

Trabajo fin de grado

Sistemas de recomendación de noticias basados
en aprendizaje profundo



Eric Morales Agostinho

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR



Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

**Sistemas de recomendación de noticias basados
en aprendizaje profundo**

Autor: Eric Morales Agostinho

Tutor: Alejandro Bellogín Kouki

junio 2021

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución comunicación pública y transformación de esta obra sin contar con la autorización de los titulares de la propiedad intelectual.
La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. del Código Penal*).

DERECHOS RESERVADOS

© 20 de Junio de 2021 por UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, nº 1
Madrid, 28049
Spain

Eric Morales Agostinho
Sistemas de recomendación de noticias basados en aprendizaje profundo

Eric Morales Agostinho
C\ Francisco Tomás y Valiente N° 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

RESUMEN

Junto con el crecimiento de los medios de noticias digitales se ha producido una irremediable sobrecarga de información hacia los usuarios. Para solucionarlo han surgido numerosos métodos de recomendación de noticias que permiten filtrar y priorizar unas noticias sobre otras de manera personalizada. Este tipo de recomendación tiene algunas características que la hacen especial. La más destacable es la cantidad de información en forma de texto que podemos extraer de los artículos; es justo en ese punto en el que destacan los algoritmos basados en aprendizaje profundo, cada día más usados en el mundo del procesamiento de lenguaje natural.

En este trabajo se ha realizado una revisión sobre las técnicas utilizadas hoy en día en el campo de la recomendación de noticias analizando el rendimiento y eficiencia de cada una. De hecho, en este análisis se ha observado que, como era de esperar, las técnicas basadas en aprendizaje profundo son las que mejores resultados consiguen, a costa de la cantidad de recursos que requieren en su entrenamiento. Precisamente en el punto de la eficiencia es donde se abren hueco algunos algoritmos, que a pesar de no conseguir unos resultados tan buenos, requieren muchos menos recursos.

Por otro lado, se ha llevado a cabo el desarrollo de una aplicación web, en la cual se tuvo como objetivo analizar los mejores métodos para poner en producción modelos de aprendizaje automático, dando como resultado una interfaz gráfica que permite al usuario comparar el funcionamiento de los diferentes algoritmos sobre ejemplos reales.

PALABRAS CLAVE

Sistema de recomendación, recomendación de noticias, aprendizaje profundo, procesamiento del lenguaje natural, factorización de matrices

ABSTRACT

Linked to a growth in digital news media, there has been an inevitable overload of information towards the users. To overcome this, numerous recommender systems have emerged that provide filtered and prioritised news in a personalised manner. This kind of recommendation has some characteristics that make it special, the most highlighted one being the amount of rich information, such as the one that can be extracted from the (news) articles. In this particular case it is where deep learning algorithms shine, which is the reason why they are increasingly used in the natural language processing field.

In this thesis we have performed a review of some techniques that are being used nowadays in the news recommendation field, by analysing their performance and efficiency. In fact, in this analysis we have observed that, as expected, the techniques based on deep learning are the best performing ones, at the expense of the huge computational cost that is needed during training. It is specifically in efficiency where other algorithms can stand out, because in spite of not being as good in terms of performance, the computational resources needed are significantly reduced.

In parallel, we have developed a web application with the objective of analysing real-world implementations of machine learning models and how to put them in production. The result being a graphical interface that allows users to compare the performance of different algorithms on real data.

KEYWORDS

Recommender systems, news recommendation, deep learning, natural language processing, matrix factorization

ÍNDICE

1	Introducción	1
1.1	Motivación del proyecto	1
1.2	Objetivos	2
1.3	Estructura del trabajo	2
2	Estado del arte	3
2.1	Sistemas de recomendación	3
2.1.1	Recomendación basada en contenido	3
2.1.2	Recomendación basada en filtrado colaborativo	5
2.1.3	Métricas	6
2.2	Procesamiento del lenguaje natural	7
2.2.1	Representación numérica de texto	7
2.2.2	Word Embeddings	9
2.2.3	Transformers y atención	10
2.3	Sistemas de recomendación de noticias	11
2.3.1	MIND Dataset	12
2.3.2	Recomendación de noticias basada en técnicas clásicas	13
2.3.3	Recomendación de noticias basada en aprendizaje profundo	13
2.4	Aplicaciones web	17
2.4.1	Desplegar un modelo: API Rest	17
2.4.2	Interfaz Web: Flask	19
2.4.3	Infraestructura	19
3	Diseño e implementación	21
3.1	Diseño	21
3.1.1	Estructura	21
3.1.2	Ciclo de vida	22
3.2	Ánalisis de requisitos	24
3.2.1	Requisitos funcionales	24
3.2.2	Requisitos no funcionales	25
3.3	Implementación	26
4	Pruebas y resultados	33
4.1	Entorno de pruebas	33
4.2	Experimentos	34
4.2.1	Datos usados en los experimentos	34

4.2.2 Recomendación basada en técnicas clásicas	35
4.2.3 Recomendación basada en aprendizaje profundo	36
4.2.4 Resumen y análisis de coste	37
5 Conclusiones y trabajo futuro	39
5.1 Conclusiones	39
5.2 Trabajo Futuro	40
Bibliografía	43
Definiciones	45
Acrónimos	47
Apéndices	49
A Página web	51
B Arquitecturas de modelos	55
B.1 Word2Vec	55
B.2 Recomendación de noticias	55

LISTAS

Lista de ecuaciones

2.1	Representación documento en modelo de espacio vectorial	4
2.2	Modelo de espacio vectorial: ponderación binaria	4
2.3	Modelo de espacio vectorial: ponderación por frecuencia	4
2.4a	Ecuación <i>Term Frequency</i> (TF)	4
2.4b	Ecuación <i>Inverse Document Frequency</i> (IDF)	5
2.4c	Modelo de espacio vectorial: ponderación por TF-IDF	5
2.5	Ecuación a minimizar filtrado colaborativo	5
2.6	Predicción <i>rating</i> filtrado colaborativo	6
2.7a	Tasa de verdaderos positivos (TPR)	6
2.7b	Tasa de falsos positivos (FPR)	6
2.8	Ecuación <i>Reciprocal Rank</i> (RR)	6
2.9a	Ecuación <i>Discounted Cumulated Gain</i> (DCG)	6
2.9b	Ecuación <i>Normalized Discounted Cumulated Gain</i> (nDCG)	6
2.9c	Ecuación <i>Ideal Discounted Cumulated Gain</i> (IDCG)	6
2.10	Ecuación <i>Normalized Discounted Cumulated Gain at k</i> (nDCG@k)	6

Lista de figuras

2.1	Ejemplos básicos de codificación de texto en vectores	8
2.2	Ejemplo de noticias visitadas por un usuario	14
2.3	Dos noticias de ejemplo	15
2.4	Intereses a corto y largo plazo	16
3.1	Diagrama estructura general	22
3.2	Diagrama secuencia ejemplo de uso de la aplicación	23
3.3	Ejemplo flujo de la aplicación	26
3.4	Apartado de recomendación página web	32
4.1	Evolución del entrenamiento de algoritmos basados en aprendizaje profundo	37
A.1	Página principal de la aplicación web	51
A.2	Cabecera del apartado de recomendación de la aplicación web	52
A.3	Cuerpo del apartado de recomendación de la aplicación web	53

B.1	Arquitectura modelo Continuous Bag-of-Words (CBOW)	55
B.2	Arquitectura modelo Skip-gram	56
B.3	Arquitectura modelo NRMS	56
B.4	Arquitectura modelo NAML	57
B.5	Arquitectura modelo LSTUR	57

Lista de tablas

2.1	Tabla de ejemplos de operaciones con vectores de <i>embeddings</i>	9
4.1	Tabla características técnicas equipo de pruebas	33
4.2	Tabla resumen tamaño <i>dataset</i>	34
4.3	Tabla resumen de los resultados del algoritmo basado en frecuencias	35
4.4	Tabla resumen de los resultados resultados del algoritmo TF-IDF	36
4.5	Tabla de mejores resultados de algoritmos basados en aprendizaje profundo	37
4.6	Tabla comparativa tiempos de computación	38

INTRODUCCIÓN

El mundo digital en el que vivimos ha cambiado muchas cosas, entre ellas la forma en la que nos informamos. La prensa en papel no está preparada para actualizarse a cada minuto, mientras que la velocidad de los medios digitales ha logrado que cualquier persona pueda conocer en tiempo real las noticias de cualquier parte del mundo. Es por esto que la prensa digital ha logrado sustituir casi de manera definitiva a la prensa escrita.

1.1. Motivación del proyecto

Esta digitalización trae consigo numerosas ventajas, como las que hemos comentado, pero también algunos problemas. El principal inconveniente de los medios digitales es precisamente la cantidad de noticias generadas por día, inabarcables por cualquier ser humano. Es ahí donde los sistemas de recomendación cobran gran importancia, aliviando esa sobrecarga de información y mejorando la experiencia de visitar un periódico en línea.

Respecto a estos, los sistemas de recomendación de noticias, tienen algunas características que los hace especiales, las cuales explotaremos a lo largo de este trabajo. La más destacable es que gran parte de la información que podemos extraer de una noticia se encuentra en forma de texto, por lo que deberemos utilizar técnicas de procesamiento de lenguaje natural para poder aprovecharla.

Además, observando la bibliografía podemos ver cómo los últimos avances en el campo de la recomendación de noticias, como muchos otros, se encuentran eclipsados por el “boom” de los algoritmos basados en técnicas de aprendizaje profundo. Es por ello que en este trabajo, además de cerciorarnos de las altas capacidades de este tipo de algoritmos, trataremos de compararlos con otros que, si bien serán más sencillos, no por ello dejarán de ser útiles (confirmando bajo qué condiciones sería mejor usar unos u otros).

1.2. Objetivos

El principal objetivo de este trabajo ha sido comprender y analizar cuáles son los últimos avances en el mundo de los sistemas de recomendación de noticias. Pero este trabajo no se ha tratado de una carrera de fondo para lograr ese objetivo, sino que hemos decidido disfrutar del camino. En este recorrido hemos aprovechado por un lado para profundizar sobre algunos conceptos de procesamiento del lenguaje natural y redes neuronales profundas en los que no se había hecho mucho hincapié hasta el momento, y por otro lado para repasar numerosos conceptos que sí hemos visto en el grado. Por ejemplo, a la hora de realizar una página web, en la que hemos aplicado conceptos del modelo cliente-servidor, sistemas informáticos distribuidos y evidentemente desarrollo web, entre otras muchas cosas, además hemos aprendido casi desde cero cuál es la forma de desplegar un modelo de aprendizaje automático en un entorno en producción, algo muy necesario actualmente debido al gran interés que han despertado estos modelos en el ámbito empresarial.

1.3. Estructura del trabajo

Capítulo 1. Introducción. Se describe el problema sobre el que gira todo el proyecto, la motivación del trabajo y la estructura del mismo.

Capítulo 2. Estado del arte. Se recopila la información que ha servido como base teórica para este trabajo. Analizando cuáles son los últimos avances en sistemas de recomendación, procesamiento del lenguaje natural, sistemas de recomendación orientados a noticias y aplicaciones web.

Capítulo 3. Diseño e implementación. Se detallan las decisiones de diseño tomadas en torno a todo el proyecto, explicando cuál es la estructura del mismo, el ciclo de vida seguido, y analizando los requisitos funcionales y no funcionales del proyecto. Por otro lado, también se exponen los detalles de implementación de todo el código del trabajo.

Capítulo 4. Pruebas y resultados. Se detallan los experimentos realizados en relación con los algoritmos utilizados, así como los resultados obtenidos en cada uno de ellos.

Capítulo 5. Conclusiones y trabajo futuro. Se establecen cuáles han sido las conclusiones de este trabajo, así como un pequeño repaso de las posibles mejoras sobre las que se puede continuar investigando en un futuro.

ESTADO DEL ARTE

En este capítulo se dará una pequeña introducción a los diferentes puntos clave que sustentan este trabajo, desde los sistemas de recomendación en general hasta la aplicación web desarrollada, pasando por el procesamiento del lenguaje natural y algoritmos de aprendizaje profundo utilizados.

2.1. Sistemas de recomendación

Los sistemas de recomendación son herramientas cuya finalidad es mostrar al usuario ítems que sean de su interés. Este tipo de sistemas están adquiriendo una gran popularidad debido a que contamos con mucha más información de la que somos capaces de consumir, por lo que cualquier herramienta que nos facilite la tarea de selección de esa información nos será de mucha ayuda.

Este tipo de sistemas se encuentran ya presentes en nuestro día a día; cada vez que hacemos una compra en línea y se nos muestran productos recomendados, cada vez que accedemos a ver una película en nuestra plataforma de *streaming* favorita, etc. y, llevándolo al tema que nos incumbe, cada vez que accedemos a consultar las noticias en línea y se nos muestran primero las que más nos pueden interesar, o incluso nos saltan notificaciones en el móvil de manera personalizada.

Para llevar a cabo esta recomendación podemos seguir diferentes técnicas, las más importantes se dividen en dos: recomendación basada en contenido y recomendación basada en filtrado colaborativo.

2.1.1. Recomendación basada en contenido

Los sistemas de recomendación basados en contenido se basan en recomendar ítems similares a los que le gustaron al usuario en un pasado [1]. En el problema que nos atañe, la recomendación basada en contenido más básica consistirá en comparar las noticias candidatas con las noticias que ya se han visitado, ordenando las primeras en base a su similitud con las segundas.

Modelo de espacio vectorial

Se trata de un modelo algebraico muy común en el ámbito de la recuperación de información. En este modelo cada documento se representa con un vector de n dimensiones, donde cada una de sus dimensiones corresponde al peso en el documento de un término concreto. Utilizando este modelo podremos convertir de forma sencilla cada una de nuestras noticias a un vector numérico, con el que será mucho más sencillo estudiar similitudes.

La representación resultante de un documento utilizando este modelo está disponible en la ecuación 2.1. Para determinar el peso de cada término podemos seguir diferentes técnicas:

$$d_j = (w_{1,j}, w_{2,j}, \dots, w_{t,j}) \quad (2.1)$$

Ponderación binaria

Es el más simple de todos, cada peso vale 1 si el término se encuentra en el documento y 0 en caso contrario (ecuación 2.2).

$$w_{bin}(i, j) = \begin{cases} 1, & \text{si } term_i \text{ en } d_j \\ 0, & \text{en otro caso} \end{cases} \quad (2.2)$$

Ponderación por frecuencia

En este caso cada peso equivale a la frecuencia del término en el documento correspondiente, es decir, un contador (ecuación 2.3).

$$w_{frecuencia}(i, j) = freq(term_i, d_j) \quad (2.3)$$

Ponderación por TF-IDF

Frecuencia de término – frecuencia inversa de documento (en inglés *Term frequency - Inverse document frequency*) se trata de una técnica para elegir el peso del término muy similar a la anterior (frecuencia), pero normalizada, quitando importancia a las palabras que aparecen en muchos documentos, ya que estas en principio deberían aportar menos información [2].

$$tf(t, d) = \begin{cases} 1 + \log_2 freq(t, d), & \text{si } freq(t, d) > 0 \\ 0, & \text{en otro caso} \end{cases} \quad (2.4a)$$

$$idf(t) = \log \frac{|\mathcal{D}|}{|\mathcal{D}_t|} \quad (2.4b)$$

$$w_{tf-idf}(i, j) = tf(t_i, d_j) \times idf(t_i) \quad (2.4c)$$

Donde $|\mathcal{D}|$ es el número de documentos totales y $|\mathcal{D}_t|$ el número de documentos en los que aparece ese término t .

2.1.2. Recomendación basada en filtrado colaborativo

Esta técnica analiza los comportamientos pasados de todos los usuarios con el objetivo de establecer conexiones entre ellos [3]. Para entender mejor la motivación del mismo pensemos por ejemplo en un conjunto de usuarios que consuman muchas noticias sobre ciencia, si de repente un subconjunto de este comienza a consumir ciertas noticias sobre política probablemente al resto también le interesen esas noticias, a pesar de no haber consumido política nunca antes.

El tipo de filtrado colaborativo más común es el basado en vecinos próximos [4], el cual a su vez se divide en dos tipos. En primer lugar el centrado en usuario, en el cual se estima una valoración desconocida de un usuario concreto a un ítem basándose en las diferentes valoraciones que usuarios similares al objetivo le han dado a ese ítem. El otro método es el basado en ítem, en el cual las valoraciones desconocidas se estiman utilizando otras valoraciones hechas por un mismo usuario en ítems similares al objetivo. Además del basado en vecinos existen muchos otros métodos, por ejemplo basados en clustering, redes neuronales y los que explicaremos en esta ocasión: los basados en factorización de matrices [5]. Nos centramos en estos últimos debido a que los algoritmos de aprendizaje profundo utilizados se basan precisamente en una aproximación similar.

Factorización de matrices

El principal objetivo de toda recomendación es conseguir predecir las preferencias de cada usuario respecto a cada posible ítem. La forma más natural de almacenar esta información podría ser una matriz R , en la que cada fila representará a un usuario y cada columna un ítem. En un principio esta matriz sería muy dispersa, es decir, la mayor parte de sus elementos estarían vacíos, y el objetivo de este algoritmo es conseguir llenar esa matriz.

Esta matriz puede descomponerse con facilidad en dos submatrices, una para los usuarios (U) y otra para los ítems (I). Para solucionar el problema lo necesario sería minimizar la expresión disponible en la ecuación 2.5:

$$\sum_{(u,i) \in \text{training}} \left[\left(r(u, i) - \sum_f u_f i_f \right)^2 + \lambda \left(\sum_f u_f^2 + \sum_f i_f^2 \right) \right] \quad (2.5)$$

Una vez hecho esto, podremos predecir el *rating* haciendo el producto escalar de ambos vectores (ecuación 2.6)

$$\hat{r}(u, i) = \sum_f u_f i_f \quad (2.6)$$

2.1.3. Métricas

Para poder evaluar el rendimiento cada uno de los algoritmos de recomendación que vamos a presentar haremos uso de algunas métricas. Las elegidas son las siguientes, muy comunes en el campo de la recuperación de información [6]:

- **AUC:** Se define como el área bajo la curva ROC. La curva ROC es una representación gráfica de la tasa de verdaderos positivos (TPR , ecuación 2.7a) y la tasa de falsos positivos (FPR , ecuación 2.7b) frente a diferentes puntos de operación, cada uno de estos puntos vendrá definido por un umbral de clasificación.

$$TPR = \frac{TP}{TP + FN} \quad (2.7a)$$

$$FPR = \frac{FP}{TN + FP} \quad (2.7b)$$

Donde TP son los verdaderos positivos, FN los falsos negativos, FP falsos positivos y TN los verdaderos negativos.

- **MRR:** *Reciprocal Rank (RR)* se define como el inverso multiplicativo de la posición del ranking que ocupa el primer resultado correcto. También conviene explicar que *MRR* (del inglés *Mean Reciprocal Rank*) es simplemente el promedio de los *RR* de diferentes predicciones de comportamiento de un usuario:

$$RR = \frac{1}{rank_i} \quad (2.8)$$

- **nDCG (@k):** Para entender la métrica *normalized Discounted Cumulated Gain (nDCG)* antes debemos entender *Discounted Cumulated Gain (DCG)*, descrito en la ecuación 2.9a en la cual $g(d_k)$ es el grado de relevancia de un ítem devuelto en la posición k . Una vez comprendido esto describimos *nDCG* como vemos en la ecuación 2.9b, solamente nos quedaría saber que *IDCG* (*ideal Discounted Cumulated Gain*), como su nombre indica es el *DCG* que tendría el ranking ideal (es decir, de todos los posibles ranking ($R \in \sigma(\mathcal{D})$)) nos quedamos con el que tenga un *DCG* máximo), descrito en la ecuación 2.9c.

$$DCG = \sum_k \frac{g(d_k)}{\log_2(k + 1)} \quad (2.9a)$$

$$nDCG = \frac{DCG}{IDCG} \quad (2.9b)$$

$$IDCG = \max_{R \in \sigma(\mathcal{D})} DCG(R, q) \quad (2.9c)$$

Respecto a los $@k$, significa simplemente “at k ”, es decir, calcular la métrica teniendo en cuenta solamente los k primeros elementos del ranking (ecuación 2.10).

$$nDCG@k = \frac{DCG@k}{IDCG@k} \quad (2.10)$$

2.2. Procesamiento del lenguaje natural

El procesamiento del lenguaje natural (NLP por sus siglas en inglés) es un área de investigación lingüística, informática y de inteligencia artificial que explora cómo los ordenadores pueden utilizarse para entender, generar y analizar textos escritos en lenguaje natural. Conseguir esto es una de las metas de la inteligencia artificial actual y, sin duda, uno de los requisitos para lograr una inteligencia artificial general [7] y en este capítulo realizaremos un análisis del estado del arte de esta tecnología.

En un primer acercamiento podemos pensar en codificar en las reglas formales todas las normas de comunicación que componen un lenguaje, pero no tardaremos mucho en darnos cuenta de que eso no es una buena idea: tendremos que tener en cuenta no solo el significado exacto de cada una de las palabras, sino que también tendremos que codificar la sintaxis (o estructura de las oraciones), el sentido de la palabra en un contexto concreto, toda la gramática, etc. Muchos puntos a tener en cuenta si a eso le sumamos que no serviría con hacerlo una vez, el lenguaje se encuentra constantemente en evolución, por lo que tendríamos que actualizar este conjunto de reglas continuamente. Siguiendo esta metodología el procesamiento de lenguaje natural se convierte en una tarea prácticamente inabordable.

Este tipo de problema nos recuerda a algunos otros que ya están siendo abordados ahora mismo, como por ejemplo reconocimiento de objetos en imágenes, en el que no tratamos de codificar unas reglas que describan qué es un perro y qué es un gato para que una máquina sea capaz de distinguirlos, sino que dejamos que un algoritmo de aprendizaje automático aprenda a resolver el problema y detecte este tipo de reglas por sí solo. Esto mismo es lo que se está haciendo actualmente en el campo del NLP.

2.2.1. Representación numérica de texto

En este apartado nos centraremos en el uso de redes neuronales, ya que es el algoritmo más utilizado para este tipo de problemas. Al intentar trabajar con texto y algoritmos de aprendizaje automático, en este caso redes neuronales, el primer problema que nos vamos a encontrar es que este tipo de algoritmos trabajan únicamente con números, por lo que tendremos que encontrar la manera de convertir esas palabras a vectores.

Para llevar esto a cabo podríamos utilizar algunos de los métodos que hemos estudiado en la sección 2.1, basados en un modelo de espacio vectorial, pero de esa forma estaríamos perdiendo el contexto de cada una de las palabras, y en esta ocasión como veremos más adelante, es importante mantenerlo. Además tendríamos que manejar vectores con n dimensiones, donde n sería el tamaño del vocabulario, y eso tiene un coste computacional muy alto.

Otra opción muy simple podría ser asignar de manera arbitraria un identificador único a cada palabra, de esta forma no perderíamos el contexto pero tendríamos algunos problemas nuevos, el primero es que no todas las representaciones ocupan lo mismo, y eso puede repercutir de manera negativa a la hora de utilizar redes neuronales.

El segundo problema es algo más complicado de explicar: por una parte, si utilizamos los identificadores como números enteros, uno por palabra, le estaremos dando a entender a nuestra red neuronal que inicialmente algunas palabras tienen una importancia muy superior a otra, por ejemplo, en la figura 2.1 podemos ver cómo la palabra “vaca” tendría 8 veces más importancia que “al”. Tras esto podemos pensar en utilizar esos mismos identificadores, pero esta vez codificados utilizando *One-Hot*¹, de esta manera no tendremos el problema que acabamos de mencionar, pero, suponiendo un espacio de N dimensiones, donde N es el tamaño del vocabulario, estaremos asumiendo que todas las palabras se encuentran a la misma distancia, es decir, de nuevo en la figura 2.1 “vaca” y “trabajo” estarían igual de relacionadas que “trabajo” y “horario”. Es fácil darse cuenta de que no es del todo correcto, por no hablar del incremento de coste computacional que tendría almacenar cada palabra utilizando un vector de N dimensiones, dando lugar finalmente a una matriz de $N * M$ para codificar una sola oración, donde N es el tamaño de vocabulario y M el número de palabras de la oración. Además, pronto nos daremos cuenta de que de esta manera las palabras están perdiendo completamente su contexto, el cual es muy importante a la hora de comprender un texto.

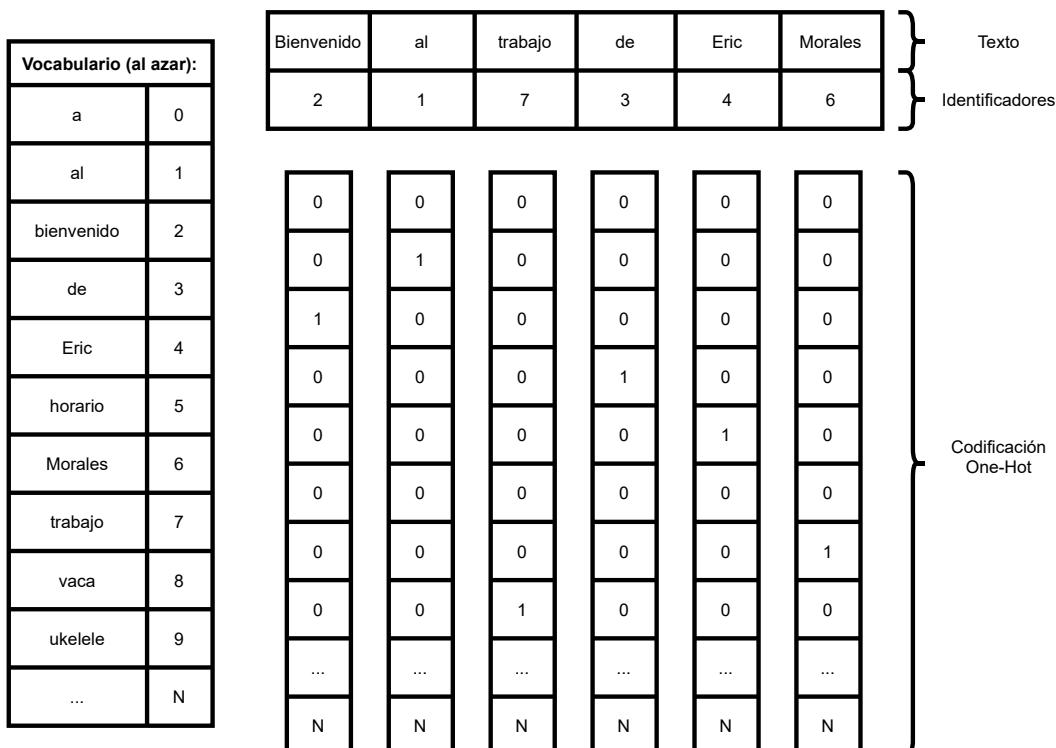


Figura 2.1: Ejemplos básicos de codificación de texto en vectores.

¹ Codificación en la cual cada palabra se representa mediante un vector binario con tantas posiciones como palabras tenga el diccionario, este vector tendrá 1 en la posición correspondiente a la palabra que representa y todas las demás posiciones serán 0.

2.2.2. Word Embeddings

Una posible solución a los problemas planteados en el apartado anterior (equidistancia, contexto, escalabilidad, ...) podría ser utilizar *embeddings*. Un *embedding* es simplemente una representación vectorial (de tamaño fijo) de prácticamente cualquier cosa, un usuario, un nodo de un grafo, un ítem, una frase (algo similar a los generados en la sección 2.1.1 utilizando *TF-IDF* y demás) o palabras, en estos últimos nos centraremos en esta sección.

El principal beneficio de esta representación es que nos permite almacenar en un solo vector, no solo la palabra en sí (para distinguirla del resto), sino también su información semántica, sintáctica y contextual. Por ejemplo nos servirá para diferenciar palabras polisémicas que dependen de su contexto, o nos permitirá ver que las conjugaciones de un mismo verbo tienen representaciones muy similares. Además, estos vectores tendrán un tamaño mucho más pequeño que los mencionados en apartados anteriores: nunca utilizaremos un vector del tamaño del vocabulario, sino que reduciremos la dimensionalidad de este a un tamaño fijo para todas las palabras.

Por otro lado, podemos observar algunos comportamientos muy interesantes, los cuales pueden ser aprovechados por algunos algoritmos, uno de ellos es la posibilidad de realizar operaciones matemáticas directamente utilizando los vectores que representan a cada palabra. Por ejemplo las ecuaciones de la tabla 2.1, en las dos primeras podemos ver cómo las palabras comprimen el significado de las palabras, hasta tal punto que se llegan a codificar incluso los sesgos, como podemos ver en la tercera ecuación [8].

$\overrightarrow{\text{software}} - \overrightarrow{\text{program}} + \overrightarrow{\text{device}} \approx \overrightarrow{\text{hardware}}$
$\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}} \approx \overrightarrow{\text{queen}}$
$\overrightarrow{\text{man}} - \overrightarrow{\text{woman}} \approx \overrightarrow{\text{computer programmer}} - \overrightarrow{\text{homemaker}}$

Tabla 2.1: Ejemplos de operaciones con vectores de *embeddings*.

Word2Vec

Uno de los métodos más conocidos para generar *word embeddings* es *word2vec*. Cabe destacar que no se trata de un algoritmo ni de un método concreto, sino de un conjunto de métodos con el mismo objetivo pero diferentes aproximaciones, en concreto, en [9] y [10] se describen los siguientes:

- **Continuous Bag-of-Words (CBOW)**: su objetivo es predecir una palabra dado su contexto.
- **Skip-gram**: su objetivo es predecir las palabras que se encuentran alrededor (contexto) de una dada.

Es muy sencillo ver el paralelismo de estos dos modelos, uno hace exactamente lo opuesto al otro. De hecho en [11] y en el anexo B.1 se puede ver la arquitectura de la red neuronal de cada modelo y son simétricas, es decir, una el “espejo” de la otra.

2.2.3. Transformers y atención

En 2017 Google lanza el conocido artículo *Attention Is All You Need* [12] el cual prácticamente revoluciona el mundo del procesamiento del lenguaje natural presentando una arquitectura *encoder-decoder* que utiliza atención bautizada como *Transformer*.

En el anterior párrafo hemos nombrado numerosas cosas nuevas, en primer lugar, arquitectura *encoder-decoder*, que se trata de una estructura que trabaja en dos fases, como su propio nombre indica: una se encarga de codificar la información de entrada (*embeddings*) a un formato interno (matrices multidimensionales) y la otra fase que se encarga de decodificar la información obteniendo unos *embeddings* de salida.

Por otro lado hemos hablado de atención, un concepto muy importante que utilizaremos más adelante, más concretamente *Self-Attention*. No vamos a entrar en la implementación interna del mismo, pero podríamos decir que se presenta como el sustituto a las redes neuronales recurrentes (LSTM, GRU, ...) utilizadas para entender el contexto de una palabra. Por ejemplo, en la oración “*El niño dijo que eso a él no le gustaba*”, gracias a *Self-Attention* podremos saber que “él” se refiere al niño. También nos ayudará a distinguir polisemias, por ejemplo distinguir “*banco de peces*” de “*banco de madera*”. Para más información sobre el funcionamiento de este método, consultar [12].

Sustituir atención por Transformadas de Fourier

En [13], un artículo de Google publicado hace escasas semanas (respecto a la redacción de este documento), se muestra el descubrimiento de que las Transformadas de Fourier, a pesar de no tener ningún parámetro, consiguen un rendimiento similar al conseguido por redes neuronales, además de escalar de manera muy eficiente, especialmente en *GPUs* (debido a la Transformada de Fourier Rápida). A esta red la denominaron *FNet*.

Reemplazando la capa de atención por transformaciones lineales es posible reducir la complejidad y el coste computacional de la arquitectura *Transformer*. De hecho la estructura *FNet* ofrece un excelente compromiso entre velocidad, memoria y rendimiento, consiguiendo el 92% del rendimiento obtenido por *BERT* en problemas de clasificación, pero entrenando siete veces más rápido. Aún así, al ser un descubrimiento tan nuevo, está por ver su utilidad práctica en otros escenarios (como analizaremos más adelante en este documento). Además también hay que tener en cuenta que hay pocas implementaciones públicas, lo cual limita y complica bastante su uso.

Implementaciones de Transformers

Tras la presentación de *Attention Is All You Need* [12], muchos investigadores empezaron a incorporarlo en sus proyectos, dando lugar a numerosos modelos de lenguajes muy utilizados en la actualidad. En este apartado nos vamos a centrar en *BERT*, de Google; y en *GPT*, de OpenAI. Ambos modelos cuentan con un entrenamiento en dos fases, un primer pre-entrenamiento no supervisado, en el cual

tratan de “absorber” toda la información posible de los textos que pasan por ellos y una segunda parte supervisada, denominada *fine-tuning* en la cual se especializa el modelo en una tarea concreta.

En cuanto a *Bidirectional Encoder Representations from Transformers (BERT)* como su propio nombre indica, su principal peculiaridad es su naturaleza bidireccional, de hecho, en el artículo citan textualmente: “*Our major contribution is further generalizing these findings to deep bidirectional architectures*” [14]. El pre-entrenamiento de este modelo se produce a través de dos tareas, la primera denominada *Masked LM*, que consiste en presentar un texto al cual le falta una palabra y la tarea del modelo es ser capaz de predecir cuál debería ser esa palabra, basándose en el contexto de la misma, tanto por la izquierda como por la derecha (de ahí el carácter bidireccional del modelo). La otra tarea utilizada es *Next Sentence Prediction (NSP)*, es decir, dado un texto es capaz de predecir cuáles serán las siguientes palabras del mismo. *BERT* fue vanguardia en numerosos aspectos del procesamiento del lenguaje natural, de hecho actualmente Google lo utiliza en su motor de búsqueda para entender mejor las consultas de los usuarios ².

Por otro lado *GPT*, con su versión *GPT-3* se ha convertido en lo que algunos catalogan como uno de los descubrimientos del siglo, su arquitectura es similar a la de *transformer*, pero suprimiendo la parte del *encoder*, trabaja directamente con *embeddings*. Su pre-entrenamiento es muy similar a la segunda tarea de *BERT*, predecir las palabras que deberían seguir a un texto dado.

Retomando *GPT-3* [15], este ha sido entrenado con la wikipedia completa, miles de artículos científicos, todos los libros públicos que se encuentran en internet y miles de noticias. Cuenta con 175.000 millones de parámetros y fue entrenado en 48 *GPUs* de 16 GB de vRAM cada una. Su importancia radica en que ha conseguido dar un paso más allá en el mundo de los modelos de lenguaje, no solo entiende texto, también sabe resolver ecuaciones matemáticas, incluso es capaz de programar páginas web. Respecto a este último punto, un ejemplo muy recurrente, es que con una simple descripción del tipo: “una página web con un formulario de inicio de sesión utilizando botones de color rojo” el modelo será capaz de programar el código HTML necesario. Además, sin duda, es una herramienta muy potente en su campo principal, siendo capaz de redactar cualquier texto que se le solicite, sea de índole periodístico, novelístico o incluso completando el diálogo de una novela teatral.

2.3. Sistemas de recomendación de noticias

Cada día, una gran cantidad de noticias nuevas son publicadas, dificultando mucho a los usuarios la tarea de encontrar noticias interesantes de manera rápida [16]. Un buen sistema de recomendación de noticias puede ayudar a los usuarios a aliviar esta sobrecarga de información y mejorar la experiencia de consultar las noticias [17].

Por otro lado, la recomendación de noticias tiene algunos retos que la caracterizan. En primer lugar,

² Simonite, Tom 2019, *Google Search Now Reads at a Higher Level*, Wired, accessed 2021-01-19. <https://www.wired.com/story/google-search-advancing-grade-reading/>

las páginas web de noticias se actualizan muy rápido. Se publican noticias nuevas continuamente y las noticias existentes pierden importancia de manera muy rápida [18]. Debido a esto, nuestro sistema sufrirá continuamente del problema de arranque en frío (en inglés *cold-start*), ya que no será capaz de extraer recomendaciones de todas las noticias nuevas, al no tener información suficiente sobre ellas. En segundo lugar, las noticias contienen información en forma de texto y como ya hemos visto en el apartado 2.2, lo más correcto no será asignar simplemente un identificador a cada texto, sino que deberemos entender el contenido de estos los mismos [19].

Además, respecto a este punto, tendremos que tener en cuenta la información que nos aporta cada atributo, por ejemplo el título nos dará mucha información de manera muy general, mientras que el cuerpo nos permitirá extraer el detalle de la noticia. Por último, debemos tener en cuenta que no hay una valoración concreta para cada noticia, sino que tendremos que inferir los gustos del usuario en base a las que ha dado o no clic, sin saber exactamente cuáles le han gustado más o menos [20], a diferencia de lo que suele ocurrir en otros dominios como la recomendación de películas o productos.

Tras esta breve introducción, necesaria para ser capaz de comprender algunos puntos del problema a resolver, pasamos a explicar cuáles son algunos de los métodos de recomendación de noticias que se están utilizando en la actualidad. Empezaremos haciendo una pequeña presentación al *dataset* elegido para este trabajo, uno de los más completos que existen actualmente orientados a este ámbito y posteriormente nos centraremos en los diferentes algoritmos que hemos analizado y finalmente han sido incluidos en la aplicación.

2.3.1. MIND Dataset

Antes de empezar a analizar los diferentes métodos de recomendación de noticias debemos dar más detalles sobre el problema a resolver. Hace unos meses Microsoft publicó uno de los *dataset* orientados a la investigación en recomendación de noticias más grandes hasta la fecha, descrito en [21]. Como se indica en ese artículo *Microsoft News Dataset (MIND)* es un *dataset* que recoge por un lado alrededor de 160.000 noticias en inglés y por otro lado más de 15 millones de impresiones (procedentes de 1 millón de usuarios) relacionadas con esas noticias. La misión de este *dataset* es fomentar y facilitar la investigación alrededor de los sistemas de recomendación de noticias, de hecho fue lanzado al público general en forma de reto científico.

Cada noticia posee título, resumen, cuerpo, categoría y algunas entidades. Cada impresión o comportamiento, que se encuentra asignada a un identificador de usuario, posee por un lado una lista de noticias en las que ese usuario ha dado clic en el pasado y por otro una lista de noticias candidatas, con información sobre cuáles han sido visitadas y cuáles no.

La labor de aquellos investigadores que quieran involucrarse en este reto es ser capaz de predecir en cuál de las noticias candidatas el usuario va a acabar haciendo clic a través de un ranking, donde la puntuación de cada noticia es la probabilidad de ser elegida. El *dataset* ha sido construido utilizando los *logs* de Microsoft News durante 6 semanas, desde el 12 de octubre hasta el 22 de noviembre

de 2019. Esto nos da una pequeña idea de la sobrecarga de información de la que hablábamos al principio de esta sección 2.2, 160.000 noticias en 6 semanas nos indica que, de media, cada día se están publicando unas 4.000 noticias, sin duda un número inabarcable para cualquier ser humano.

2.3.2. Recomendación de noticias basada en técnicas clásicas

Muchos métodos convencionales de recomendación de noticias se basan en utilizar modelos básicos, sin utilizar redes neuronales ni algoritmos complejos para formar así las representaciones de usuarios y noticias. Por ejemplo, [22] propone utilizar un modelo bayesiano [23] para predecir las categorías y los intereses y con ellos representar las diferentes noticias y utilizar la distribución del historial de clics para representar al usuario. Por otro lado [24] utilizó un modelo de Asignación Latente de Dirichlet [25] (en inglés *LDA*, *Latent Dirichlet Allocation*) para generar una representación de noticias basada en una distribución de categorías. En general, estos algoritmos representan una sesión utilizando la distribución de categorías de las noticias visitadas en esa sesión y las representaciones de los usuarios se crean a partir de las representaciones de sus sesiones, ponderadas por el tiempo (separando intereses a largo y corto plazo).

Los dos principales problemas de estas “técnicas clásicas” son, en primer lugar, que se trata de métodos demasiado artesanales, que requieren un conocimiento muy extenso del ámbito sobre el cual lo estamos aplicando para poder conseguir que funcionen correctamente. Por otro lado en este tipo de algoritmos el contexto y el orden de las palabras no se tienen en cuenta, y es algo muy importante para entender el significado semántico de las noticias y con ello se podría mejorar bastante la construcción de representaciones de noticias y usuarios.

2.3.3. Recomendación de noticias basada en aprendizaje profundo

Al igual que en los métodos comentados en el apartado anterior, la principal dificultad de los métodos de recomendación basados en aprendizaje profundo es encontrar una manera de representar al usuario y a las noticias. Por ejemplo, [16] propone generar la representación de las noticias a través del cuerpo de las mismas utilizando *auto-encoders*³ y generar las de los usuarios a partir de las representaciones de las noticias que ha visitado, utilizando redes neuronales recurrentes (en concreto *GRU*, del inglés *Gated Recurrent Unit*). Los dos principales problemas de esta aproximación es que *GRU* consume muchísimo tiempo, no tiene en cuenta el contexto de las palabras y, además, si contamos con un historial muy largo, no será capaz de capturar todo por completo.

Por otro lado [26] propone generar la representación de las noticias utilizando redes neuronales convolucionales (del inglés *CNN*, *Convolutional Neural Networks* [27]) basadas en conocimiento (*knowledge-aware*) utilizando los títulos de las noticias, y aprender la representación de los usuarios

³A partir de este punto se utilizarán algunos tecnicismos que no tienen una traducción muy clara al español, como puede ser *self-attention*, *encoder*, ... por lo que se utilizará directamente su versión en inglés, sin traducir.

basándose en la similitud entre las noticias visitadas y las noticias candidatas. Esta idea también tiene algunos problemas, en primer lugar las redes convolucionales no tienen en cuenta la relación entre las noticias visitadas, ni es capaz de capturar el contexto de las palabras si estas se encuentran muy separadas, además necesita almacenar el historial completo de cada usuario, lo que puede generar una latencia muy alta.

A continuación se detallan algunos de los algoritmos basados en estos métodos utilizados a lo largo del trabajo, la implementación de todos ellos ha sido extraída de la librería *recommenders*, disponible en el repositorio *open source* de Microsoft [21], [28].

NRMS

A la hora de crear este algoritmo los autores observaron algunos detalles que convendría tener en cuenta para llevar a cabo la tarea propuesta [29]. En primer lugar, observaron que las relaciones entre las palabras del título de la noticia son muy importantes para entender las noticias. Por ejemplo en la figura 2.2, la palabra “Rockets” tiene una relación muy fuerte con “Bulls”. Además, una sola palabra puede interactuar con varias palabras, por ejemplo “Rockets” tiene interacciones semánticas tanto con “Bulls” como con “trade”. En segundo lugar, las diferentes noticias que han sido visitadas por un usuario tienen también una relación. Por ejemplo, en la figura 2.2 la segunda noticia está relacionada con la primera y la tercera.

Por último, cada una de las palabras pueden tener una importancia diferente, que variará en función de la información que nos aporte de cara a representar a la noticia, por ejemplo, en la figura 2.2 la palabra “NBA” aporta más información que “2018”. Además las diferentes noticias visitadas por un mismo usuario también deberían tener una importancia diferente al construir la representación de ese usuario, por ejemplo, de nuevo en la figura 2.2, las primeras tres noticias aportan más información que la última.

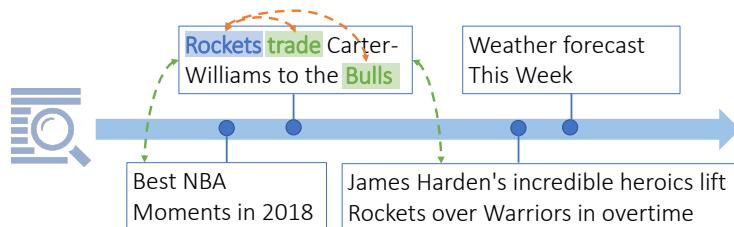


Figura 2.2: Ejemplo de noticias visitadas por un usuario. Las líneas verdes y naranjas representan las interacciones entre palabras y noticias, respectivamente [29].

En el artículo citado anteriormente proponen un “*Neural News Recommendation with Multi-Head Self-Attention (NRMS)*” El núcleo de esta aproximación es, por un lado, un *encoder* de noticias y por otro uno de usuarios. En el *encoder* de noticias se aprenden las representaciones de las noticias utilizando *multi-head self-attention* para modelar las interacciones entre las palabras. En el *encoder* de usuarios se generan las representaciones a partir de las noticias visitadas, utilizando *multi-head self-*

attention para capturar sus relaciones. Además se aplica *attention* a ambos *encoders* para seleccionar las palabras más importantes.

NAML

La motivación de este algoritmo parte de las siguientes observaciones [30]. En primer lugar, una noticia normalmente contiene diferentes atributos, como el título, el cuerpo y la categoría, los cuales pueden ser útiles para representar a una noticia. De hecho el artículo critica que otros autores solo utilicen algunos de ellos para aprender las representaciones de noticias y usuarios, diciendo que esto podría limitar el rendimiento del sistema de recomendación. Por ejemplo en la figura 2.3 el título de la primera noticia indica que trata sobre el contrato de un jugador de béisbol y un equipo, y el cuerpo aporta más información sobre ese contrato. Además, la categoría de esta noticia también aporta información, si alguien elige esta noticia y muchas otras sobre deportes, podremos inferir que ese usuario está muy interesado en deportes.

En segundo lugar, cada uno de los atributos es muy diferente del resto. Los títulos son cortos y concisos, mientras que los cuerpos son muy largos y detallados; por otro lado las categorías suelen ser etiquetas con pocas palabras. Teniendo esto en cuenta, podemos ver claramente que cada atributo debe procesarse de diferente forma.

En tercer lugar, en algunas noticias estos atributos pueden no aportarnos información, por ejemplo en la figura 2.3 el título de la primera noticia es muy preciso e importante para representar esa noticia, mientras que el de la segunda deja bastante que desear. Por otro lado, las mismas palabras en diferentes noticias pueden tener una importancia distinta. Por ejemplo, en la figura 2.3 “Outfielder” es más importante que “familiar” para representar la primera noticia. Además, puede que no todas las noticias aporten la misma información de cara a representar al usuario: no es lo mismo que lea noticias sobre la “Super Bowl”, que es muy popular, a que lo haga sobre la “Olimpiada Matemática Internacional”, que es mucho menos conocido, la segunda nos dará mucha más información sobre sus intereses que la primera. Esta idea se asemeja mucho al concepto de IDF descrito en la sección 2.1.1, en el cual quitábamos importancia a las palabras que aparecían en muchos documentos, aquí le quitamos importancia a las noticias que ha visto mucha gente.

Category	Sports	Entertainment
Title	Astros improve outfield , agree to 2-year deal with Brantley	The best games of 2018
Body	Outfielder Michael Brantley agreed to a two-year, \$32 million contract with Houston , sources familiar with the deal told Yahoo Sports, bringing his steady left-handed bat to the top of an Astros ...	The Best Games of 2018 Superheroes , super-dads , and Super Mario parties brought the joy in 2018 to millions of players who ate up the year's impressive achievements in gaming ...

Figura 2.3: Dos noticias de ejemplo. El subrayado rojo señala las palabras que tienen más importancia [30].

En el artículo citado anteriormente proponen un “*Neural News Recommendation with Attentive*

Multi-View Learning (NAML) ” para aprender las representaciones de las noticias y los usuarios explotando diferentes tipos de información. Este contiene dos componentes, igual que **NRMS** , por un lado un *encoder* de noticias y por otro uno de usuarios. En el *encoder* de noticias proponen un “*attentive multi-view learning framework*” para aprender representaciones de las noticias a partir de sus títulos, cuerpos y categorías, incorporando cada uno como una nueva “*view*” de la noticia (podemos imaginarnos una “*view*” como una dimensión). Además, como algunas palabras y *views* pueden no aportarnos información en la representación de noticias, aplican redes de atención para seleccionar las palabras y *views* más importantes, para así construir correctamente la representación final de la noticia. En el *encoder* de usuarios, se aprende la representación del usuario a partir de la representación de las noticias que ha seleccionado. Como cada noticia puede tener también una importancia diferente a la hora de representar a un usuario, aplican mecanismos de atención para seleccionar las noticias más importantes y aprender con ellas la representación final del usuario.

LSTUR

El trabajo del artículo [31] de este último algoritmo se ha visto motivado por la observación de que los intereses de los usuarios son muy diversos. Algunos intereses se mantienen a lo largo del tiempo, incluso sobre los mismos usuarios [24]. Por ejemplo, en la figura 2.4 si un usuario es fan del equipo “*Golden State Warriors*”, este usuario tenderá a leer muchas noticias sobre baloncesto y sobre este equipo de la NBA durante varios años. Los autores denominan a este tipo de preferencias intereses a largo plazo. Por otro lado, otro tipo de intereses son los que evolucionan con el tiempo y se ven disparados por algún contexto o circunstancia concreta. Por ejemplo, en la figura 2.4 el haber leído varias noticias sobre la película “*Bohemian Rhapsody*” provoca que el usuario lea algunas noticias relacionadas del tipo “*Rami Malek Wins the 2019 Oscar*”, ya que Rami Malek es el actor principal de esta película, incluso sin que ese usuario haya leído noticias sobre este actor anteriormente. Los autores denominan a este tipo de preferencias intereses a corto plazo. Tanto los intereses a largo plazo como a corto plazo son importantes para la recomendación de noticias personalizada, y distinguirlos puede ayudar a construir mejores representaciones de los usuarios.

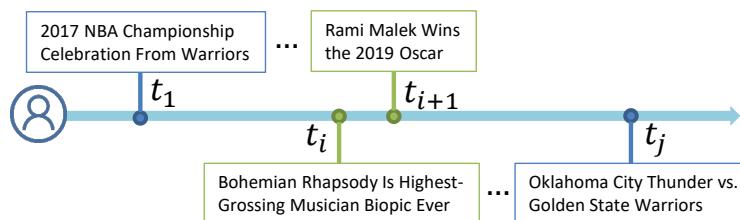


Figura 2.4: Ejemplo ilustrativo de intereses a corto plazo y a largo plazo en noticias [31].

En el artículo mencionado se propone un sistema de recomendación de noticias basado en redes neuronales, que utiliza representaciones de los intereses del usuario a largo y a corto plazo (**LSTUR**). Al igual que el resto de algoritmos que hemos comentado, es posible dividirlo en dos componentes, el *encoder* de noticias y el de usuarios. El *encoder* de noticias se utiliza para aprender la representación

de los artículos a partir de sus títulos y sus categorías. Los autores utilizan mecanismos de atención para construir la representación de las noticias, seleccionando las palabras más importantes. El *encoder* de usuarios se divide en dos módulos, uno para representar al usuario a largo plazo (*LTUR*) y otro para hacerlo a corto plazo (*STUR*). En *STUR* utilizan una red *GRU* para aprender las representaciones a corto plazo a partir de las noticias visitadas. En *LTUR* se aprende las representaciones de los usuarios a partir de *embeddings* de sus IDs, sin fijarse en nada más. Además proponen dos métodos para combinar estas representaciones a corto y largo plazo. El primero consiste en utilizar las representaciones a largo plazo para inicializar el estado oculto de la red *GRU* del modelo *STUR*. El segundo consiste en concatenar las representaciones a largo plazo con las de corto plazo, para así obtener un vector que unifique a ambas.

Predicción de clics

En todos los modelos, se calcula la probabilidad de ser elegida de cada una de las noticias candidatas (\hat{y}) como el producto escalar del vector que representa al usuario (U) y los vectores que representan a las noticias candidatas (r^c), es decir: $\hat{y} = U^T r^c$, de una forma muy similar a la explicada en el apartado 2.1.2, factorización de matrices.

2.4. Aplicaciones web

Para condensar todo lo explicado en las anteriores secciones y ejemplificar una de sus posibles aplicaciones, en este apartado se presentará la forma en la que se ha creado una interfaz gráfica en forma de aplicación web que permite visualizar y comparar diferentes modelos de recomendación de noticias. En primer lugar se hablará sobre la forma en los modelos de aprendizaje automático han sido desplegados, en segundo lugar cuáles han sido las tecnologías utilizadas a la hora de desarrollar la página web y por último las diferentes opciones que hemos barajado para desplegar nuestra web en algún servicio de computación en la nube.

2.4.1. Desplegar un modelo: API Rest

En la actualidad cada vez más empresas utilizan modelos de aprendizaje automático en sus aplicaciones. Es por ello que se ha tenido que buscar la manera de tratar a estos modelos como una pieza más de software, a pesar de que tienen algunos puntos bastante característicos.

Para el software “tradicional” existen las denominadas *DevOps* [32], también conocidas como “infraestructura ágil”, se trata de un conjunto de prácticas en las cuales los desarrolladores y los ingenieros de operaciones trabajan juntos a lo largo de todo el ciclo de vida del software, con el objetivo de acortar el mismo y conseguir desplegar software a producción de una manera más rápida y con mayor escalabilidad. Por el contrario, en el mundo del aprendizaje automático tenemos *MLOps* [33], que imitan

la filosofía de las *DevOps* pero esta vez utilizando modelos de aprendizaje automático, en lugar de software tradicional.

La principal característica diferenciadora de estos modelos respecto a una pieza de software tradicional es su capacidad de aprender, por ejemplo a través de un entrenamiento. Este es un punto muy a tener en cuenta al tratar de desplegarlo en un sistema en producción, ya que el modelo que se encuentre en explotación en cada momento deberá estar correctamente entrenado, y cualquier pequeña modificación en este modelo es probable que suponga un nuevo entrenamiento del modelo o incluso, en algunas ocasiones, será necesario entrenar de nuevo el modelo sin modificar su arquitectura. De hecho, en *DevOps* es muy común hablar de las prácticas *CI/CD* (integración continua y despliegue continuo, del inglés *continuous integration, continuous deployment*), mientras que en *MLOps* ya se está empezando a hablar de *CI/CD/CT* (integración continua, despliegue continuo y entrenamiento continuo, del inglés *continuous training*).

Como se verá en el resto del documento, en este trabajo no hemos utilizado ninguna plataforma de *MLOps*. El principal motivo para tomar esta decisión es que los modelos no van a sufrir cambio alguno, solamente van a pasar a producción una vez y no va a haber iteraciones sobre los mismos. Además estos modelos se van a desplegar ya entrenados y no va a haber modificaciones ni en los datos ni en el modelo, por lo que no habrá que repetir el entrenamiento en ningún modelo. Por lo tanto, lo que hemos hecho finalmente teniendo todo esto en cuenta, es imitar algunas de las metodologías que se usan en este tipo de plataformas, por ejemplo, el despliegue de un modelo encapsulado dentro de una API.

Una de las formas más sencillas de desplegar un modelo es utilizar una *API Rest*, encapsulándolo detrás de una URL y solicitando predicciones a través de peticiones HTTP. Para ello hemos utilizado una de las librerías que más relevancia ha adquirido en los últimos años precisamente gracias a lo rápida y eficiente que es para la tarea que nos atañe. Se trata de *Fast API*⁴, una de las librerías Python más utilizada para desplegar modelos de aprendizaje automático⁵. Antes de tomar esta decisión fueron descartadas librerías como *Django* o *Flask*, la primera ya que hemos considerado que era demasiado pesada, nos aporta muchísimas herramientas que no vamos a utilizar y mermaría el rendimiento de nuestra API; la segunda justo por ser el extremo contrario a *Django*, era demasiado básica y echábamos en falta algunas cosas, como una documentación automática de la API o algunas características de seguridad, que si aporta *FastAPI*. Solamente barajamos la opción de librerías Python debido a la gran cantidad de recursos que posee este lenguaje de cara a aprendizaje automático y a que los algoritmos descritos en 2.3 ya se encontraban desarrollados en él.

⁴Ramírez, Sebastián 2018, *Fast API*, Github, accessed 2021-02-12. <https://github.com/tiangolo/fastapi>

⁵Mavuduru, Amol 2021, *How you can quickly deploy your ML models with FastAPI*, Medium, accessed 2021-04-03. <https://towardsdatascience.com/how-you-can-quickly-deploy-your-ml-models-with-fastapi-9428085a87bf>

2.4.2. Interfaz Web: Flask

Gracias a tener el modelo encapsulado en la API solamente necesitamos una página web que consuma de esta, recogiendo los datos del usuario, mandándolos a la API y mostrando la respuesta de una manera agradable. Para analizar las tecnologías utilizadas en esta aplicación nos centraremos en dos apartados, por un lado el *backend* y por otro el *frontend*, aunque en primer lugar cabe destacar que ambos consumen de la API, tanto el *backend*, utilizando la librería *requests* de Python, como el *frontend*, directamente desde JQuery utilizando peticiones asíncronas (**AJAX**).

En el apartado de *backend* barajamos diferentes opciones, en primer lugar de lenguajes, que de nuevo nos quedamos con Python, debido a que no necesitábamos una web tan extensa como para crear un proyecto en Java y a que lenguajes como C podrían quedarse a demasiado bajo nivel. Dentro de Python en esta ocasión barajamos principalmente Django y Flask, para finalmente quedarnos con la segunda, ya que de nuevo Django era demasiado pesado para lo que necesitábamos, ni siquiera íbamos a utilizar una base de datos. A pesar de esto, cabe destacar que existen algunas alternativas que habrían sido sin duda una buena decisión, por ejemplo algunas basadas en Javascript, como NodeJS o ReactJS, pero fueron descartadas debido a que ya teníamos el entorno Python preparado.

De cara al *frontend* dudamos bastante menos, como casi todas las páginas web actuales utilizamos HTML, CSS y Javascript, incorporando las librerías de Bootstrap y JQuery. Para empezar partimos de la plantilla “Blog”, disponible en la página web de Bootstrap⁶, pero la modificamos prácticamente entera.

2.4.3. Infraestructura

El primer paso sería el entrenamiento de modelos, para el cual hemos probado la plataforma de inteligencia artificial de Azure (Microsoft), la cual nos facilita acceso a cuadernos de Jupyter y nos permite utilizar máquinas con aceleradores *GPU* dedicados. Pero por desgracia la versión gratuita solo nos da acceso a *GPUs* muy básicas, por lo que acabamos entrenando estos modelos en un ordenador de sobremesa.

Tras el entrenamiento, llega el momento del despliegue, tanto de la API como de la página web, y para ello hemos barajado diferentes opciones, desde las más simples hasta las más complejas. En primer lugar, la API, conviene destacar que se trata de un programa que va a tener que ejecutar modelos de redes neuronales, administrar todos los datos de la aplicación y además responder a las peticiones de la página web. Para poder hacer todo esto es necesario que tenga un poder de cómputo más que aceptable, además de requerir una configuración concreta de paquetes. Teniendo en cuenta todo esto, primero nos planteamos desplegar en un servicio en la nube. Descartamos Heroku debido a problemas de compatibilidad debido al entorno Python que estábamos utilizando, un entorno personalizado de Conda, después de esto pasamos a **Amazon Web Services (AWS)**, este si nos

⁶Bootstrap, *Bootstrap Blog Example*, accessed 2021-03-03. <https://getbootstrap.com/docs/4.0/examples/blog/>

permitió instalar algunos paquetes, pero debido a que la versión gratuita tiene una memoria limitada, no fue posible poner en funcionamiento la API completa. Debido a esto, de nuevo, acabamos descartando cualquier servicio en la nube, y conformándonos con desplegar esta página web en nuestro ordenador de sobremesa. A pesar de esto, también sería posible utilizar algún dispositivo pensando para este tipo de implementaciones, una Raspberry probablemente se quedaría pequeña, pero un Nvidia Jetson si que sería suficiente.

Por último, de cara al despliegue de la página web, sin duda podría haber sido en un servidor independiente, por ejemplo Heroku, de hecho hemos probado y funciona a la perfección. El problema es que al tener la API en un ordenador doméstico, nos encontramos bajo una IP dinámica, por lo que, como no era el corazón del TFG, decidimos dejarlo todo instalado en local e invertir ese tiempo en otras tareas de investigación y desarrollo, que se comentarán en el siguiente capítulo.

Si tuviéramos los recursos económicos necesarios lo ideal sería tener la API alojada en un servicio como [AWS Sagemaker](#), especializado en este tipo de implementaciones; con un *endpoint* público, del cual consumiría la web, que podría estar alojada también en [AWS](#) o incluso podría usarse simplemente Heroku con su versión gratuita, ya que ahora entonces sí que tendríamos acceso sin problema a la API.

DISEÑO E IMPLEMENTACIÓN

En este capítulo se detallará cuál ha sido el diseño que se ha seguido a la hora de desarrollar el software central de este trabajo, su estructura general, el ciclo de vida elegido y los requisitos de este proyecto. Además se darán detalles sobre la implementación de este sistema, explicando en detalle el funcionamiento de cada uno de los módulos del mismo.

3.1. Diseño

En esta sección, se presenta el diseño del trabajo, dando detalles sobre su estructura, explicando los diferentes módulos existentes y describiendo la comunicación que puede haber entre cada uno. Además, detallaremos el ciclo de vida y el tiempo aproximado que se ha dedicado a cada apartado del trabajo a lo largo del curso.

3.1.1. Estructura

El sistema se encuentra claramente dividido en diferentes módulos, en la figura 3.1 podemos ver cuál ha sido la distribución elegida. Podría resumirse en los siguientes puntos:

- Administrador de datos (*Data Manager*). Se encarga de descargar el *dataset* e implementa las funciones necesarias para obtener los datos necesarios en forma de *DataFrame* de pandas. A él se conectan tanto los modelos a la hora de entrenar como la API a la hora de evaluar los modelos y servir a la página web.
- Modelos (*Models*). Todos los modelos de recomendación son entrenados en cuadernos Jupyter y posteriormente almacenados serializados, para su posterior explotación. Tanto los modelos de *Microsoft Recommenders* como los implementados en la librería *ClassicRecommender* pasarán por esta etapa.
 - *Recommenders*. Librería de código abierto de Microsoft, de la cual hemos elegido los tres modelos (*NRMS*, *NAML* y *LSTUR*) que nos han parecido más interesantes y hemos incluido en nuestra aplicación.
 - *ClassicRecommenders*. Librería propia en la que se han desarrollado los algoritmos detallados en la sección 2.1.1.
- API Rest. Prácticamente el corazón de la aplicación, en ella se leen modelos ya entrenados (que fueron serializados en Jupyter) para posteriormente ser cargados en RAM y evaluados bajo demanda de la página web. Además sirve la información relativa a las noticias y los usuarios que solicite también la web.
- Aplicación web (*Web App*). Se trata de la página web consumidora de la API.

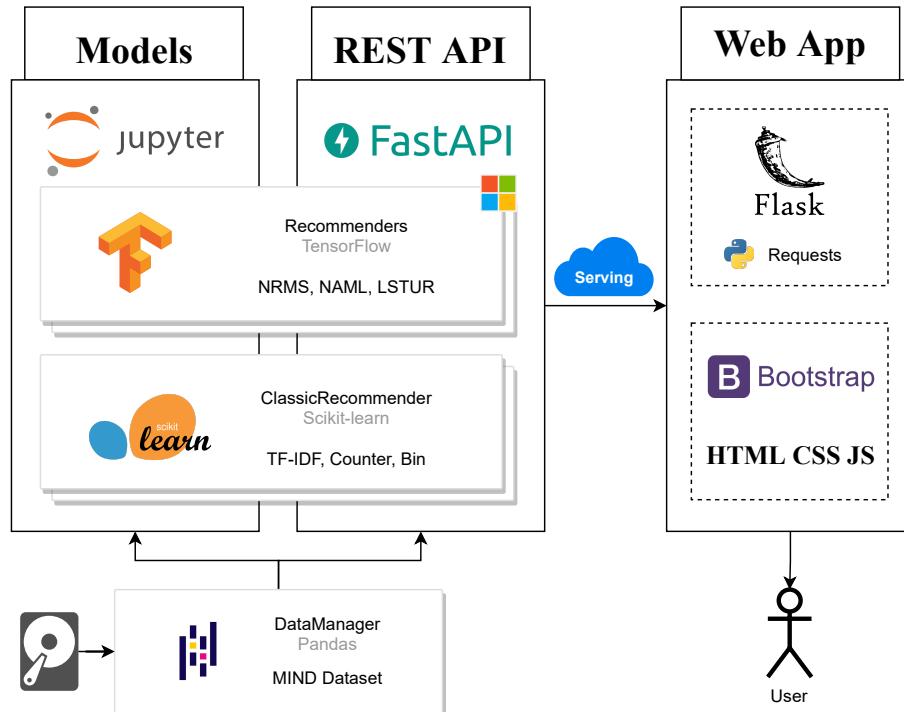


Figura 3.1: Diagrama estructura general.

Diagrama de secuencia

El funcionamiento de la aplicación que hemos repetido en varias ocasiones a lo largo de este documento, puede verse detallado en la figura 3.2. En ella podemos ver un diagrama de secuencia que define cuál sería la sucesión de llamadas que tienen lugar en un ejemplo de uso típico de la aplicación, detallando en todo momento el tráfico que habría entre la *API REST* y la página web. En este ejemplo, desde la página web se seleccionan un identificador de usuario y unos algoritmos, y esta devuelve las recomendaciones correspondientes.

3.1.2. Ciclo de vida

Se trata de un ciclo de vida iterativo. En un primer momento el sistema solamente era capaz de trabajar con los modelos de *ClassicRecommender*, pero a medida que fueron sucediendo iteraciones se fueron ampliando, adaptando los modelos de *recommenders*. El principal motivo de utilizar esta metodología ha sido el tener la posibilidad de tener un producto entregable en cada iteración, sobre el que poder hacer pruebas, comparar modelos e incluso, en base a diferentes resultados, decidir que otros modelos podrían aportar más y deberían ser añadidos al sistema. Además, al tratarse de un sistema tan modularizado no había ningún problema en desarrollarlo de esta manera, más bien al contrario, era la forma más intuitiva. Otra técnica utilizada que ha optimizado bastante el ciclo de vida de la aplicación, ha sido el uso de módulos ficticios (en inglés denominados *dummy*), sobretodo en las primeras etapas del desarrollo. Gracias a este método, hemos podido por ejemplo desarrollar la página

web utilizando una API que no se encontraba completa todavía, solo simulaba su funcionamiento de manera aleatoria.

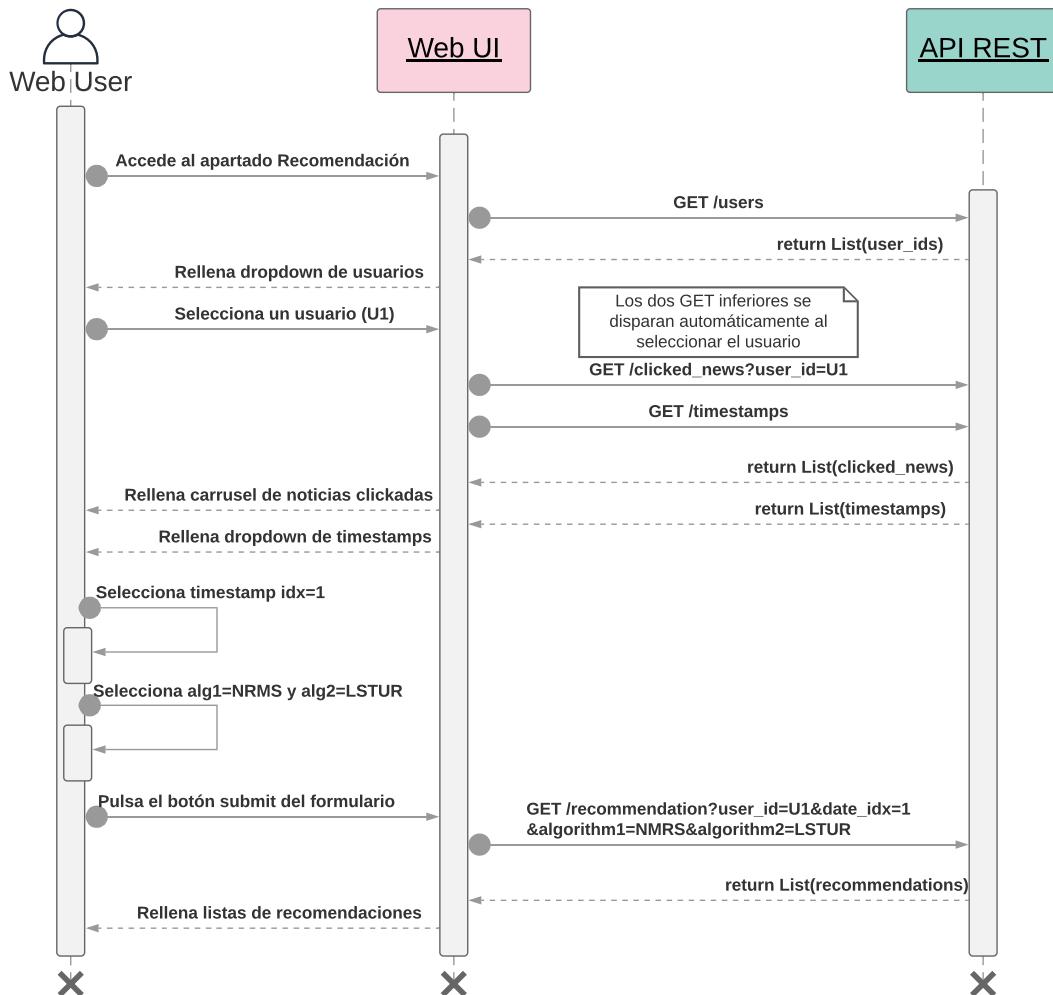


Figura 3.2: Diagrama secuencia ejemplo.

Temporalidad

A continuación describiremos cuál ha sido el proceso seguido para llegar a obtener el trabajo que estamos presentando en la actualidad. Cabe destacar que, incluso meses antes de decidir el tema concreto del **TFG** se empezó a investigar sobre diferentes temas relacionados con él, por ejemplo empezando desde cero en el mundo del procesamiento del lenguaje natural (ver 2.2).

Después de muchas horas de lectura de bibliografía, empezó a definirse algo más el tema, coincidiendo con la publicación del *dataset MIND* y los algoritmos de recomendación de noticias de *recommenders*. En ese punto empezamos a investigar sobre estos algoritmos, en primer lugar simplemente entendiendo cuál era su funcionamiento, tarea nada sencilla con los conocimientos de ese momento y con un código no demasiado bien documentado. Tras esto ya teníamos claro el tema del **Trabajo de Fin de Grado (TFG)**, y con ello empezamos a investigar sobre recomendación de manera general, para

poder describir en el trabajo la progresión desde los algoritmos más sencillos (ver 2.1.1) hasta los más complejos (ver 2.3.3), en ese momento desarrollamos las primeras versiones de la página web, la API y la librería *ClassicRecommender*. Aquí es donde verdaderamente empezó el ciclo de vida iterativo.

Una vez integradas esas primeras versiones llegó el momento de combinar los algoritmos complejos de la librería *recommenders* con nuestro sistema de recomendación, no fue tarea fácil, fueron muchas horas las que nos llevó conseguir adaptar ese código (preparado casi exclusivamente para la competición de *MIND*) a nuestra aplicación, en la que tendrá que evaluar a los usuarios uno a uno.

Cuando ya teníamos todo esto funcionando, apareció Google para iluminarnos en lo que podría ser nuestra siguiente “aventura”, publicando un artículo [13] a escasas semanas de entregar el trabajo. En este artículo se presentaban una versión de la estructura *Transformer* presentada en la sección 2.2.3, pero que en lugar de utilizar atención utilizaba transformadas de Fourier. Esto nos llevó a intentar hacer lo mismo partiendo de los algoritmos de *recommenders*, en concreto *NRMS*. Le dedicamos mucho tiempo a hacer una implementación en Tensorflow de la red *FNet* que presentaba el artículo y a incorporarla en el código de *NRMS*, el cual por suerte ya entendíamos bastante bien, pero por desgracia no logramos unos resultados destacables, ni en rendimiento ni en tiempo de ejecución.

3.2. Análisis de requisitos

En esta sección se detallarán los requisitos que deberá cumplir el sistema. Estos se encuentran divididos, en primer lugar en funcionales y no funcionales, y en segundo lugar, dentro de los funcionales hemos creado diferentes secciones para cada módulo del sistema.

3.2.1. Requisitos funcionales

Requisitos de los datos

RF-1.– Los datos son estáticos, no se van a añadir ni quitar ninguno.

RF-2.– El sistema deberá funcionar, al menos, con los datos del *dataset MIND* tal cual vienen.

RF-3.– El sistema será capaz de descargarse los datos del servidor de Microsoft automáticamente en el caso de que no estuvieran ya en el disco del servidor.

RF-4.– El sistema mantendrá los datos cargados en RAM en forma de *DataFrame*, para así optimizar tiempos de respuesta. Al no modificar nada en ellos podrán borrarse y volverse a recuperar en cada reinicio.

RF-5.– Los datos mostrados en la página web serán exclusivamente los datos de test, nunca los de entrenamiento.

RF-6.– La página web no almacenará ningún dato, todos los datos necesarios se encuentran almacenados en la API.

RF-7.– La página web no almacenará ninguna información de los usuarios, estos deben ser completamente anónimos, sin *cookies* propias ni de terceros.

Requisitos de la página web

- RF-8.**– La página web tendrá al menos una página principal con información general.
- RF-9.**– La página web tendrá una página con un formulario, en función del cual se mostrarán unas recomendaciones u otras.
- RF-10.**– El sistema debe permitir elegir un ID de un usuario y una fecha concreta, para mostrar las recomendaciones que recibiría ese usuario en ese momento concreto.
- RF-11.**– El sistema debe permitir elegir en ese formulario qué algoritmo utilizar para recomendar.
- RF-12.**– La página web debe mostrar los IDs de usuarios aleatorias enviadas desde la API.
- RF-13.**– La página web mostrará un resultado de recomendación en cuanto se le indique el usuario (sin necesidad de tocar ningún botón), si este usuario tuviera varias apariciones en el *dataset*, se tomará por defecto la primera, dando la oportunidad al usuario de cambiarlo si así lo desea.
- RF-14.**– La página web permitirá comparar los rankings de dos algoritmos, mostrándolos uno al lado del otro en dos columnas.
- RF-15.**– La página web mostrará cuales son las noticias que fueron seleccionadas dentro de cada ranking.
- RF-16.**– La página web permitirá dar clic en las noticias y ver el contenido de las mismas.

Requisitos de la API Rest

- RF-17.**– La API permitirá seleccionar dos algoritmos con los que hacer la predicción, de esta manera se podrán comparar de manera visual en la página web.
- RF-18.**– Las peticiones y las respuestas de la API tendrán los mismos parámetros para todos los algoritmos, incluyendo los clásicos y los de aprendizaje profundo.
- RF-19.**– La API tendrá una documentación web en la que consultar las posibles llamadas que se pueden hacer y qué parámetros necesita cada una. Además será posible probar la API directamente desde esa documentación.
- RF-20.**– La API debe facilitar IDs de usuarios aleatorias. Estas servirán de ejemplo al usuario de la página web para elegir cuál utilizar.
- RF-21.**– Los datos no podrán en ningún caso ser modificados por la API, por lo que no habrá métodos dedicados a esto.

Requisitos de los modelos de recomendación

- RF-22.**– Todos los modelos tendrán la misma salida: una lista con las puntuaciones de cada una de las noticias candidatas (a más puntuación más probabilidad de ser elegida).
- RF-23.**– El sistema debe permitir solicitar recomendaciones de un usuario concreto en un momento (*timestamp*) concreto.
- RF-24.**– Los modelos de recomendación deberán poder devolver recomendaciones a todos los usuarios del *dataset*, independientemente del comportamiento del resto de módulos en el sistema.

3.2.2. Requisitos no funcionales

- RNF-1.**– La página web tendrá un diseño *responsive*, permitiendo adaptarse a diferentes tamaños de pantalla.
- RNF-2.**– La página web se diseñará utilizando una paleta de colores sencilla y un diseño moderno, priorizando la facilidad de lectura.
- RNF-3.**– La página web tendrá un diseño intuitivo, cualquiera podrá utilizarla sin necesidad de utilizar ningún manual

de usuario.

RNF-4.- En caso de utilizar datos de usuarios (para recomendar) estos se utilizarán anonimizados, como los que vienen en el *dataset* MIND, con el objetivo de respetar la Ley de Protección de Datos.

RNF-5.- La API estará protegida mediante una *API Key* en las llamadas más pesadas (recomendación), mientras que otras más ligeras (muestras de usuarios o noticias), no lo estarán.

RNF-6.- La API deberá ofrecer tiempos de respuesta bajos, para ello utilizará por ejemplo un acelerador hardware GPU.

RNF-7.- Todo el código estará programado en Python y se ejecutará bajo un entorno *conda*, la API utilizando el archivo *yaml* de *recommenders* y la web uno propio.

RNF-8.- Los modelos se entrenarán en un cuaderno de Jupyter, para poder hacerlo de una forma más cómoda, además de reutilizable.

3.3. Implementación

En esta sección describiremos en detalle cómo ha sido la implementación de los módulos descritos en el apartado 3.1.1. Empezaremos describiendo cuál es el flujo de ejecución del sistema, para posteriormente explicarlo módulo a módulo.

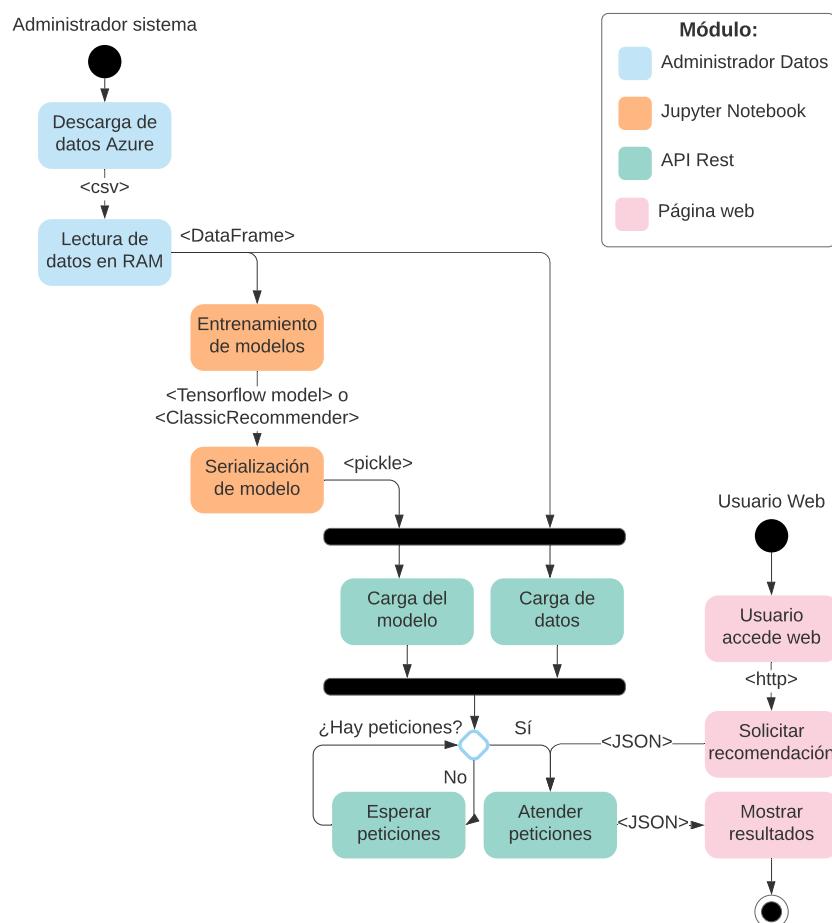


Figura 3.3: Ejemplo flujo de la aplicación.

Flujo de la aplicación

En la figura 3.3 podemos ver un diagrama de actividades que describe, en dos caminos, cuál sería el flujo de ejecución de la aplicación. En la parte izquierda podemos ver cuál es el camino que se seguiría desde el punto de vista del administrador, que pone en marcha el sistema. En la parte derecha veremos el punto de vista de un usuario de la página web, que solo solicita una recomendación.

Administración de datos

En primer lugar hablaremos del lugar del que parte todo el trabajo, los datos. Como ya hemos comentado anteriormente, todo el trabajo gira en torno al *dataset MIND*, de Microsoft. Un *dataset* orientado al estudio de métodos de recomendación de noticias (más información apartado 2.3.1). En este apartado nos centraremos en la forma en la que hemos utilizado estos datos, desde su descarga hasta su presentación en la página web.

Por otro lado, todo lo relacionado con el *dataset* ha quedado encapsulado en una sola clase creada por nosotros mismos, la clase *MINDDataset*. En esta clase tenemos un constructor, este requiere una ruta de la carpeta en la que se quieren almacenar todos los archivos del *dataset* y el tipo de *dataset* que se quiere utilizar. Respecto a este segundo punto cabe destacar que existen tres versiones (*demo*, *small* y *large*) y que en este trabajo hemos utilizado exclusivamente la versión “*small*”, como ya justificaremos en el apartado de resultados. Continuando con el constructor, este será el encargado de almacenar el *path* concreto de cada uno de los archivos que conforman el *dataset*, así como de descargarlo en el caso de que este aún no se encuentre en disco.

Además del constructor, esta clase cuenta con diferentes funciones en las que es posible leer las noticias y los comportamientos de los usuarios directamente utilizando *DataFrames* de pandas. De hecho todo el flujo de datos de la aplicación se hace utilizando este tipo de datos, debido a su facilidad para trabajar directamente en JSON, y lo que eso supone de cara al uso de API.

Modelos de recomendación

Recomendación basada en técnicas clásicas

Estos modelos tienen un funcionamiento muy sencillo, ya que solo utiliza recomendación basada en contenido (más información en 2.1.1). De esta manera, estos modelos aprenden la representación de los usuarios basándose únicamente en el contenido de las noticias que han visitado en un pasado, calculando su similitud con el contenido de las noticias candidatas.

Como detallamos en la sección 2.1.1 tendremos que transformar cada texto a un vector, para ello hemos utilizado los algoritmos allí descritos, en concreto hemos decidido utilizar la implementación de la librería *SkLearn*¹. Por otro lado, en cuanto a las funciones de similitud es posible utilizar la *distancia*

¹ SkLearn, *Feature Extraction: Text*, accessed 2021-02-10. <https://scikit-learn.org/>

de Manhattan, la similitud coseno y la distancia euclídea, todas también de *SkLearn*.

Para utilizar la implementación de *SkLearn* hemos decidido desarrollar una librería propia (*Classi-
cRecommender*), la cual encapsula el comportamiento de todos los modelos para que estos puedan
ser llamados de la misma manera. Dentro de esta librería tenemos la posibilidad de entrenar nuestros
modelos (el entrenamiento consiste solamente en analizar las palabras que contienen las noticias pa-
ra así construir el vocabulario, que a su vez definirá el tamaño de la representación de cada noticia).
Además del entrenamiento podremos realizar predicciones, que consistirán en calcular las distancias
de cada una de las noticias candidatas a las noticias del historial, de forma que cada noticia candidata
tenga una distancia estimada a todas las noticias visitadas.

Por otro lado esta librería incluye además un módulo utilidades, en el cual hemos desarrollado dife-
rentes funciones para preprocesar algunos datos del *dataset* (por ejemplo dividir las listas de noticias
que vienen en forma de cadena de texto) y además las funciones para ejecutar la evaluación de nues-
tros algoritmos, teniendo la posibilidad de optimizar la ejecución de esta evaluación paralelizando los
cálculos de similitud entre los diferentes núcleos de nuestra CPU.

Por último, cabe destacar un intento de mejora realizado en este módulo, el cual consiste en uti-
lizar *BERT* (más información en sección 2.2.3) para preprocesar el texto. Es decir, antes de que el
texto pase al *vectorizer* de *SkLearn*, utilizaremos un *tokenizer* de *BERT* para preprocesar nuestras
palabras. De esta manera cada palabra queda reducida a un *embedding*, y a la hora de tener en cuen-
ta las frecuencias de cada una de las palabras de la frase estaremos teniendo en cuenta no solo la
palabra como tal, sino también su contexto, como ya explicamos en la sección 2.2. Por desgracia, co-
mo ya veremos en la sección de resultados, esta aproximación no tuvo mucho éxito, además de ser
extremadamente ineficiente.

Recomendación basada en aprendizaje profundo

En este apartado explicaremos como hemos incorporado a nuestro sistema los algoritmos basados
en aprendizaje profundo detallados en 2.3.3. La implementación de Microsoft de estos algoritmos venía
preparada para ser utilizada en la competición de *MIND*, es por ello que el objetivo de los algoritmos
era hacer todas las predicciones sobre el *dataset* de test de golpe, para posteriormente evaluar ese
resultado. Sin embargo en nuestro sistema era necesario hacer predicciones de manera individual y lo
más rápido posible, por lo que ha sido necesario desarrollar una librería intermedia (*mind_helper.py*)

La forma de hacer esto es, en primer lugar, utilizando el modelo pre-entrenado generar la represen-
tación de las noticias candidatas y la representación del usuario. Con ellos podremos hacer el producto
escalar de los vectores correspondientes y con ello calcular el ranking de noticias de ese usuario en
concreto (para más información ver 2.3.3).

Una gran parte del tiempo dedicado a este trabajo se ha invertido en estudiar y replicar la estructura
de los módulos utilizada en estos algoritmos, la cual vamos a detallar a continuación. El funcionamiento
de estos modelos puede dividirse en dos partes: los iteradores de datos y los modelos.

Un iterador en *Python* es un objeto utilizado para iterar sobre objetos (iterables), como listas, tuplas, diccionarios y conjuntos. Pero en este caso cuando hablamos de iteradores no hablamos solo de eso, sino de un **framework** que provee datos a los modelos en forma de **mini-batch** (en grupos), en lugar de devolver todo de golpe y obligar a cargarlo todo en RAM. De estos iteradores existen dos, uno que trabaja únicamente con los títulos de las noticias (*mind_iterator.py*) y otro que trabaja con toda la información que tenemos de cada noticia (a excepción del cuerpo, que viene en forma de enlace), en concreto utiliza título, resumen, categoría y subcategoría (*mind_all_iterator.py*).

Por parte de los modelos, tenemos una estructura de clases relativamente sencilla, en la cual encontramos una clase “padre” (*base_model.py*) de la cual heredan todos los modelos. Estos modelos solamente sobrescriben los métodos relacionados con el *encoder* de noticias, el *encoder* de usuarios y el modelo como tal, el resto lo heredan.

Con todo esto en mente hemos integrado en nuestro sistema tres algoritmos, *NRMS*, *NAML* y *LSTUR* (sección 2.3.3). De estos, como ya se explicó en el estado de arte, tanto *LSTUR* como *NRMS* utilizan solamente el título de la noticia, y por tanto el iterador *mind_iterator.py*, mientras que *NAML* requiere más información sobre la noticia, por lo que utiliza *mind_all_iterator.py*.

Para la definición de estos modelos se ha utilizado la librería *Tensorflow*², y todas las capas declaradas en la misma, a excepción de las capas de *attention* y *self-attention*, las cuales han sido implementadas manualmente por los autores en la propia librería de *recommenders*.

Siguiendo este camino tratamos de desarrollar nuestro propio algoritmo de aprendizaje profundo. Este algoritmo estaría basado en el ya comentado *NRMS* pero sustituyendo sus capas de *attention* por capas *FNet*, utilizando una lógica similar a la que utilizó Google con la arquitectura *Transformer* (más detalle en sección 2.2.3). Para llevar a cabo esta implementación nos vimos obligados a desarrollar nuestra propia versión de *FNet*, lo cual no fue tarea para nada fácil, ya que la única versión existente de *FNet* se encuentra en la última versión de PyTorch, mientras que nosotros nos vemos obligados a utilizar *Tensorflow* en versión 1.15 (un poco antigua), ya que es la utilizada en la librería *recommenders*. Pero finalmente conseguimos hacerlo, declaramos la capa *FNet* junto con las capas de *attention* creadas por Microsoft y creamos nuestra propia versión de *NRMS*. A pesar de todo, como ya veremos en el siguiente capítulo, no obtuvimos unos resultados demasiado buenos y por ello finalmente decidimos no integrarlo en la librería, pero sin duda nos sirvió para aprender mucho sobre este tipo de redes neuronales.

Entrenamiento de modelos

Tanto para los modelos basados en técnicas clásicas como los más complejos basados en aprendizaje profundo hemos utilizado el mismo procedimiento para entrenar y desplegar los modelos: en primer lugar los modelos se entrena en cuadernos Jupyter, ya sea en local o en entornos alojados en la nube, como Google Colab o Azure; a continuación se serializa el objeto del modelo a un archivo y

²*Tensorflow*, accessed 2021-03-08. <https://www.tensorflow.org/>

por último se envía este archivo a la API, la cual solamente tiene que leerlo y ponerlo en producción.

La diferencia entre unos modelos y otros radica esencialmente en el punto de la serialización ya que los modelos clásicos es posible serializarlos directamente utilizando la librería *pickle* integrada en el propio Python, mientras que en los modelos de aprendizaje profundo no es así. Debido a la estructura de red utilizada por estos modelos y al uso de Tensorflow 1.15, para serializar estos modelos tendremos que hacerlo en varias partes. En primer lugar guardaremos los pesos de la red del *encoder* de noticias y el *encoder* de usuarios, y por otro lado almacenaremos el iterador y la representación de las noticias de test precalculadas. Con todo esto podremos enviar los archivos a la API y esta podrá reconstruir el estado exacto de cada uno de los correspondientes modelos, sin apenas gasto computacional.

API Rest

En cuanto a la API, como ya se ha explicado anteriormente se trata prácticamente del corazón de la aplicación, en ella se leen modelos ya entrenados (que fueron serializados en Jupyter) para posteriormente ser cargados en RAM y evaluados bajo demanda de la página web. Además sirve los datos relativos a las noticias y los usuarios que solicite también la web. Sin embargo, hay algo en lo que no se ha hecho hincapié en ninguna de las anteriores secciones, y es que la API no es una sola, sino que son cuatro. Tenemos 4 APIs debido a que por incompatibilidades de versiones de *CUDA* y *Tensorflow*, no ha sido posible ejecutar todos los modelos de aprendizaje profundo al mismo tiempo en un solo programa, y como este es uno de nuestros requisitos, para poder compararlos, hemos decidido crear una API para cada modelo de aprendizaje profundo (una para *NRMS*, otra para *NAML* y otra para *LSTUR*). Estas API auxiliares comparten código, solo que cada una se ejecuta haciendo uso de su propio archivo de configuración, en el que establecemos el modelo que vamos a utilizar, los archivos donde se encuentra el modelo serializado y el puerto donde se va a desplegar. Además de estas tres API, existe otra más, la principal, que será la encargada de llamar a las auxiliares cuando sea necesario y con la única con la que la web se comunicará.

En cuanto a las APIs auxiliares solo cabe destacar que se trata de programas muy sencillos, que leen el archivo de configuración, cargan el modelo y levantan el servidor, el cual solamente tiene un servicio, el de recomendación. Por último habría que añadir que también se encuentran equipados con una memoria caché, la cual consiste en un diccionario, que almacena las peticiones que se le van haciendo a la API junto a la recomendación que da el algoritmo. De esta manera conseguiremos evitar sobrecargar la *GPU* pidiéndole recomendaciones que ya ha hecho, cosa que no tendría ningún sentido, ya que, como los datos son estáticos y los modelos no se re-entrenan, siempre devolverá lo mismo. Este diccionario se va actualizando a medida que le llegan peticiones y se serializa en un archivo a la hora de apagar la API. En resumen, contamos con tres diccionarios que son utilizados como memoria caché de cada uno de los modelos.

Una vez comentado el funcionamiento de las API auxiliares, nos podemos centrar en la principal; en ella se encuentran los modelos de recomendación clásicos, y todas las peticiones a través de la cual solicita datos la página web, es decir, todo excepto los tres modelos de aprendizaje profundo.

Además de todo lo mencionado, la API cuenta con un crawler encargado de la recolección de imágenes de noticias, es decir, como el *dataset* no tenía imágenes de portada para cada una de las noticias, decidimos programar este *crawler*, el cual busca el titular en Google Imágenes y se queda con la URL de la imagen que se encuentre en el primer resultado. Todas estas imágenes se encuentran cacheadas en otro diccionario, donde la clave es el ID de la noticia y el valor es el URL de la imagen correspondiente a esa noticia, este diccionario se va actualizando a medida que se van consultando las imágenes y se serializa a un archivo al apagar la API.

Todas las API han sido programadas en Python utilizando la librería FastAPI. Además, como ya hemos comentado, la web solamente se pondrá en contacto con la API principal y lo hará utilizando los siguientes servicios.

- **GET /recommendation:** Enviando un ID de un usuario, una fecha y dos algoritmos, se devolverá un *DataFrame* de pandas con toda la información relativa a cada una de las noticias candidatas, incluyendo una columna con la recomendación del primer algoritmo, otra con la recomendación del segundo y por último una con lo que realmente ocurrió, es decir, a cuál dio clic finalmente.
- **GET /clicked_news:** Dado un ID de usuario devuelve un *DataFrame* con todas las noticias que ha visto ese usuario en el pasado.
- **GET /timestamp:** Dado un ID de usuario devuelve una lista con todas las situaciones en las que se tiene almacenado el comportamiento de ese usuario, es decir, diferentes listas de noticias candidatas, a las que dio o no clic.
- **GET /users_sample:** Devuelve una muestra de usuarios completamente aleatoria, utilizada para que el usuario de la página web tenga ejemplos de usuarios a evaluar.
- **GET /news_sample:** Devuelve una muestra de noticias completamente aleatoria, utilizada en la página principal de la web.

Página web

Por último, llegamos al apartado de la página web, cuyo principal objetivo es ofrecer la posibilidad de simular ser un usuario de un portal web de noticias en un momento concreto, en la cual puede ver su historial de noticias que ya ha visto junto con otras noticias que todavía no ha visto. Estas últimas se encuentran repetidas en dos columnas, cada una de ellas ordenada respecto a un algoritmo.

Hemos utilizado las siguientes tecnologías, que ya se describieron en 2.4.2: en el *backend* un servidor utilizando la librería Flask en lenguaje Python, mientras que en el *frontend* los elegidos han sido HTML, CSS y Javascript, utilizando las librerías Bootstrap y JQuery. De cara al estilo de la página web este esta basado en la plantilla “Blog” de Bootstrap, pero con numerosas modificaciones.

La página web ha sido programada utilizando *templates* de Jinga2 (utilizados en Flask), que no son más que documentos HTML con fragmentos de código Python en ellos, este código Python lo que hace es básicamente construir el documento HTML final en base a reglas y datos descritas en el propio servidor Flask. Esto nos permitirá generar páginas web de manera dinámica, en nuestro caso utilizamos esto en numerosas ocasiones:

- En primer lugar, utilizamos un *template* base, del cual heredan casi todos los demás, es el que contiene la

Figura 3.4: Apartado de recomendación página web (Ver anexo A para más imágenes).

cabecera y el pie de página de nuestra web, además desde él se importan los archivos CSS y las librerías JavaScript comunes a todas las páginas.

- Tenemos un *template* con la página principal de la web, una página bastante sencilla, pero con una sección de noticias que se rellena automáticamente con noticias aleatorias enviadas desde el Flask, el cual, previamente hizo una petición a la API para conseguirlas.
- *Template* de recomendación, el cual codifica el esqueleto de la página más compleja de la web (figura 3.4), en ella se encuentra el formulario principal, se importa el código JavaScript necesario para el correcto funcionamiento de la página y además se describen los “huecos” que ocupará el historial de noticias y las noticias recomendadas.

- Por otro lado tenemos dos *templates* algo diferentes, ya que no heredan del base ni son páginas web como tal, sino que sirven para llenar los “huecos” mencionados en el apartado anterior: uno rellena un carrusel Bootstrap con las noticias del historial y otro rellena los dos rankings de noticias recomendadas.

La forma de generar estos *templates* es un poco complicada de explicar (y de implementar), pero es la clave para conseguir el funcionamiento que queríamos y evitar actualizar la web completa todo el rato, en resumen consiste en lo siguiente: en primer lugar un evento de la página web dispara la función JQuery, esta función llama a un servicio en el *backend* del servidor que lo que hace es, con la información enviada desde JQuery, generar un HTML correspondiente, este HTML se manda a JQuery y este rellena el hueco que había con él, o sustituye el código existente si el hueco estuviera relleno. Los eventos que disparan este código JQuery son una modificación del id del usuario en el caso del historial de noticias, o un clic en el botón de *submit* en el caso de la recomendación. Además cabe destacar que todas las llamadas hechas desde JavaScript son totalmente asíncronas (**AJAX**). (Para más información sobre el tráfico de llamadas entre la API y la página web revisar el diagrama de secuencia de la figura 3.2.)

PRUEBAS Y RESULTADOS

En este capítulo se analizarán los resultados de los diferentes algoritmos que hemos incluido en el sistema. En primer lugar analizaremos el entorno de pruebas sobre el que se ejecutarán estas pruebas, en segundo lugar se analizarán los resultados de los algoritmos basados en métodos clásicos y por último comentaremos los resultados obtenidos con los algoritmos basados en aprendizaje profundo. Este último punto puede parecer redundante, ya que muchos de esos resultados están analizados en el artículo [21], pero todos ellos utilizan la versión del *dataset* grande, al contrario que nosotros, por lo que hemos decidido replicar esos resultados adaptándonos a los recursos con los que contábamos.

4.1. Entorno de pruebas

A la hora de realizar las pruebas, todas las mediciones se han realizado en un entorno local, con las características técnicas detalladas en la tabla 4.1. A destacar que, como podemos ver, estamos utilizando un entorno de Python basado en Anaconda, en concreto hemos utilizado el recomendado por la librería *recommenders* de Microsoft, ya mencionada anteriormente.

Componente	Características
CPU	Intel Core i7 4770k
GPU	Nvidia GTX 1660 SUPER 6 GB VRAM
RAM	16 GB
S.O.	Ubuntu 20.04 LTS
Python Enviroment	Python 3.6.11 conda-forge

Tabla 4.1: Tabla características técnicas equipo de pruebas.

Antes de elegir definitivamente este equipo, hicimos pruebas en algunos servicios de computación en la nube. En concreto probamos Google Colab y Azure AI. En el primero los recursos estaban bastante bien, se nos daba acceso a una GPU Nvidia Tesla T4, pero se encuentra limitado a la hora de hacer ejecuciones de muchas horas. A partir de la hora y media, empieza a aparecer un *captcha* en la pantalla, por lo que te obliga a estar pendiente, y aunque lo estés, a las doce horas el entorno se apaga definitivamente, y se borran todos los archivos que allí dentro queden. Esto es así ya que lo que Google ofrece gratis es un entorno interactivo, no un servicio de computación en la nube, para eso venden Vertex AI, el cual por desgracia no he podido utilizar al ser de pago.

Por parte de Microsoft Azure AI el problema no fueron las limitaciones de tiempo, ya que podías utilizar las máquinas todo el tiempo que necesitases, esta vez el problema son los recursos. Con la cuenta gratuita para estudiantes solamente dan acceso a una Nvidia Tesla k80, la cual parece incluso estar compartida entre varios usuarios, porque ofrece un rendimiento bastante deficiente.

Teniendo todo esto en cuenta y haciendo algunas mediciones de tiempo finalmente decidimos quedarnos con el equipo descrito en la tabla 4.1, a pesar de que en este también contaría con limitaciones de tiempos, al tratarse de un ordenador doméstico y no poder estar siempre encendido.

4.2. Experimentos

En esta sección detallaremos los experimentos realizados para este **TFG**, para ello, empezaremos dando algún detalle más sobre los datos a utilizar, continuaremos con una pequeña comparación de los algoritmos basados en técnicas clásicas, siguiendo con otro de las técnicas basadas en aprendizaje profundo y por último, haremos un pequeño análisis del tiempo de ejecución de cada algoritmo.

4.2.1. Datos usados en los experimentos

Respecto a los datos, como ya hemos comentado en otras secciones, hemos utilizado el *dataset* MIND (más detalles 2.3.1). Pero hay dos puntos a destacar de cara a hacer experimentos: el primero la partición de los datos utilizados para entrenamiento y para test (que viene hecha por defecto) y el segundo es que existen diferentes versiones de este *dataset*, en función de su tamaño.

Respecto al primer punto, como hemos dicho los datos ya vienen divididos. En realidad vienen divididos en tres particiones, entrenamiento, validación y test, pero por desgracia este último conjunto de datos no es público, sino que es el utilizado en la competición de MIND de manera privada para evaluar a los participantes. Por lo que nos centraremos en entrenamiento y validación (utilizando este segundo para evaluar nuestros algoritmos). Para los datos de entrenamiento fueron utilizadas las cuatro primeras semanas de toma de muestras y para los datos de validación se reservó el último día de cada una de esas semanas.

Nº Muestras	train_news	valid_news	train_behaviors	valid_behaviors
demo	26.740	18.723	21.480	7.335
small	51.282	42.416	153.727	70.938
large	101.527	72.023	2.186.683	365.201

Tabla 4.2: Tabla resumen tamaño *dataset*.

Respecto al tamaño, como ya hemos dicho, existen tres versiones de este *dataset*, “demo”, “small” y “large”, de más pequeño a más grande, y teniendo en cuenta que, como ya se ha explicado en el apartado anterior, nos encontrábamos bastante limitados de recursos de cómputo y tras hacer algunas pruebas y ver que los tiempos de entrenamiento del *dataset* “large” superaban las 24 horas por época, decidimos quedarnos con la versión mediana, es decir, “small”. En la tabla 4.2 podemos ver en detalle

el tamaño de cada una de las versiones, cabe destacar los importantes saltos de magnitud, sobretodo si nos fijamos en las columnas de “comportamientos” (“behaviors”).

4.2.2. Recomendación basada en técnicas clásicas

En este apartado describiremos las pruebas realizadas con los algoritmos de recomendación explicados en las secciones 2.1.1 y 3.3, las cuales se han centrado en comparar el funcionamiento de cada uno de ellos, modificando funciones de distancia y utilizando o no el algoritmo *BERT* como preprocesado de los datos. Aunque respecto a este hay que hacer una apreciación, debido a lo costoso computacionalmente que es este algoritmo, hemos tenido que conformarnos con evaluarlo sobre una muestra aleatoria muy reducida, pero que ya supuso un entrenamiento bastante largo.

Para realizar esta comparación hemos utilizado las funciones de evaluación propias descritas en el capítulo de implementación en conjunto con las métricas de la librería de *recommenders*, por lo tanto, las métricas utilizadas son las descritas en la sección 2.1.3: *AUC*, *MRR*, *nDCG@5* y *nDCG@10*. Los resultados obtenidos son los siguientes:

- En la tabla 4.3 podemos ver los resultados que ha obtenido el algoritmo de ponderación basado en frecuencias, es decir, el peso de cada palabra en el vector de representación es el número de veces que aparece esa palabra en el texto (ecuación 2.3). La tabla se encuentra organizada en dos partes la parte izquierda sin utilizar *BERT*, a la derecha utilizándolo y para cada una de ellas tenemos los resultados de las diferentes métricas a medida que variamos la función de distancia a utilizar. Los comentarios que podríamos hacer al respecto sería la estabilidad general, ya que los resultados están bastante bien por lo general, pero ocurre algo bastante curioso: para la métrica *AUC* la ganadora es la similitud del coseno sin utilizar *BERT*, mientras que para el resto de métricas los que destacan son los algoritmos que utilizan *BERT*, en concreto la distancia euclídea y la distancia de Manhattan.
- Por otro lado, en la tabla 4.4 podemos ver los resultados obtenidos por el algoritmo *Term frequency - Inverse document frequency (TF-IDF)*, es decir, similar al anterior solo que perjudicando a las palabras que aparecen en muchos sitios, ya que aportan menos información. con una organización idéntica a la anterior (ecuación 2.4c). En cuanto a los mejores, podríamos decir que son muy similares a los de la tabla anterior, para *AUC* sin *BERT* y para el resto con él.

Algoritmo Frecuencia	Sin <i>BERT</i> (Dataset completo)			Con <i>BERT</i> (500 muestras)		
	Sim. coseno	Euclídea	Manhattan	Sim. coseno	Euclídea	Manhattan
AUC	0,5015	0,4977	0,4976	0,4856	0,4861	0,4902
MRR	0,2190	0,2173	0,2173	0,2423	0,2756	0,2773
nDCG@5	0,2237	0,2222	0,2220	0,2259	0,2923	0,2995
nDCG@10	0,2866	0,2850	0,2852	0,3325	0,3304	0,3315

Tabla 4.3: Tabla resumen de los resultados del algoritmo basado en frecuencias.

Con estos comportamientos lo cierto es que no estaría muy claro qué algoritmo sería el mejor, ya que dependerá de cuál sea la métrica que queramos optimizar, es decir, de cuál sea nuestro objetivo. Por ejemplo, si nuestro objetivo es conseguir que aparezca una noticia relevante lo más arriba posible en el ranking (sin importarnos las demás), nos tendremos que fijar en *MRR*, o si al contrario queremos estimar la probabilidad de que un elemento relevante se encuentre detrás de uno no relevante entonces

elegiremos el algoritmo según *AUC*, ya que cuando todas las noticias relevantes aparecen al principio esta métrica vale 1. Una situación intermedia la mide *nDCG*, premiando que las noticias relevantes aparezcan todas lo más arriba posible. Ante esta variabilidad, hemos decidido fijarnos en la que es más habitual en los artículos revisados de la literatura, que para recomendación de noticias suele ser *AUC*. Por ello, según esta métrica es mejor no usar *BERT*, ya que siempre empeora los resultados, además de los ya mencionados problemas de costo computacional que este tiene.

Algoritmo TF-IDF	Sin <i>BERT</i> (Dataset completo)			Con <i>BERT</i> (500 muestras)		
	Sim. coseno	Euclídea	Manhattan	Sim. coseno	Euclídea	Manhattan
AUC	0,5005	0,5004	0,4977	0,4743	0,4776	0,4997
MRR	0,2191	0,2191	0,2175	0,2606	0,2619	0,2729
nDCG@5	0,2238	0,2238	0,2223	0,2493	0,2647	0,2997
nDCG@10	0,2865	0,2865	0,2853	0,3397	0,3407	0,3392

Tabla 4.4: Tabla resumen de los resultados resultados del algoritmo *TF-IDF*.

4.2.3. Recomendación basada en aprendizaje profundo

De cara a los experimentos realizados sobre los algoritmos de aprendizaje profundo, al ser solamente cuatro hemos podido hacer, por un lado, un análisis de la evolución del entrenamiento de cada uno, y por otro lado hemos recopilado cuales han sido los mejores resultados que hemos obtenido con cada uno, detallando las diferencias que hemos observado respecto al artículo de Microsoft [21]. Antes de empezar, cabe destacar una distinción entre los algoritmos de la librería *recommenders* (NRMS, NAML y LSTUR) y el algoritmo propio utilizando transformadas de Fourier (NRMS+FNet), ya que a lo largo de este apartado nos referiremos a ellos utilizando estos grupos.

Respecto a la evolución del entrenamiento, disponible en la figura 4.1, podemos destacar los siguientes comportamientos:

- Podemos observar que no todos los algoritmos tienen el mismo número de épocas de entrenamiento, esto es debido, en el caso de los algoritmos de *recommenders*, a un límite de horas de entrenamiento, que fue de 4 horas, al ver que ninguno de los algoritmos parecía ser capaz de mejorar mucho más. Mientras que en el caso de *FNet*, fue debido a que, no es que no aprendiera, sino que cada época que avanzaba iba a empeorando algo más.
- Por otro lado es curioso ver cómo estos algoritmos, incluso antes de entrenar, ya tienen puntuaciones bastante altas en todas las métricas, esto probablemente sea debido a que hacen uso de *embeddings* preentrenados (más información en 2.2.2).
- Relacionado con el punto anterior, lo cierto es que no se produce una mejora muy grande, a pesar del costoso entrenamiento que requieren, esto es un comportamiento que puede ser muy interesante para según qué aplicaciones.
- Por último destacar que lo que hemos hecho para seleccionar al mejor algoritmo es guardar un *checkpoint* en cada una de las iteraciones, de esta forma hemos podido recuperar sin problemas la versión del algoritmo que mejor puntuación ha logrado.

Los mejores resultados obtenidos con cada algoritmo, como ya hemos dicho, han sido extraídos utilizando *checkpoints*, por lo que, en la tabla 4.5 podemos observar, además de las métricas de cada

algoritmo, la época de la que se ha extraído. A destacar de esta tabla algo que puede resultar muy curioso, y es que *NRMS* no destaca en ninguna métrica, los mejores algoritmos son *LSTUR* y *NAML*, al contrario de lo que ocurre en el artículo de Microsoft [21], en el que presentan *NRMS* como el mejor algoritmo. Esto probablemente se deba a que las redes que utilizan atención necesitan muchos datos para aprender correctamente, y en esta ocasión precisamente utilizamos menos datos que el artículo y *NRMS* está basado completamente en atención. A pesar de esto también cabe destacar que *NRMS* es el algoritmo (de los de *recommenders*) que menos épocas necesita para entrenar.

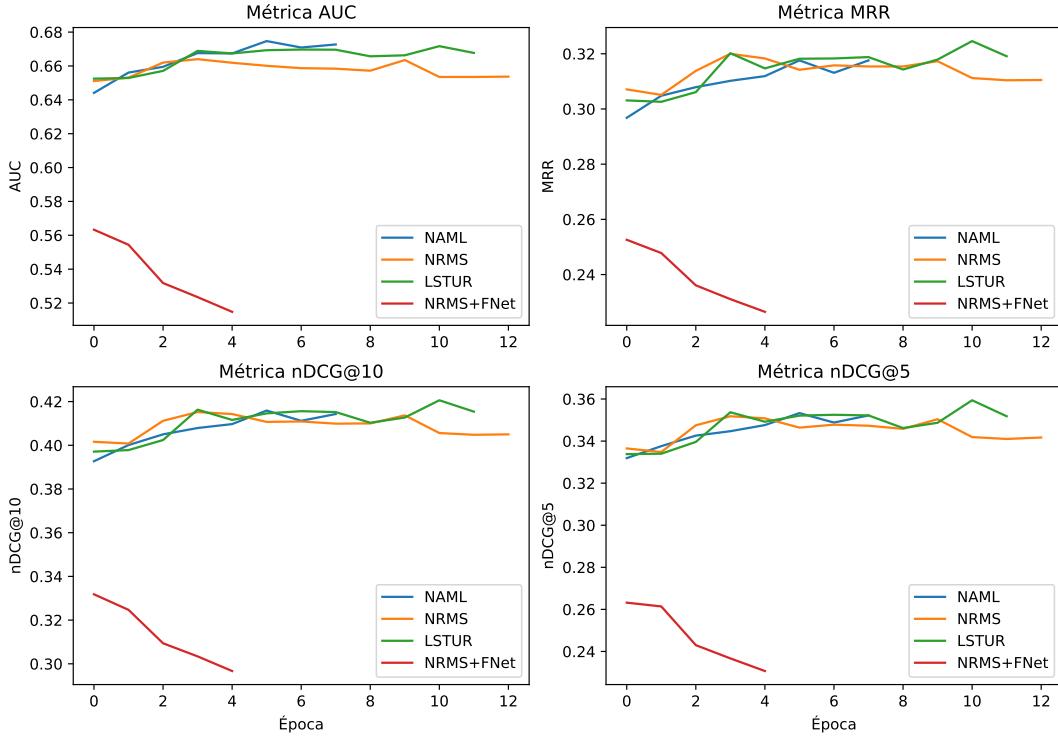


Figura 4.1: Evolución del entrenamiento de algoritmos basados en aprendizaje profundo.

Algoritmo	Época	AUC	MRR	nDCG@10	nDCG@5
NAML	6	0,6747	0,3176	0,4159	0,3533
NRMS	4	0,6640	0,3200	0,4152	0,3518
LSTUR	11	0,6717	0,3246	0,4206	0,3594
NRMS+FNet	1	0,5633	0,2526	0,3318	0,2632

Tabla 4.5: Tabla de mejores resultados de algoritmos basados en aprendizaje profundo.

4.2.4. Resumen y análisis de coste

Por último realizaremos un pequeño análisis del tiempo consumido por cada uno de los algoritmos planteados, teniendo en cuenta también cuál ha sido el rendimiento de ese algoritmo finalmente. Antes de nada destacar que se han omitido las versiones que utilizaban *BERT* debido a que sería una comparación injusta al haber sido entrenadas con una cantidad de datos mucho menor.

En cuanto al tiempo, hemos analizado por un lado el tiempo de evaluación del algoritmo, esto es, el tiempo que se tarda en evaluar el algoritmo sobre la partición de test del dataset completo (versión *small*). Por otro lado está el tiempo de entrenamiento, que es el tiempo que invierte el algoritmo en

aprender todo lo que puede de la partición de entrenamiento, también del *dataset* completo, respecto a este tiempo cabe destacar que solo se ha tenido en cuenta el tiempo necesario para llegar al mejor modelo, es decir, solamente el tiempo de entrenamiento hasta las épocas mostradas en la tabla 4.5.

Los resultados están disponibles en la tabla 4.6, en la que, lo primero que nos puede llamar la atención es el tiempo de entrenamiento de los modelos basados en técnicas clásicas, el cual es prácticamente despreciable. Esto es así debido a que su entrenamiento consiste simplemente en la construcción del diccionario que necesitan para construir las representaciones de cada noticia (más información 2.1.1), por otro lado la evaluación de estos algoritmos consiste en el cálculo de miles de distancias, donde Manhattan sale perdiendo, ya que para poder calcularla se ve obligado a convertir las matrices dispersas en canónicas, cosa que lleva bastante tiempo. Otro punto a tener en cuenta es el otro extremo, el tiempo de entrenamiento de los modelos basados en aprendizaje profundo, que requieren varias horas de computación en GPU.

Cabe destacar además lo sorprendentemente eficiente que es la versión de *NRMS* que utiliza transformadas de Fourier en lugar de atención (*NRMS+FNet*), con un entrenamiento de escasos 10 minutos, y el tiempo de evaluación más bajo de la tabla, obtiene unos resultados más que aceptables, varios puntos por encima que los algoritmos más clásicos. Este es un comportamiento que podríamos esperar, ya que más o menos se cumple lo descrito en el artículo [13]. A pesar de esto, a este algoritmo le queda mucho por mejorar, solo hace falta ver el pésimo comportamiento que tiene al entrenar.

Intentar extraer un ganador de esta tabla es una tarea realmente complicada, teniendo en cuenta los tres grupos que hay quizá podríamos decir que, del apartado de algoritmos clásicos, el más equilibrado sería el basado en frecuencias utilizando la similitud del coseno. Del segundo grupo no está tan claro, cada uno es bueno en una cosa, *NAML* el que mejores resultados consigue, *NRSM* el que menos tarda en entrenar y *LSTUR* el que menos tarda en evaluar.

Para el uso que les vamos a dar en este trabajo lo más interesante sería escoger los algoritmos que más rápido realicen la evaluación y mejor la hagan, sin fijarnos demasiado en el tiempo de entrenamiento, ya que este solo se va a realizar una vez. Por lo que, teniendo esto en cuenta probablemente la mejor elección de cada grupo serían *LSTUR*, con *NRMS+FNet* y por último con el algoritmo basado en frecuencias utilizando la similitud del coseno, ordenados de más favorable a menos.

Algoritmo	Tiempo de evaluación	Tiempo de entrenamiento	AUC
TF-IDF Sim. coseno	0:10:10	0:0:01	0,5005
TF-IDF Euclídea	0:09:58	0:0:01	0,5004
TF-IDF Manhattan	0:29:09	0:0:01	0,4977
Frecuencia Sim. coseno	0:09:38	0:0:01	0,5015
Frecuencia Euclídea	0:09:27	0:0:01	0,4977
Frecuencia Manhattan	0:29:10	0:0:01	0,4976
NAML	0:06:31	2:59:06	0,6747
NRMS	0:02:34	1:21:29	0,6640
LSTUR	0:02:25	3:44:57	0,6717
NRMS+FNet	0:01:40	0:10:45	0,5633

Tabla 4.6: Tabla comparativa tiempos de computación.

CONCLUSIONES Y TRABAJO FUTURO

5.1. Conclusiones

Los dos principales pilares de este Trabajo de Fin de Grado han sido, por un lado, la investigación, donde hemos realizado una revisión de algunos de los algoritmos de recomendación de noticias más utilizados en la actualidad, centrándonos en los más complejos basados en aprendizaje profundo, pero sin olvidar los más sencillos, basados en técnicas más clásicas. El otro pilar ha sido la parte de desarrollo, implementando una aplicación web en un entorno cliente-servidor con una *API REST*, con la que mostrar cuál ha sido el procedimiento para lograr tener uno de estos algoritmos en un entorno en producción.

Si nos centramos en el análisis de los algoritmos, podemos extraer diferentes conclusiones. En primer lugar, hemos visto que los métodos basados en técnicas clásicas tienen sus ventajas, a pesar de no funcionar tan bien como los algoritmos más complejos. Respecto a estos segundos, su eficacia no tiene, indudablemente, rival alguno por el momento, pero también es cierto que requieren una capacidad computacional muy a tener en cuenta. Por último, cabe destacar que están empezando a aparecer algunas líneas de investigación que precisamente intentan buscar una solución intermedia: un algoritmo con una eficacia aceptable y con una buena eficiencia computacional. Sin duda es un campo en el que aún queda mucho estudio por delante, como veremos en la siguiente sección.

Desde el punto de vista más personal, este trabajo me ha servido para aprender muchas cosas. En primer lugar para empezar a adentrarme en el curioso mundo de la investigación, algo que siempre me había llamado la atención pero que no había podido probar hasta ahora. Además, me gustaría destacar una de mis principales inquietudes a la hora de realizar este trabajo, y es intentar que este se encuentre lo más actualizado posible, al menos en el momento de entregarlo. Esto ha provocado numerosas dificultades, como que la librería de *recommenders* al principio no estuviera muy rematada o que a última hora se hayan tenido que añadir algunos algoritmos como el basado en transformadas de Fourier, pero sin duda considero que ha merecido la pena.

Por último, querría destacar que todo el código relativo al trabajo se encuentra en la siguiente página web:

<https://ericmrls.github.io/tfg.html>

5.2. Trabajo Futuro

A continuación, detallaremos algunos de los aspectos que, tras mucha investigación, consideramos que puede ser interesante tener en cuenta de cara a próximos trabajos, pero que por falta de tiempo y/o recursos no hemos podido entrar en detalle en esta ocasión, a pesar de que nos habría gustado intentarlo.

Uno de los puntos que sin duda sería muy interesante investigar es la combinación de redes *LSTM* y técnicas de atención. Esto es algo que ninguno de los artículos investigados realiza y que podría dar muy buenos resultados; de hecho en [21] se menciona como algo con mucho potencial. Esto es así debido a la gran capacidad de las redes *LSTM* para aprender los intereses a corto y largo plazo, que podríamos utilizar para construir la representación del usuario, y por otro lado tener la capacidad de las técnicas de atención en el campo del procesamiento de lenguaje natural, que podríamos usar para construir la representación de las noticias.

Otra de las implementaciones que podría obtener también muy buenos resultados sería, en lugar de utilizar las redes de atención propuestas en los artículos mencionados, crear una capa de atención basada en *Transformer* (más detalle en 2.2.3), o al menos una estructura basada en un *encoder-decoder*, no exclusivamente capas de atención. Esto podría tener un gran impacto a la hora de construir la representación de las noticias.

Por otro lado, a pesar de que no se le ha hecho mucho hincapié, todos los algoritmos basados en aprendizaje profundo requieren como entrada unos vectores de *embedding*. En esta ocasión Microsoft propone utilizar los vectores creados por *Glove*¹, pero sería interesante probar con otros *embeddings*, incluso, en lugar de utilizar uno de carácter general, entrenar unos de manera que se especialicen en el ámbito de la recomendación de noticias.

Además, podría ser interesante continuar investigando en el apartado de utilizar transformadas de Fourier en lugar de atención, ya que en este trabajo, debido a la fecha en la que se publicó ese descubrimiento, fue muy complicado conseguir una versión de este tipo de algoritmos que funcionara del todo bien. Pero ya solo con los resultados preliminares obtenidos podemos ver cuál puede ser el potencial de una aproximación de este estilo.

Por último, un campo en el que no hemos podido entrar ya que no teníamos datos suficientes, pero que sin duda puede ser clave en este tipo de problemas, es el tener en cuenta las imágenes de las noticias (probablemente también usando aprendizaje profundo). Ello se propone porque un usuario, a la hora de dar clic o no a una noticia, es probable que se fije más en la imagen que acompaña a la misma que al pequeño resumen que aparece debajo. Por ello creemos que puede ser una muy buena forma de complementar a los algoritmos basados en el texto, como los que hemos analizado a lo largo de este trabajo.

¹ *Glove, Global Vectors for Word Representation*, accessed 2021-04-15. <https://nlp.stanford.edu/projects/glove/>

BIBLIOGRAFÍA

- [1] M. De Gemmis, P. Lops, C. Musto, F. Narducci, and G. Semeraro, *Semantics-aware content-based recommender systems*, pp. 119–159. Springer, 2015.
- [2] M. Sahlgren, “An introduction to random indexing,” in *Methods and applications of semantic indexing workshop at the 7th international conference on terminology and knowledge engineering*, 2005.
- [3] Y. Koren and R. M. Bell, “Advances in collaborative filtering,” in *Recommender Systems Handbook* (F. Ricci, L. Rokach, and B. Shapira, eds.), pp. 77–118, Springer, 2015.
- [4] X. Ning, C. Desrosiers, and G. Karypis, “A comprehensive survey of neighborhood-based recommendation methods,” in *Recommender Systems Handbook* (F. Ricci, L. Rokach, and B. Shapira, eds.), pp. 37–76, Springer, 2015.
- [5] Y. Koren, “Factorization meets the neighborhood: a multifaceted collaborative filtering model,” in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008* (Y. Li, B. Liu, and S. Sarawagi, eds.), pp. 426–434, ACM, 2008.
- [6] A. Gunawardana and G. Shani, “Evaluating recommender systems,” in *Recommender systems handbook*, pp. 265–308, Springer, 2015.
- [7] C. de la Lengua Española Sevilla 1992, “Lenguas y tecnologías de la información,” 1994.
- [8] T. Bolukbasi, K. Chang, J. Y. Zou, V. Saligrama, and A. T. Kalai, “Man is to computer programmer as woman is to homemaker? debiasing word embeddings,” in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain* (D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, eds.), pp. 4349–4357, 2016.
- [9] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2013.
- [10] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States* (C. J. C. Burges, L. Bottou, Z. Ghahramani, and K. Q. Weinberger, eds.), pp. 3111–3119, 2013.
- [11] X. Rong, “word2vec parameter learning explained,” *CoRR*, vol. abs/1411.2738, 2014.
- [12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, eds.), pp. 5998–6008, 2017.

- nathan, and R. Garnett, eds.), pp. 5998–6008, 2017.
- [13] J. Lee-Thorp, J. Ainslie, I. Eckstein, and S. Ontañón, “Fnet: Mixing tokens with fourier transforms,” *CoRR*, vol. abs/2105.03824, 2021.
- [14] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)* (J. Burstein, C. Doran, and T. Solorio, eds.), pp. 4171–4186, Association for Computational Linguistics, 2019.
- [15] 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,” in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), 2020.
- [16] S. Okura, Y. Tagami, S. Ono, and A. Tajima, “Embedding-based news recommendation for millions of users,” in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pp. 1933–1942, ACM, 2017.
- [17] C. Wu, F. Wu, M. An, J. Huang, Y. Huang, and X. Xie, “NPA: neural news recommendation with personalized attention,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019* (A. Tredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, eds.), pp. 2576–2584, ACM, 2019.
- [18] A. Das, M. Datar, A. Garg, and S. Rajaram, “Google news personalization: scalable online collaborative filtering,” in *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007* (C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, eds.), pp. 271–280, ACM, 2007.
- [19] M. Kompan and M. Bieliková, “Content-based news recommendation,” in *E-Commerce and Web Technologies, 11th International Conference, EC-Web 2010, Bilbao, Spain, September 1-3, 2010. Proceedings* (F. Buccafurri and G. Semeraro, eds.), vol. 61 of *Lecture Notes in Business Information Processing*, pp. 61–72, Springer, 2010.
- [20] I. Ilievski and S. Roy, “Personalized news recommendation based on implicit feedback,” in *Proceedings of the 2013 International News Recommender Systems Workshop and Challenge, NRS '13*, (New York, NY, USA), p. 10–15, Association for Computing Machinery, 2013.
- [21] F. Wu, Y. Qiao, J. Chen, C. Wu, T. Qi, J. Lian, D. Liu, X. Xie, J. Gao, W. Wu, and M. Zhou, “MIND: A large-scale dataset for news recommendation,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020* (D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, eds.), pp. 3597–3606, Association for Computational Linguistics, 2020.
- [22] J. Liu, P. Dolan, and E. R. Pedersen, “Personalized news recommendation based on click behavior,”

- in *Proceedings of the 15th International Conference on Intelligent User Interfaces, IUI 2010, Hong Kong, China, February 7-10, 2010* (C. Rich, Q. Yang, M. Cavazza, and M. X. Zhou, eds.), pp. 31–40, ACM, 2010.
- [23] F. V. Jensen, *Bayesian Networks and Decision Graphs*. Statistics for Engineering and Information Science, Springer, 2001.
- [24] L. Li, L. Zheng, F. Yang, and T. Li, “Modeling and broadening temporal user interest in personalized news recommendation,” *Expert Syst. Appl.*, vol. 41, no. 7, pp. 3168–3177, 2014.
- [25] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, 2003.
- [26] H. Wang, F. Zhang, X. Xie, and M. Guo, “DKN: deep knowledge-aware network for news recommendation,” in *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018* (P. Champin, F. Gandon, M. Lalmas, and P. G. Ipeirotis, eds.), pp. 1835–1844, ACM, 2018.
- [27] Y. LeCun, Y. Bengio, and G. E. Hinton, “Deep learning,” *Nat.*, vol. 521, no. 7553, pp. 436–444, 2015.
- [28] A. Argyriou, M. González-Fierro, and L. Zhang, “Microsoft recommenders: Best practices for production-ready recommendation systems,” in *Companion Proceedings of the Web Conference 2020*, pp. 50–51, 2020.
- [29] C. Wu, F. Wu, S. Ge, T. Qi, Y. Huang, and X. Xie, “Neural news recommendation with multi-head self-attention,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019* (K. Inui, J. Jiang, V. Ng, and X. Wan, eds.), pp. 6388–6393, Association for Computational Linguistics, 2019.
- [30] C. Wu, F. Wu, M. An, J. Huang, Y. Huang, and X. Xie, “Neural news recommendation with attentive multi-view learning,” in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019* (S. Kraus, ed.), pp. 3863–3869, ijcai.org, 2019.
- [31] M. An, F. Wu, C. Wu, K. Zhang, Z. Liu, and X. Xie, “Neural news recommendation with long- and short-term user representations,” in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers* (A. Korhonen, D. R. Traum, and L. Màrquez, eds.), pp. 336–345, Association for Computational Linguistics, 2019.
- [32] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, “Devops,” *IEEE Softw.*, vol. 33, no. 3, pp. 94–100, 2016.
- [33] S. Mäkinen, H. Skogström, E. Laaksonen, and T. Mikkonen, “Who needs mlops: What data scientists seek to accomplish and how can mlops help?,” *CoRR*, vol. abs/2103.08942, 2021.
- [34] S. Craw, “Manhattan distance,” in *Encyclopedia of Machine Learning* (C. Sammut and G. I. Webb, eds.), (Boston, MA), pp. 639–639, Springer US, 2010.

DEFINICIONES

Distancia de Manhattan La distancia de Manhattan entre dos puntos $x = (x_1, x_2, \dots, x_n)$ e $y = (y_1, y_2, \dots, y_n)$ en un espacio de n dimensiones, es la suma de las distancias en cada dimensión: $d(x, y) = \sum_{i=1}^n |x_i - y_i|$ [34].

Distancia euclídea La distancia euclídea entre dos puntos $x = (x_1, x_2, \dots, x_n)$ e $y = (y_1, y_2, \dots, y_n)$ en un espacio de n dimensiones, es la raíz cuadrada de la suma de las distancias en cada dimensión al cuadrado: $d(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$.

Similitud coseno La similitud coseno de dos vectores $x = (x_1, x_2, \dots, x_n)$ e $y = (y_1, y_2, \dots, y_n)$ en un espacio de n dimensiones, se define como el coseno del ángulo que forman estos dos vectores: $\cos(\vec{x}, \vec{y}) = \cos \frac{\vec{x} \cdot \vec{y}}{|\vec{x}| |\vec{y}|}$ [4].

ACRÓNIMOS

AJAX Asynchronous JavaScript And XML.

AWS Amazon Web Services.

BERT Bidirectional Encoder Representations from Transformers.

GPT Generative Pre-training Transformer.

LSTUR Neural News Recommendation with Long- and Short-term User Representations.

MIND Microsoft News Dataset.

NAML Neural News Recommendation with Attentive Multi-View Learning.

nDCG normalized Discounted Cumulated Gain.

NLP Natural Language Processing.

NRMS Neural News Recommendation with Multi-Head Self-Attention.

RR Reciprocal Rank.

TFG Trabajo de Fin de Grado.

TF-IDF Term frequency - Inverse document frequency.

APÉNDICES

PÁGINA WEB

[Recomendar](#)

[Inicio](#) [Recomendar](#)

Bienvenido

Página web demostración del Trabajo de Fin de Grado de Eric Morales Agostinho.

Se trata de una página web con un backend programado en Python utilizando el framework Flask.

El esqueleto HTML/CSS esta basado en la plantilla [Blog de Bootstrap](#).

sports

The Eagles' reliance on DeSean Jackson is a problem, and there are a few people to blame

Nov 11

PHILADELPHIA -- After the Eagles lost to the Falcons in Atlanta on Sept. 15, DeSean Jackson lay on a trainer's table at Mercedes-Benz Stadium and called his personal trainer. Jackson had strained his abdomen, which had already been bothering him before the game, but neither he or the Eagles trainers thought the injury was serious. Jackson eschewed surgery with the hope that he'd return in time ...

[Continue reading](#)



news

AP sources: State Dept. worried about defending ambassador

Nov 11

The State Department's third-ranking official is expected to tell Congress that political considerations were behind the agency's refusal to deliver a robust defense of the former U.S. ambassador to Ukraine. People familiar with the matter say the highest-ranking career diplomat in the foreign service, David Hale, plans to tell congressional impeachment investigators on Wednesday that Secretary of State Mike Pompeo and other...

[Continue reading](#)



autos

Marine Corps Marathon to shut down roads in D.C. and Arlington this Sunday

Nov 11

The race will take over much of the National Mall and the Potomac riverfront

[Continue reading](#)



sports

Kenyans Geoffrey Kamworor, Joyciline Jepkosgei win New York City Marathon

Nov 11

Kenyans swept the New York City Marathon with the top Americans finishing sixth in each race.

[Continue reading](#)



sports



sports



Figura A.1: Página principal de la aplicación web.

Recommender

[Inicio](#) [Recomendar](#)



Recomendación

Completando la información de la derecha se podrá recibir una recomendación personalizada.

En la parte inferior se puede ver las noticias que previamente ha visto el usuario y debajo ordenadas se mostrarán las noticias recomendadas.

Usuario:

Timestamp:

Algoritmo 1:

Algoritmo 2:

Noticias visitadas

news

President Donald Trump greeted with boos at Game 5 of World Series, fans chant 'lock him up'

Nov 11

President Donald Trump was greeted with a thunderous chorus of boos from the sold-out crowd of baseball fans at National[...]

[Continue reading](#)

news

Mulvaney allies to try to stonewall Democrats' impeachment inquiry, officials say

Nov 11

The budget chief and other top aides will attempt to create firewall after other senior officials gave testimony that qu[...]

[Continue reading](#)

Noticias candidatas

Algoritmo NRMS

news



news

Algoritmo LSTUR

news



news

Figura A.2: Cabecera del apartado de recomendación de la aplicación web.

Noticias visitadas

news

President Donald Trump greeted with boos at Game 5 of World Series, fans chant 'lock him up'

Nov 11

President Donald Trump was greeted with a thunderous chorus of boos from the sold-out crowd of baseball fans at National[...]

[Continue reading](#)



news

Mulvaney allies to try to stonewall Democrats' impeachment inquiry, officials say

Nov 11

The budget chief and other top aides will attempt to create firewall after other senior officials gave testimony that qu[...]

[Continue reading](#)



Noticias candidatas

Algoritmo NRMS

news

Trump asks Supreme Court to shield his tax returns from prosecutors, setting up historic separation-of-powers showdown

Nov 11

The president's personal lawyers have asserted broad immunity claims in trying to block investigations into his business records.

[Continue reading](#)



Algoritmo LSTUR

news

Giuliani Faces U.S. Probe on Campaign Finance, Lobbying Breaches

Nov 11

Rudy Giuliani, President Donald Trump's personal lawyer, is being investigated by federal prosecutors for possible campaign finance violations and a failure to register as a foreign agent as part of an active investigation into his financial dealings, according to three U.S. officials.

[Continue reading](#)



finance

Jack Ma Says U.S.-China Trade Tension Could Last 20 Years

Nov 11

Jack Ma, the co-founder and former chairman of Alibaba Group Holding Ltd., said the U.S.-China relationship could face 20 years of "turbulence" if the two superpowers aren't careful in how they handle trade.

[Continue reading](#)



finance

Jack Ma Says U.S.-China Trade Tension Could Last 20 Years

Nov 11

Jack Ma, the co-founder and former chairman of Alibaba Group Holding Ltd., said the U.S.-China relationship could face 20 years of "turbulence" if the two superpowers aren't careful in how they handle trade.

[Continue reading](#)



lifestyle

Meghan Markle and Hillary Clinton Secretly Spent the Afternoon Together at Frogmore Cottage

Nov 11

...
...
...



news

Trump asks Supreme Court to shield his tax returns from prosecutors, setting up historic separation-of-powers showdown



Figura A.3: Cuerpo del apartado de recomendación de la aplicación web.

ARQUITECTURAS DE MODELOS

B.1. Word2Vec

Figuras B.1 y B.2.

B.2. Recomendación de noticias

Figuras B.3-B.5

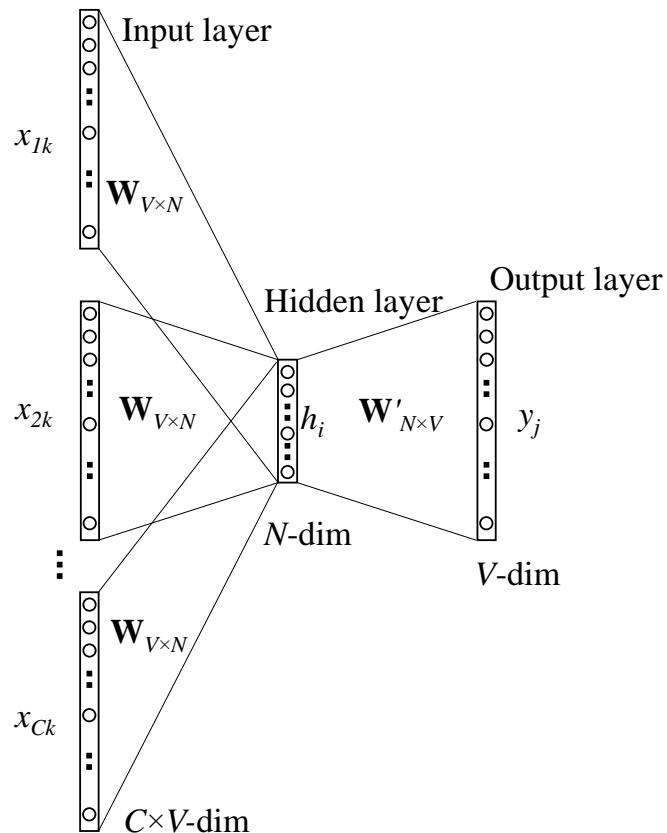


Figura B.1: Arquitectura modelo Continuous Bag-of-Words (CBOW) [11].

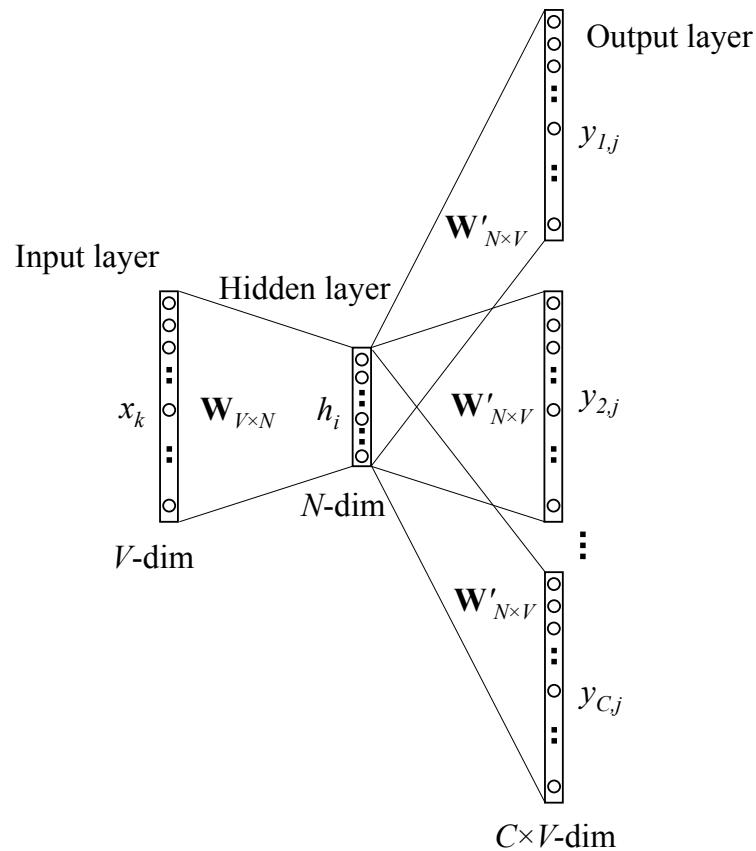


Figura B.2: Arquitectura modelo Skip-gram [11].

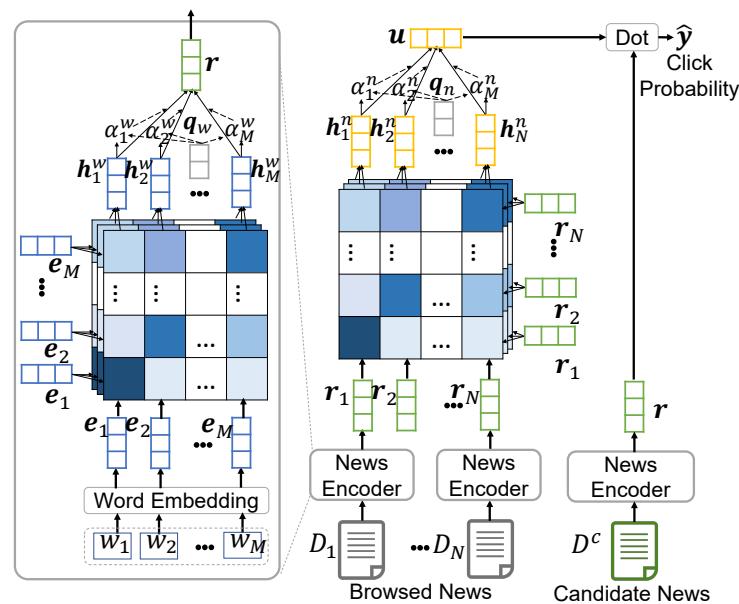


Figura B.3: Arquitectura modelo NRMS [29].

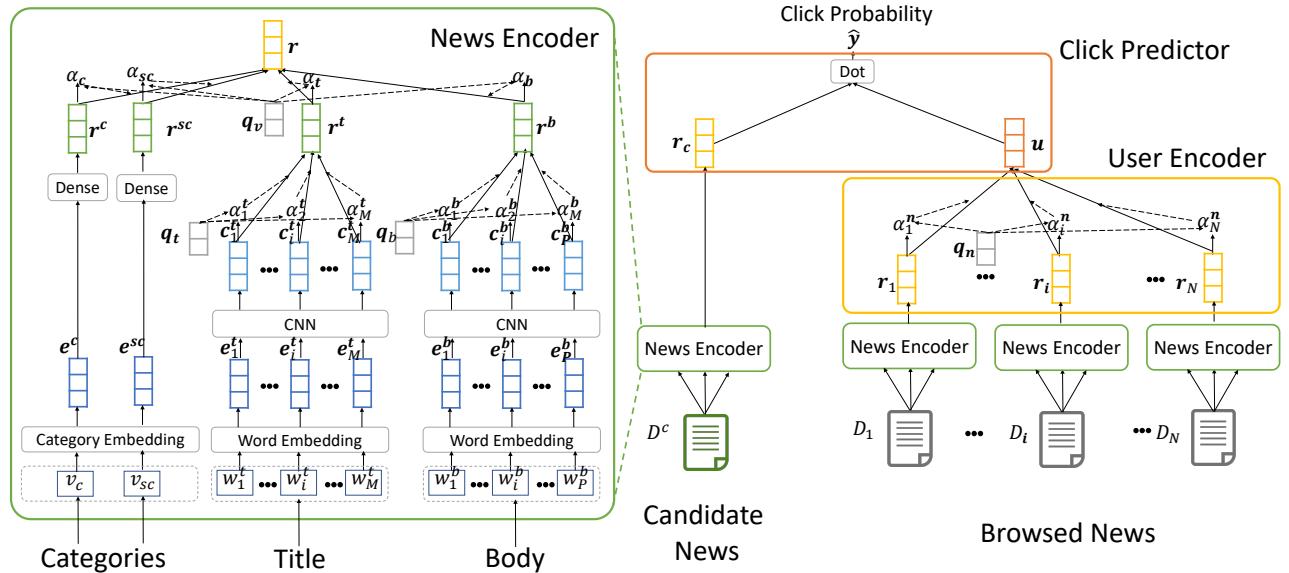


Figura B.4: Arquitectura modelo NAML [30].

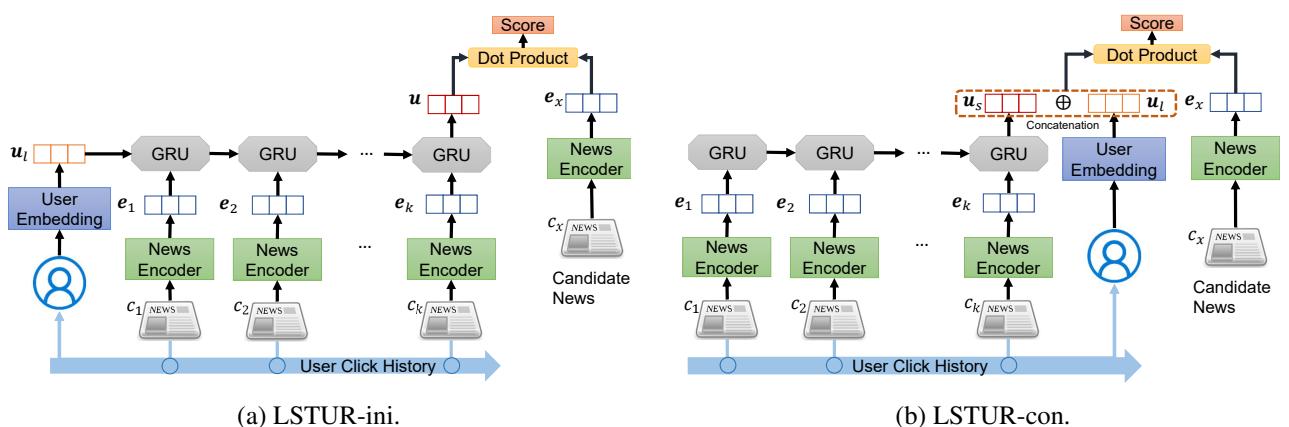


Figura B.5: Arquitectura modelo LSTUR [31].

