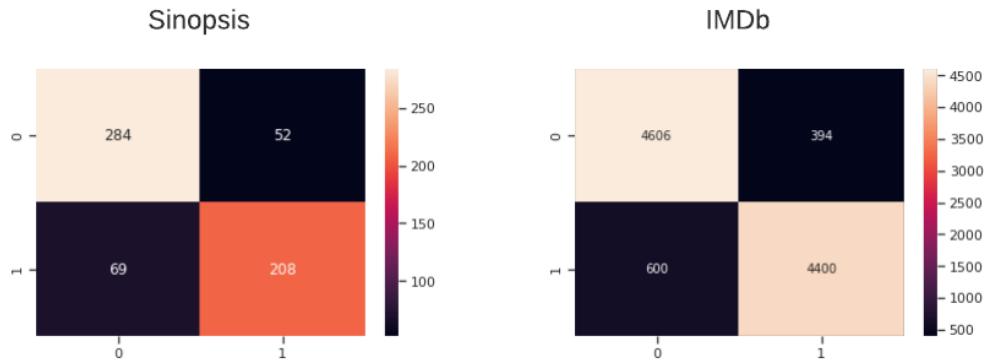


Figura 5.10: Matrices de confusión red prototipo.

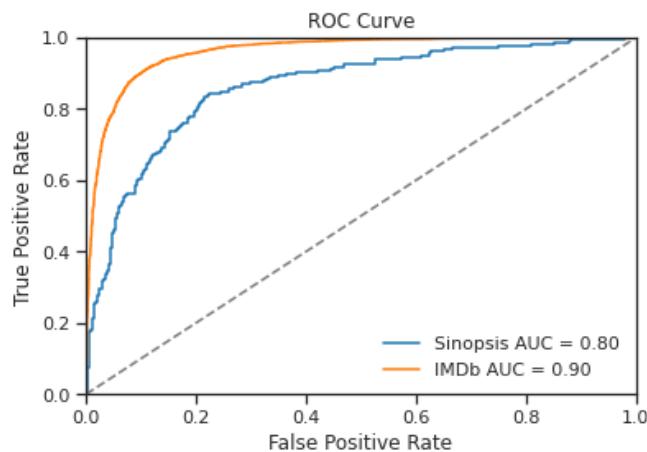


Figura 5.11: Curva ROC red prototipo.

La curva ROC (curva azul de la figura 5.11) nos permite obtener un umbral óptimo $\theta = 0,45$, el cual maximizaría el recall con la tasa de falsos positivos más baja posible, pero penalizaría la precisión y el accuracy. El AUC resultante es del 80 % y nos indica que el modelo dista de ser aleatorio ($AUC = 50\%$).

Hemos aplicado esta misma arquitectura de red neuronal sobre el IMDb Dataset, un conjunto de datos empleado habitualmente como *benchmark* (referencia) de los modelos y comparador del rendimiento que nos ayudará a catalogar mejor nuestra red.

El dataset está formado por 25000 reviews de películas negativas (0) y 25000 positivas (1), se ha separado un 80 % para el train y un 20 % para test manteniendo la proporción de etiquetas.

El modelo se ha entrenado con los siguientes parámetros: longitud de secuencia (512), tamaño del embedding (64), épocas (6), *learning rate* (0.001) y batch size (16). Paró en la tercera época debido al early stopping asignado (3 épocas de espera). Los resultados se observan en la tabla 5.2 y la curva ROC en la figura 5.11 con un AUC del 90 %. El umbral óptimo que se obtiene de la curva es $\theta = 0,46$.

Todos los parámetros definidos durante este primer experimento, para ambos datasets, se aplican también en el resto.

5.4. Modificación ResNet

Para el segundo experimento se ha introducido una variante sobre el bloque de red convolucional, se ha modificado la CNN por una red de tipo Resnet. El nuevo bloque de CNN-Resnet ahora consta de:

- Una capa de convolución 1D plana, igual a la vista en la sección de implementación 4.4.
- Dos unidades residuales (*Residual unit*), como la vista en la figura 3.8 donde la convolución tiene 32 filtros de tamaño 3 y *strides* 1.
- Un maxpooling 1D, igual al visto en la sección de implementación 4.4..

El resto de la red se ha mantenido igual a la arquitectura prototipo.

Con el dataset de sinopsis, el modelo resultante se entrenó durante 29 épocas, obteniendo los resultados que se observan en la tabla 5.1. Logramos un AUC (figura 5.13) y un accuracy del 82 % en ambos casos, mejorando así al modelo prototipo. El valor umbral óptimo es $\theta = 0,4$.

Para el IMDb dataset, los resultados (tabla 5.2) también han mejorado pasando a tener un 91 % de accuracy y un 91 % de AUC tras 4 épocas de entrenamiento. El valor umbral óptimo es $\theta = 0,5$.

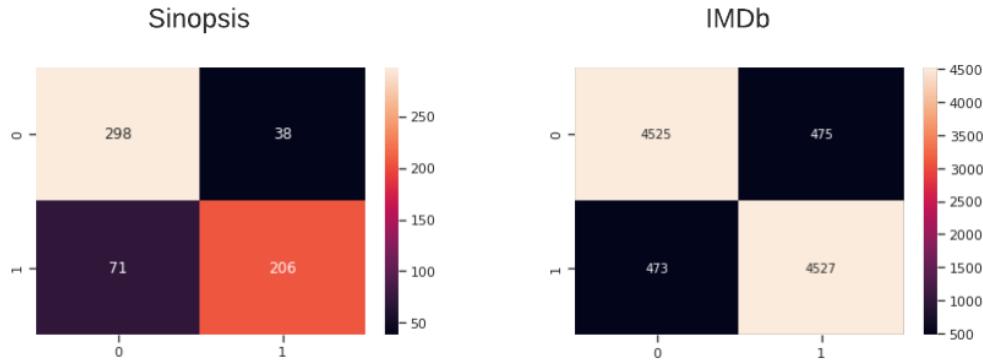


Figura 5.12: Matrices de confusión red ResNet.

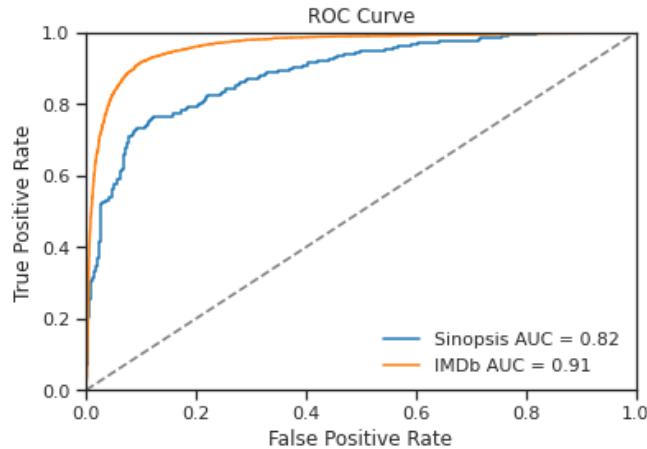


Figura 5.13: Curva ROC red ResNet.

5.5. Modificación GRU Bidireccional

En el tercer experimento hemos modificado la LSTM bidireccional por una GRU bidireccional, manteniendo las 16 unidades definidas inicialmente para cada LSTM. El módulo CNN se ha mantenido igual al de la red prototipo.

Para el dataset de sinopsis, tras 40 épocas de entrenamiento, conseguimos peores resultados en todas las métricas (tabla 5.1). La curva ROC (figura 5.13) nos da un umbral óptimo $\theta = 0,41$.

En el IMDb dataset se consigue mejorar únicamente la métrica de precisión al 93 % (tabla 5.2), entrenando la red durante 5 épocas. El umbral óptimo que obtenemos es $\theta = 0,44$.

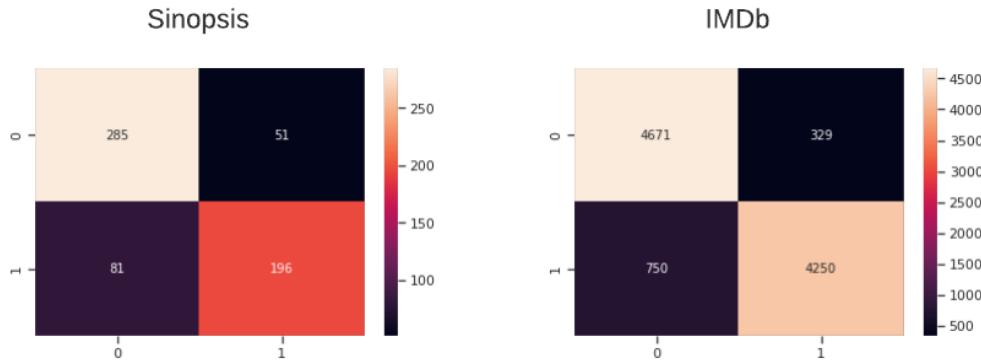


Figura 5.14: Matrices de confusión red GRU Bidireccional.

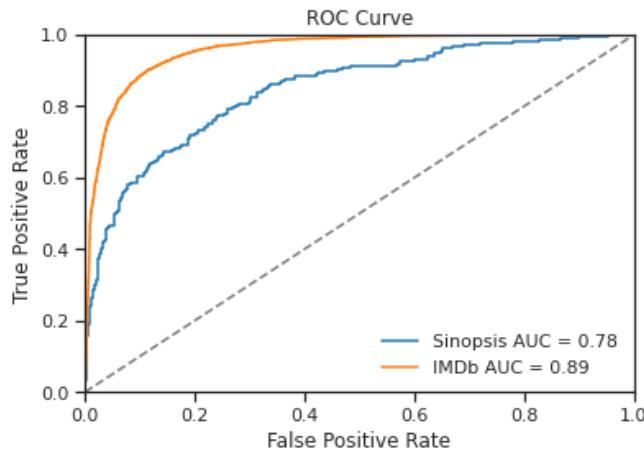


Figura 5.15: Curva ROC red GRU Bidireccional.

5.6. Modificación GRU + GRU

En este último experimento se ha modificado la forma en la que se concatenan las dos GRU. En lugar de utilizar una capa bidireccional se han ordenado de manera secuencial una a continuación de la otra. Para ello es necesario que la primera GRU devuelva el resultado de cada secuencia procesada y no solo de la última. De la segunda GRU sí que queremos mantener como salida únicamente el último estado, un vector de dimensión 16. La CNN se ha mantenido como en la red inicial.

Al aplicar la red al dataset de sinopsis logramos una mejora del recall pasando al 79 % (tabla 5.1) tras 26 épocas de entrenamiento. La curva ROC nos devuleve un umbral óptimo $\theta = 0,55$.

Sobre el IMDb dataset se mejora igualmente el recall al 92 % (tabla 5.2) después de 3 épocas. El umbral óptimo resultante es $\theta = 0,53$.

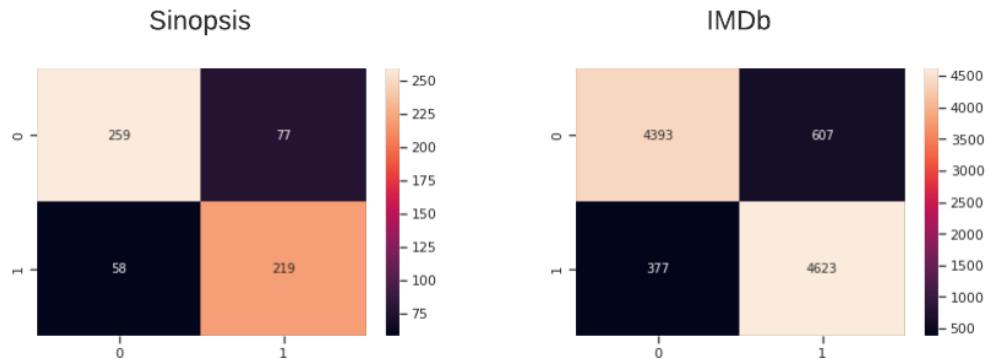


Figura 5.16: Matrices de confusión red GRU + GRU.

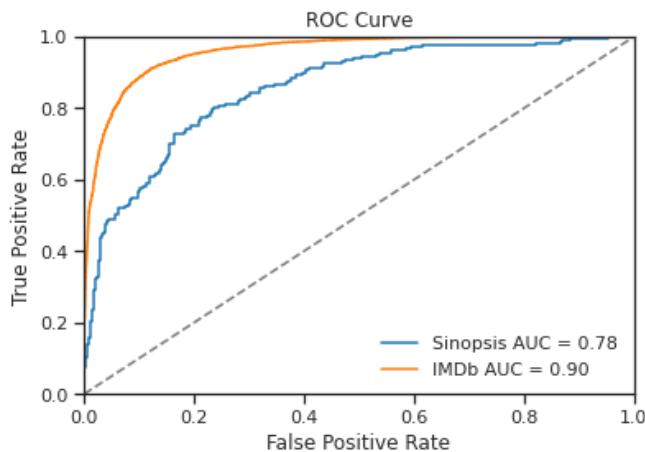


Figura 5.17: Curva ROC red GRU + GRU.

5.7. Resultados finales

Finalmente, vamos a exponer las conclusiones obtenidas del conjunto de los cuatro experimentos.

Observado los resultados para cada experimento (tablas 5.1 y 5.2), podemos determinar que el mejor modelo y el más equilibrado, para los dos datasets, es la

Mé. Ex.	Prototipo	ResNet	Bi GRU	GRU+GRU
Accuracy	80.3	82.2	78.5	78.0
AUC	79.8	81.5	77.8	78.0
Precision	80.0	84.4	79.4	74.0
Recall	75.1	74.4	70.8	79.1
F1	77.5	79.1	74.8	76.4

Tabla 5.1: Resultados sinopsis dataset (Mé: Métrica, Ex: Experimento).

Mé. Ex.	Prototipo	ResNet	Bi GRU	GRU+GRU
Accuracy	90.1	90.5	89.2	90.2
AUC	90.1	90.5	89.2	90.2
Precision	91.8	90.5	92.8	88.4
Recall	88.0	90.5	85.0	92.5
F1	89.9	90.5	88.7	90.4

Tabla 5.2: Resultados IMDb dataset (Mé: Métrica, Ex: Experimento).

modificación ResNet, es la que mejor accuracy y AUC ofrece en ambos casos. Por lo tanto, podemos asegurar así, que tiene un mayor impacto sobre el resultado final el modificar el bloque convolucional frente a cambiar el bloque recurrente.

De la variante GRU + GRU se observa que es la que mayor recall obtiene para ambos datasets, pudiendo ser considerada así como una buena arquitectura para priorizar dicha métrica.

Una alternativa para optimizar el recall o la precisión es variar el umbral de corte del mejor modelo (ResNet). Como variante a la curva ROC se puede pintar la curva de precisión vs recall y elegir θ en función de dicha curva. Se elegirá un θ con mayor recall cuando el usuario tenga prioridad por no perderse ninguna película buena, aunque tenga que verse algunas malas y se elegirá un θ con mayor precisión cuando el usuario quiera ver únicamente películas buenas, aunque pueda perderse alguna de ellas.

Para el dataset de sinopsis, el AUC obtenido de 81,5 % lo podemos considerar realmente un buen resultado. Y el AUC de 90,5 % en el IMDb dataset se puede considerar como un resultado muy bueno.

Comparando los resultados obtenidos dentro del [IMDb Benchmark](#), nuestro modelo se encontraría dentro del top 10 de los modelos que no usan datos extra para el entrenamiento, gracias al 90.5 % de accuracy logrado.

Todos los notebooks con los que se han realizado los experimentos se encuentran dentro de la carpeta [experiments](#) del proyecto en Github.

6

Conclusiones

Tras explicar el proyecto solo queda exponer las conclusiones extraídas y los logros conseguidos de su realización. Además de presentar nuevas posibles mejoras que no se han podido abordar durante este trabajo.

6.1. Logros

Un trabajo de fin de grado ha de estar enfocado en la adquisición de competencias vistas durante el grado, desarrollando y aplicando las metodologías estudiadas durante el mismo. Un TFG también debe centrarse en la obtención de nuevos conocimientos sobre ámbitos vistos durante las asignaturas que conforman el plan de estudios. Este proyecto se centra en una línea de trabajo dentro del área del aprendizaje automático, ámbito visto durante la asignatura de Ingeniería del Conocimiento de manera superficial y más si hablamos de la rama del procesamiento del lenguaje natural, en la cual hemos enfocado nuestro proyecto.

Durante el proyecto hemos trabajado con un lenguaje de programación no visto durante los estudios (Python), incluyendo además algunas sus librerías de aprendizaje automático como Keras y de programación web como Flask. Se han aprendido técnicas de NLP para el procesamiento de los textos y su codificación. Además se han ampliado conceptos en redes neuronales como las convoluciones y recurrentes, donde hemos tenido que hacer un trabajo de investigación para su entendimiento e implementación. Y finalmente hemos tenido que aplicar metodologías de trabajo ágiles con la limitación que supone su utilización para trabajos

en solitario.

Con la finalización de nuestro proyecto hemos conseguido elaborar un modelo que es capaz de devolver la afinidad que tiene un usuario a una película, todo ello empleando únicamente la sinopsis de la misma. A pesar de la ya compleja elaboración del modelo en sí, también hemos trabajado en su puesta en producción dentro de una aplicación web, para que se encuentre ubicado en un entorno real y con utilidad final para el usuario general.

En resumen, se han aprendido nuevas tecnologías, lenguajes y librerías como son Python, Keras y Flask. Además se han ampliado nuevos conceptos en redes neuronales y NLP. Y todo ello aplicando metodologías de desarrollo ágiles.

6.2. Trabajos futuros

En este último punto vamos abordar el futuro de nuestro proyecto, ya que está ideado para ser evolucionado e incorporado dentro de un proyecto más grande y vivo.

Las posibles ampliaciones y mejoras futuras son:

- Optimización de la red neuronal mediante la modificación de sus parámetros o aplicando nuevas arquitecturas con capas de atención.
- Mejora el esquema de la red aplicando *transformers*, que permitan aprender de la propia estructura del lenguaje de las sinopsis.
- Aplicación de aprendizaje por transferencia (*transfer learning*) para que el modelo inicial no necesite de una cantidad grande de datos. Se basa en usar modelos ya entrenados y optimizarlos para tu problema.
- Incorporar la capacidad de procesar imágenes en la red neuronal para que tenga en cuenta el cartel de la película.
- Mejora de la interfaz para facilitar la interacción por parte del usuario.
- Incorporar nuevas funciones en la web para que el usuario pueda hacer búsqueda directa de películas, por género, por año, etc.
- Desarrollar una aplicación móvil a partir de la web ya mejorada.

Bibliografía

- [1] “Incorporaciones: Traslado a un espacio de dimensiones bajas.” [Online]. Available: <https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space?hl=es>
- [2] V. Nigam, “Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning,” Jan. 2021. [Online]. Available: <https://medium.com/analytics-vidhya/understanding-neural-networks-from-neuron-to-rnn-cnn-and-deep-learning-cd88e90e0a90>
- [3] C. Olah, “Understanding LSTM Networks – colah’s blog.” [Online]. Available: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- [4] A. Geron, *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems.* Sebastopol, CA: O'Reilly Media, 2017.
- [5] M. Moore, “Netflix shows would take you four years to watch,” 2019. [Online]. Available: <https://www.thetimes.co.uk/article/netflix-shows-would-take-you-four-years-to-watch-bcksqb7b0>
- [6] A. L. Gatos, “Netflix gana un 48 % más en 2020 y supera los 200 millones de abonados,” Jan 2021. [Online]. Available: <https://elpais.com/economia/2021-01-20/netflix-gana-un-48-mas-en-2020-y-supera-los-200-millones-de-abonados.html>
- [7] “Cómo funciona el sistema de recomendaciones de Netflix.” [Online]. Available: <https://help.netflix.com/es-es/node/100639>
- [8] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, “Deep contextualized word representations,” 2018.
- [9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2017.
- [10] A. Graves, “Sequence transduction with recurrent neural networks,” 2012.
- [11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2019.

BIBLIOGRAFÍA

- [12] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” 2020.
- [13] “A robot wrote this entire article. are you scared yet, human? — gpt-3,” Sep 2020. [Online]. Available: <https://www.theguardian.com/commentisfree/2020/sep/08/robot-wrote-this-article-gpt-3>
- [14] H. Takeuchi and I. Nonaka, “The new new product development game,” *Harvard Business Review*, 1986.
- [15] Javier Garzas, “¿Qué es eso de ATDD?” Jul. 2015. [Online]. Available: <https://www.javiergarzas.com/2015/07/que-es-eso-de-atdd.html>
- [16] Z. S. Harris, “Distributional structure,” *WORD*, vol. 10, no. 2-3, pp. 146–162, 1954.
- [17] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” 2013.
- [18] J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation,” pp. 1532–1543, 2014.
- [19] Q. Liu, M. J. Kusner, and P. Blunsom, “A survey on contextual embeddings,” 2020.
- [20] A. C. Infante, *Redes neuronales clásicas*. Reconocimiento de Patrones, 2018.
- [21] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, pp. 1735–80, 12 1997.
- [22] H. P. Luhn, “Key word-in-context index for technical literature (kwic index),” *American Documentation*, vol. 11, no. 4, pp. 288–295, 1960.
- [23] M. Porter, “An algorithm for suffix stripping,” *Program*, vol. 14, no. 3, pp. 130–137, Jan. 1980.
- [24] “Spanish stemming algorithm - Snowball.” [Online]. Available: <https://snowballstem.org/algorithms/spanish/stemmer.html>
- [25] A. NG, “Machine learning yearning,” pp. 14–19, 2018.

Apéndices

A

Códigos

A.1. Redes neuronales

Listing A.1: Lectura del dataset.

```
1 def read_data(file, overview_column='overview', label_column='like
  '):
2     df = pd.read_csv(file, sep='#', encoding='utf-8', lineterminator=
      '\n')
3     df = df.rename(columns={overview_column: 'overview',
        label_column: 'like'})
4     df['text_array'] = df.overview.str.split(" ")
5     df['n_words'] = df['text_array'].apply(lambda x: len(x))
6     df = df.drop(columns=['text_array'])
7     df = df[df['n_words']>=5]
8
9     return df
```

Listing A.2: Importar librerías.

```

1 import pandas as pd
2 import numpy as np
3 import tensorflow as tf
4 import nltk
5 from nltk.tokenize import WordPunctTokenizer
6 from nltk import word_tokenize
7 from nltk.stem import SnowballStemmer
8 import keras
9 import re
10 import string
11 from collections import Counter
12 nltk.download("popular")
13 nltk.download('stopwords')
14 import matplotlib.pyplot as plt
15 import seaborn as sns
16 from sklearn.feature_extraction.text import CountVectorizer
17 from sklearn.model_selection import train_test_split
18 from sklearn.metrics import accuracy_score, f1_score,
   confusion_matrix, precision_score, recall_score, roc_curve,
   roc_auc_score
19 from tensorflow.keras.preprocessing.text import Tokenizer
20 from tensorflow.keras.preprocessing.sequence import pad_sequences
21 from tensorflow.keras.models import Sequential, Model
22 from tensorflow.keras import layers
23 from tensorflow.keras import regularizers

```

Listing A.3: Normalización de los textos.

```

1 def normalize(x):
2     x = x.lower()
3     replacements = (
4         ("á", "a"),
5         ("é", "e"),
6         ("í", "i"),
7         ("ó", "o"),
8         ("ú", "u"),
9         ("ñ", "n"))
10    for a, b in replacements:
11        x = x.replace(a, b)
12    x = x.translate(str.maketrans('', '', string.punctuation))
13    x = x.translate(str.maketrans(' ', ' ', 'aoíç@'))
14    x = x.replace(" ", " ")
15
16    return x

```

Listing A.4: Eliminación de Stopwords.

```
1 stop_words = pd.read_csv("data/stopwords-es.txt",header=None)
2 stop_words = stop_words[0].tolist()
3 stop_words = [normalize(word) for word in stop_words]
4 def delete_stop_words(x):
5     words = x.split(' ')
6     words = [word for word in words if word not in stop_words]
7     x = str(' '.join(words))
8
9 return x
```

Listing A.5: Stemming de los textos.

```
1 stemmer = SnowballStemmer("spanish", ignore_stopwords=True)
2 def stem_sentence(sentence):
3     stemmed_text = [stemmer.stem(word) for word in word_tokenize(
4         sentence)]
5     stemmed_text = " ".join(stemmed_text)
6
7 return stemmed_text
```

Listing A.6: Preparación de los textos.

```
1 def preprocessing(sentece):
2     sentece = normalize(sentece)
3     sentece = delete_stop_words(sentece)
4     sentece = stem_sentence(sentece)
5
6 return sentece
```

Listing A.7: Separación en train y test.

```
1 def split_train_test(df):
2     X = df['overview']
3     y = df['like']
4     X_train, X_test, y_train, y_test = train_test_split(X, y,
5             test_size=0.30, stratify=y, random_state=10)
6
7 return X_train, X_test, y_train, y_test
```

Listing A.8: Codificación de los textos.

```

1 def tokenize(X_train, X_test, maxlen=None, test_mode=False):
2     tokenizer = Tokenizer()
3     tokenizer.fit_on_texts(X_train)
4     X_train = tokenizer.texts_to_sequences(X_train)
5     X_test = tokenizer.texts_to_sequences(X_test)
6     real_maxlen = len(max(X_train, key=len))
7     if maxlen is None:
8         maxlen = real_maxlen
9     vocab_size = len(tokenizer.word_index) + 1 #Adding 1 reserved 0
10    index
11    if test_mode is False: # plot histogram tokens before padding
12        n_words_train = np.array([len(i) for i in X_train])
13        sns.distplot(n_words_train, hist=True, kde=False,
14                      bins=20,
15                      hist_kws={'edgecolor':'tab:blue', 'linewidth': 2})
16        plt.title('Histograma numero de palabras')
17        plt.xlabel('Numero palabras')
18        plt.ylabel('Peliculas')
19        mean = n_words_train.mean()
20        plt.vlines(mean, 0, 215, color='crimson', ls=':')
21    X_train = pad_sequences(X_train, padding='pre', maxlen=maxlen,
22                           truncating='post')
23    X_test = pad_sequences(X_test, padding='pre', maxlen=maxlen,
24                           truncating='post')
25    return X_train, X_test, vocab_size, real_maxlen

```

Listing A.9: Red Neuronal CNN + RNN.

```
1 def my_model(vocab_size, maxlen=90, embedding_dim=64, lr=0.001,
2     epochs=40):
3     inputs = layers.Input(shape=(maxlen,))
4     # Embedding
5     x = layers.Embedding(input_dim = vocab_size, output_dim =
6         embedding_dim)(inputs)
7     # CNN 1
8     x = layers.Conv1D(32, 3, padding="same", activation="relu",
9         strides=1)(x)
10    x = layers.BatchNormalization()(x)
11    x = layers.MaxPooling1D(pool_size=2, strides=2)(x)
12    # CNN 2
13    x = layers.Conv1D(32, 3, padding="same", activation="relu",
14        strides=1)(x)
15    x = layers.BatchNormalization()(x)
16    x = layers.MaxPooling1D(pool_size=2, strides=2)(x)
17    # LSTM Bi
18    x = layers.Bidirectional(layers.LSTM(16))(x)
19    x = layers.BatchNormalization()(x)
20    # FNN
21    x = layers.Dense(16, activation="relu")(x)
22    x = layers.BatchNormalization()(x)
23    x = layers.Dropout(0.5)(x)
24    x = layers.Dense(8, activation="relu")(x)
25    x = layers.BatchNormalization()(x)
26    x = layers.Dropout(0.5)(x)
27    # Output layer, 1 neuron with sigmoid:
28    predictions = layers.Dense(1, activation="sigmoid", name="predictions")(x)
29    model = tf.keras.Model(inputs, predictions)
30    model.compile(loss='binary_crossentropy', optimizer=tf.keras.
31        optimizers.Adam(learning_rate=lr, decay = lr/epochs), metrics
32        =['accuracy'])
33    return model
```

Listing A.10: Entrenamiento de la red.

```

1 def train_model(model, X_train, y_train, X_test, y_test, epochs,
2                 batch_size):
3     callback = tf.keras.callbacks.EarlyStopping(monitor='loss',
4                                                 patience=10)
5     history = model.fit(X_train, y_train,
6                          epochs=epochs,
7                          verbose=True,
8                          validation_data=(X_test, y_test),
9                          batch_size=batch_size,
10                         callbacks=[callback])
11
12 return history, model

```

Listing A.11: Ejecución completa.

```

1 import methods as f
2 from keras.utils.vis_utils import plot_model
3 #params
4 maxlen=90
5 embedding_dim=64
6 lr=0.001
7 epochs=60
8 batch_size=4
9 df = f.read_data("data/overviews_final.csv")
10 df['overview'] = df['overview'].progress_apply(lambda x: f.
11                                                 preprocessing(x))
12 X_train, X_test, y_train, y_test = f.split_train_test(df)
13 X_train, X_test, vocab_size, real maxlen = f.tokenize(X_train,
14                                                       X_test, maxlen)
15 model = f.my_model(maxlen=maxlen, embedding_dim=embedding_dim,
16                     vocab_size=vocab_size, lr=lr, epochs=epochs)
17 model.summary()
18 plot_model(model, to_file='model_plot.png', show_shapes=True,
19             show_layer_names=True)
20 history, model = f.train_model(model=model,
21                                 X_train=X_train, y_train=y_train,
22                                 X_test=X_test, y_test=y_test,
23                                 epochs=epochs, batch_size=
24                                 batch_size)

```

A.2. Pruebas

Listing A.12: Librerías testing.

```
1 import unittest
2 import pandas as pd
3 import numpy as np
4 import methods as f
5 original_df = 'data/overviews_final.csv'
6 testing_df = 'data/overviews_testing.csv'
```

Listing A.13: Test dataset.

```
1 class TestDataset(unittest.TestCase):
2     def setUp(self):
3         '''Set up an instance prior to every test execution'''
4         self.df = f.read_data(original_df)
5     def test_label_and_overview_column(self):
6         '''Test case function for label column'''
7         self.assertTrue('like' in self.df.columns)
8         self.assertTrue('overview' in self.df.columns)
9     def test_enough_labels(self):
10        '''Test case function for enough labels'''
11        like_1 = self.df.like.value_counts()[1]
12        like_0 = self.df.like.value_counts()[0]
13        self.assertIn(like_1, range(800,1200))
14        self.assertIn(like_0, range(800,1200))
15    def test_empty_overview(self):
16        '''Test case function for empty overview'''
17        self.assertFalse(any(x == True for x in self.df['n_words']<5))
```

Listing A.14: Test preparación.

```

1 class TestPreprocessing(unittest.TestCase):
2     def setUp(self):
3         '''Set up an instance prior to every test execution'''
4         self.sentence = 'La princesa Leia, líder del movimiento
5                         rebelde que desea restaurar la República.\u00d1@?\$\%'
6         self.new_sentence = f.preprocessing(self.sentence)
7     def test_normalize(self):
8         '''Test case function for normalize'''
9         def is_ascii(s):
10             return all(ord(c) < 128 for c in s)
11             self.assertTrue(self.new_sentence.islower())
12             self.assertTrue(is_ascii(self.new_sentence))
13     def test_stop_words(self):
14         '''Test case function for stop words'''
15         stop_words = pd.read_csv(stop_words_df, header=None)
16         stop_words = stop_words[0].tolist()
17         stop_words = [word.lower() for word in stop_words]
18         words = self.new_sentence.split(' ')
19         self.assertFalse(any([word for word in words if word in
20                             stop_words]))

```

Listing A.15: Test separación de datos.

```

1 class TestSplit(unittest.TestCase):
2     def setUp(self):
3         '''Set up an instance prior to every test execution'''
4         self.df = f.read_data(testing_df)
5         self.X_train, self.X_test, self.y_train, self.y_test = f.
6             split_train_test(self.df)
7     def test_prop(self):
8         '''Test case function for proportion'''
9         train_len = len(self.X_train)
10        test_len = len(self.X_test)
11        len_df = len(self.df)
12        prop_train = round(train_len/len_df, 2)
13        prop_test = round(test_len/len_df, 2)
14        self.assertEqual(prop_train, 0.7)
15        self.assertEqual(prop_test, 0.3)
16    def test_prop_label(self):
17        '''Test case function for label proportion'''
18        prop_train = round(self.y_train.value_counts(normalize=True)
19                           [1],1)
20        prop_test = round(self.y_test.value_counts(normalize=True)[1],
21                          1)
22        self.assertEqual(prop_train, prop_test)

```

Listing A.16: Test codificación de textos.

```

1 class TestTokenize(unittest.TestCase):
2     def setUp(self):
3         '''Set up an instance prior to every test execution'''
4         self.df = f.read_data(testing_df)
5         self.X_train, self.X_test, self.y_train, self.y_test = f.
6             split_train_test(self.df)
7         self.X_train, self.X_test, self.vocab_size, self.real maxlen =
8             f.tokenize(self.X_train, self.X_test, test_mode=True)
9     def test_array_of_ints(self):
10        '''Test case function for array of ints'''
11        self.assertEqual(self.X_train.dtype, 'int32')
12    def test_max_len(self):
13        '''Test case function for max len'''
14        self.assertEqual(len(self.X_train[0]), self.real maxlen)
15    def test_vocab_size(self):
16        '''Test case function for vocab size'''
17        self.assertEqual(np.amax(self.X_train), self.vocab_size - 1)

```

Listing A.17: Test Modelo.

```

1 class TestModel(unittest.TestCase):
2     def setUp(self):
3         '''Set up an instance prior to every test execution'''
4         self.df = f.read_data(testing_df)
5         self.X_train, self.X_test, self.y_train, self.y_test = f.
6             split_train_test(self.df)
7         self.X_train, self.X_test, self.vocab_size, self.real maxlen =
8             f.tokenize(self.X_train, self.X_test, test_mode=True)
9         self.model = f.my_model(maxlen=self.real maxlen, embedding_dim
10             =5,
11             vocab_size=self.vocab_size, lr=0.001,
12             epochs=1)
13     def test_model_structure(self):
14         '''Test case function for model structure'''
15         layers_names = [self.model.get_layer(index=i).name for i in
16             range(0, len(self.model.layers))]
17         self.assertTrue(any("conv" in s for s in layers_names))
18         self.assertTrue(any("lstm" in s for s in layers_names))
19         or any("bidirectional" in s for s in layers_names))
20         self.assertTrue(any("embedding" in s for s in layers_names))
21         self.assertTrue(any("dense" in s for s in layers_names))
22     def test_training(self):
23         '''Test case function for model training'''
24         history = self.model.fit(self.X_train, self.y_train, epochs=1,
25             batch_size=len(self.X_train))
26         self.assertTrue(len(history.history)>0)
27     def test_predictions(self):
28         '''Test case function for model predictions'''
29         self.model.fit(self.X_train, self.y_train, epochs=1,
30             batch_size=len(self.X_train))
31         y_pred = self.model.predict(self.X_test).flatten()
32         max_pred = y_pred.max()
33         min_pred = y_pred.min()
34         self.assertTrue((0<=max_pred<=1) and (0<=min_pred<=1))
35         self.assertEqual(len(y_pred), len(self.y_test))

```