Graduado en Ingeniería del Software

Desarrollo de un buscador inteligente de películas mediante búsqueda híbrida

Development of an intelligent movie search engine through hybrid search

Realizado por

Pablo Márquez Benítez

Tutorizado por
Eduardo Guzmán de los Riscos

Departamento
Lenguajes y Ciencias de la Computación

MÁLAGA, junio de 2025

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA DEL SOFTWARE

Desarrollo de un buscador inteligente de películas mediante búsqueda híbrida

Development of an intelligent movie search engine through hybrid search

Realizado por
Pablo Márquez Benítez

Tutorizado por
Eduardo Guzmán de los Riscos

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2025

Fecha defensa: julio de 2025

Resumen

Este Trabajo de Fin de Grado se centra en el estudio del funcionamiento de los sistemas de recuperación de la información y cómo las arquitecturas híbridas pueden mejorar la calidad de estos sistemas. Para ello se toma como caso de uso el diseño, desarrollo e implementación de un motor de búsqueda inteligente de películas, ofreciendo una experiencia de búsqueda novedosa en el ámbito y una herramienta diseñada por y para los usuarios, que se adapta a sus necesidades y les ayuda a descubrir nuevo contenido.

Se ha realizado un desarrollo de motor de búsqueda completo, desde la recopilación, limpieza e indexación de los datos, hasta la mejora de diferentes componentes de búsqueda y decisiones de diseño, a partir de evaluaciones simuladas de forma inteligente y con usuarios reales, buscando ofrecer la mejor experiencia a los usuarios y gestionando los recursos de forma eficiente.

El motor de búsqueda desarrollado permite a los usuarios tanto realizar búsquedas de contenido exacto por el título, director, actores, etc. hasta realizar descubrimiento de películas nuevas a partir de la descripción de temáticas, conceptos abstractos, etc. Todo en una misma búsqueda. Además se le han incorporado funcionalidades como un corrector personalizado para el dominio de las películas, con objetivo de ayudar a los usuarios a evitar fallos en sus búsquedas. También se ha integrado el uso de inteligencia artificial generativa para ofrecer a los usuarios una reseña general de los resultados obtenidos y reseñas concretas de las películas que deseen, en caso de que tengan dudas de si les interesa o no verlas. Con esto último esta herramienta se convierte en un RAG, uno de los sistemas de recuperación de información más revolucionarios en los tiempos recientes.

Palabras clave: Recuperación de Información, Búsqueda Híbrida, Búsqueda Semántica, IA Generativa, Modelos de Lenguaje Grande

Abstract

This Final Project studies how information retrieval works and how hybrid architectures can enchance the quality of these systems. For that, the use case of design, development and implementation of an intelligent search engine is taken, offering a novel search experience in the field and a tool designed by and for users, that adapts to their needs and help them to discover new content.

A complete development of a search engine has been carried out, from the stage of data collection, cleanse and indexation to the improvement of different search components and design decisions through intelligent simulated evaluations and real user evaluations. Always looking to offer the best experience to users and mananging the resources efficiently.

The developed search engine allows user both to search for exact content by title, director, actors, ect. and to discover films by its themes, abstract concepts, etc. all in one search. Also different functionalities have been integrated like a spell checker personalized for the movies domain, to help users avoid misspelled on their searches. In addition to that generative artificial intelligence has been integrated too, providing users withgeneral reviews of the search results and specific reviews to movies they want, in order to decide whether they want to see them or not. With this last feature, this tool becomes a RAG, one of the most revolutionary information retrieval systems in recent times.

Keywords: Information Retrieval, Hybrid Search, Semantic Search, Generative AI, Large Language Models

Agradecimientos

Agradecer a mi tutor Eduardo, por la implicación y los buenos consejos que ha ofrecido, tanto durante el desarrollo de este proyecto como a nivel académico.

Quisiera dar también las gracias a mis amigos, a *Taco Team*, y en particular al *Equipo APND*, con los que he compartido esta etapa universitaria, aprendiendo y compartiendo tanto los buenos como los malos momentos.

Por último a mi familia, y especialmente a mi madre, por el esfuerzo y el apoyo que ha dedicado para que hoy me encuentre aquí.

Índice

1. Introducción **9**

1.1. Motivación . . . 9

1.2. Objetivos . . . 10

1.3. Tecnologías usadas . . . 11

1.3.1. PostgreSQL . . . 11

1.3.2. Python . . . 11

1.3.3. ReactJS . . . 13

1.3.4. Tailwind CSS . . . 13

1.3.5. Cohere Rerank v3.5 . . . 13

1.3.6. Gemini Flash 2.0 . . . 13

1.4. Metodología utilizada . . . 14

1.5. Estructura del documento . . . 14

2. Marco Teórico **17**

2.1. Motores de búsqueda tradicionales . . . 17

2.1.1. Índice invertido . . . 17

2.1.2. Funcionamiento de un motor de búsqueda tradicional. . . . 18

2.1.3. Limitaciones de las búsquedas por palabras clave . . . 20

2.1.4. Búsqueda booleana 21

2.2. Búsqueda semántica 22

2.2.1. Procesamiento de lenguaje natural 22

2.2.2. Técnicas de NLP 23

2.2.3. Codificación y decodificación del contexto 31

2.2.4. Modelos transformadores 32

2.2.5. Funcionamiento de un motor de búsqueda semántico 34

2.2.6. Limitaciones de la búsqueda semántica 37

2.3. Búsqueda híbrida 38

2.3.1. Técnicas de fusión 38

2.3.2. Arquitecturas híbridas 42

3. Desarrollo del Buscador **43**

3.1. Obtención, visualización y tratamiento de los datos 43

3.2. Almacenamiento y gestión de los datos 47

3.3. Evaluación de los buscadores y estrategias de mejora 49

3.3.1. Métricas para evaluar sistemas de IR 49

3.3.2. Diseño de las evaluaciones 54

3.4. Desarrollo de la búsqueda por palabras clave 58

3.4.1. Extracción de palabras clave 58

3.4.2. Técnicas de ranking y ponderación 59

3.4.3. Funciones de búsqueda 60

3.4.4. Indexación y optimización 61

3.4.5. Detalles de la implementación 62

3.5. Desarrollo de la búsqueda semántica 66

3.5.1. Selección del modelo y almacenamiento de embeddings 67

3.5.2. Indexación y optimización 72

3.5.3. Detalles de la implementación 75

3.6. Desarrollo de la búsqueda híbrida 80

3.6.1. Implementación de RRF 81

3.6.2. Implementación de Crossencoder 82

3.6.3. Combinación de fusiones 83

3.6.4. Detalles de la implementación 83

3.6.5. Evaluación final con usuarios reales 86

4. Desarrollo de la Aplicación Web 89

4.1. Desarrollo del Frontend 89

4.1.1. Microservicios 91

4.2. Funcionalidades adicionales de la aplicación 92

4.2.1. Corrector 92

4.2.2. RAG 94

5. Conclusiones y Líneas Futuras **97**

5.1. Conclusiones 97

5.2. Líneas Futuras 98

1 Introducción

1.1. Motivación

Los motores de búsqueda aparecen, entre otras cosas, como una herramienta para facilitar al usuario el acceso a aquellos productos que podrían desear consumir, creando un modelo de negocio beneficioso tanto para el usuario como para los vendedores. Para la búsqueda de películas, siempre han existido diversos enfoques en los buscadores en sitios como plataformas de streaming, webs de reseñas de cine entre otros. Sus buscadores generalmente están basados en palabras clave (coincidencia de título, género, actores, etc). Sin embargo, es raro encontrar buscadores que te otorguen la libertad de buscar películas usando un lenguaje más natural para describirlas, y cuando estos existen, no siempre ofrecen resultados efectivos.

Esta limitación hace que en muchos casos, los usuarios se vean frustrados por la falta de buenos resultados o directamente la ausencia de ellos cuando intentan hacer búsquedas que se salgan mínimamente de las palabras clave o cuando intentan describir películas por su argumento.

Este proyecto pretende resolver esto aprovechando las ventajas del procesamiento de lenguaje natural que hoy nos facilita la Inteligencia Artificial. Se busca ofrecer al usuario la libertad de describir el tipo de película que desea ver de forma natural, ya sea de manera abstracta o más específica. De esta forma tanto si el usuario busca usando palabras exactas, como si decide ser más creativo en la descripción, o incluso ambas cosas a la vez, podrá obtener resultados relacionados con lo que buscaba.

Es importante destacar que los principios de este buscador pueden ser fácilmente aplica-

do a otros campos como puede ser el comercio electrónico, búsquedas generales, búsquedas académicas, etc. En general cualquier tarea que conlleve la búsqueda de información sobre un volumen de datos de gran magnitud.

Esto será posible siempre y cuando se disponga de palabras clave y datos descriptivos de aquello que se busca en cuestión. No obstante, para este trabajo se ha decidido usar el cine como dominio de aplicación.

1.2. Objetivos

El objetivo principal de este TFG es diseñar, desarrollar e implementar un buscador de películas usando un mecanismo híbrido, que fusione la búsqueda por palabras clave con la búsqueda semántica, Para conseguir este objetivo se desglosan los siguientes subobjetivos:

1. **Estudio y entendimiento** del funcionamiento de los sistemas de recuperación de información.
2. **Tratamiento y visualización de los datos** de un dataset de películas.
3. **Diseño y gestión de los datos** en un gestor de base de datos relacional, con objeto de hacer búsquedas eficientes de los datos, especialmente de los vectores generados por un modelo del lenguaje grande (LLM, Large Language Model).
4. **Diseñar el flujo de trabajo** de los componentes de búsquedas.
5. **Desarrollar el software** necesario para que:
   - Dada una consulta del usuario se pueda realizar una búsqueda por palabras clave y obtener las películas con mayor coincidencia en los datos.
   - Dada una consulta del usuario se pueda computar una búsqueda semántica y obtener las películas con mayor coincidencia en los datos.
   - Fusionar las dos capas anteriores por medio de una arquitectura híbrida, técnicas de fusión y ranking.

6. **Optimización de las búsquedas**, mediante métodos como los **índices inversos** para la búsqueda por palabras clave, el uso de *(ANN)* para el caso de la búsqueda vectorial o cualquier otra técnica que se considere adecuada para cumplir con este objetivo.

7. **Evaluación del desempeño de las búsquedas** a partir de métricas de calidad obtenidas a partir de la evaluación de diferentes configuraciones de componentes de búsqueda, arquitecturas híbridas, etc. con el fin de mantener el mejor enfoque, equilibrando velocidad y precisión.

8. **Creación de una interfaz web** para poder probar las funcionalidades implementadas a nivel de usuario.

1.3. Tecnologías usadas

1.3.1. PostgreSQL

Se utilizó PostgreSQL como sistema de gestión de bases de datos relacional para almacenar la información estructurada sobre las películas. Su robustez y soporte para extensiones lo hicieron ideal para manejar grandes volúmenes de datos, y hacer búsquedas por texto completo.

Pgvector

Pgvector es una extensión para PostgreSQL que permite almacenar, indexar y consultar vectores de alta dimensión, facilitando búsquedas semánticas eficientes con embeddings.

1.3.2. Python

Python fue el lenguaje principal para la implementación del backend, procesamiento de datos, manipulación de modelos de lenguaje y procesamiento de lenguaje natural, gracias a su amplia variedad de librerías y su facilidad de prototipado.

Kaggle

Se utilizó Kaggle como fuente para descargar el dataset público de películas, facilitando la obtención de datos limpios y estructurados para alimentar el buscador y también para crear cuadernos de Python con los que poder realizar la visualización y limpieza del dataset.

Pandas

La librería Pandas permitió la limpieza, transformación y análisis de los datos tabulares de películas, facilitando su manipulación para su posterior indexación y consulta.

Spacy

Se empleó en Python un modelo de reconocimiento de entidades nombradas en inglés de Spacy. Esto se usó para potenciar el componente de búsquedas clave.

Sentence Transformers

Esta librería se usó para la generación de *embeddings* de los documentos del dataset de películas y de las consultas del usuario, facilitando las búsquedas semánticas.

FastAPI

FastAPI fue el framework escogido para desarrollar la API REST que expone el buscador, gracias a su alto rendimiento, facilidad de uso y capacidad para documentar automáticamente los endpoints.

Pydantic

Se utilizó para validar y tipar datos en el backend y en las respuestas de los LLM, garantizando su consistencia.

LangChain

Se usó principalmente por su herramienta *PydanticOutputParser*, que facilitó la generación y validación de respuestas estructuradas de los LLM.

1.3.3. ReactJS

Para la interfaz de usuario se usó como framework ReactJS, permitiendo crear una experiencia dinámica e interactiva en la búsqueda y visualización de películas.

1.3.4. Tailwind CSS

Tailwind CSS se empleó para diseñar la interfaz con estilos modernos y responsivos, acelerando el desarrollo del frontend con utilidades predefinidas.

1.3.5. Cohere Rerank v3.5

Se utilizó este modelo para reordenar los resultados de búsqueda según su relevancia semántica, mejorando la precisión de las búsquedas.

1.3.6. Gemini Flash 2.0

Gemini Flash 2.0 se utilizó para generar evaluaciones de los resultados de búsqueda con los que computar métricas de calidad y para generar reseñas de los resultados y crear un RAG a partir del buscador.

1.4. Metodología utilizada

Se ha aplicado una metodología incremental iterativa basada en los principios ágiles para el desarrollo del proyecto. Esta metodología ha permitido al proyecto adaptarse a los cambios (requisitos, tecnologías, nuevos modelos de IA, etc.) de forma flexible al mismo tiempo que se mantiene el desarrollo activo. El proceso se dividirá en iteraciones, o sprints, cada uno con una duración variable.

Al inicio de cada sprint, se definen los objetivos concretos y las funcionalidades a implementar durante ese periodo. Al final de estos se realiza una revisión para evaluar el trabajo realizado y planificar las siguientes iteraciones. También se ha mantenido una documentación detallada del propio proceso de desarrollo, la planificación, los diseños desarrollados, etc.

1.5. Estructura del documento

Esta memoria se organiza en varios capítulos que detallan cada aspecto del proyecto, desde su motivación inicial a las conclusiones y propuesta de mejoras a futuro. A continuación, se describe la estructura de la memoria según esos capítulos:

- En este primer capitulo, la introducción al documento, se ha desgranado la motivación del proyecto, los objetivos que se pretenden alcanzar, un resumen de las tecnologías utilizadas y la metodología aplicada.
- En el segundo capítulo se expone el marco teórico del proyecto, donde se analiza desde el funcionamiento y las técnicas utilizadas en los motores de búsqueda tradicionales hasta los enfoques más modernos.
- En el tercer capítulo se detalla el desarrollo completo del buscador, comenzando por la obtención y tratamiento de los datos, seguido del desarrollo y mejora de los distintos componentes de búsqueda, en paralelo a sus respectivas evaluaciones.
- En el cuarto capítulo se presenta la aplicación web desarrollada a partir del buscador, junto con las diferentes funcionalidades que se le han integrado.

- En el quinto capítulo se presentan las conclusiones obtenidas del desarrollo del proyecto y las posibles líneas futuras que pueden surgir de este.

2

Marco Teórico

2.1. Motores de búsqueda tradicionales

La recuperación de información (Information Retrieval, IR) es un proceso que facilita la recuperación efectiva y eficiente de información relevante de grandes colecciones de datos no estructurados o semiestructurados. Los sistemas de IR ayudan a buscar localizar y presentar información que coincida con la búsqueda o necesidad de información de un usuario. [A la información recuperada se le suele denominar como “documentos”] [18].

Los motores de búsqueda (o buscadores) tradicionales han sido el pilar del IR durante décadas. Estos generalmente se basan en la coincidencia literal de palabras clave para buscar documentos relevantes en grandes volúmenes de datos. Estas palabras clave son aquellas palabras incluidas en un documento que cuentan con la capacidad de representar su contenido.

2.1.1. Índice invertido

Un índice invertido es una estructura de datos fundamental usada en sistemas de IR para localizar eficientemente documentos que contengan palabras clave específicas. Esta mapea palabras a una lista de documentos que contienen esas palabras. Un índice invertido tiene dos componentes principales [29]:

- **Vocabulario:** lista de términos únicos (palabras) encontradas en la colección.
- **Índice invertido:** mapeo entre los términos y los documentos que los contienen. Cada termino es asociado con una lista de identificadores de documentos y sus frecuencias de

aparición correspondientes.

Un ejemplo sencillo de esta estructura se puede observar en la tabla 1

Término	Documentos	Frecuencias
Acción	D1, D3	D1: 2, D3: 1
Romance	D2, D3, D4	D2: 1, D3: 2, D4: 1
Drama	D1, D2	D1: 1, D2: 1

Cuadro 1: Ejemplo de índice invertido

2.1.2. Funcionamiento de un motor de búsqueda tradicional.

Los buscadores tradicionales siguen el siguiente procedimiento general:

1. **Descubrimiento de datos:** se determina el contenido sobre el que se podrá buscar. Podrán ser desde datos estructurados y no estructurados previamente seleccionados (en nuestro caso con datos de películas) hasta contenido de internet seleccionados usando técnicas de optimización de motores de búsqueda (*Search Engine Optimization*, SEO) como el *crawling*, en el caso de las búsquedas en la web.
2. **Indexación:** Una vez obtenido los datos, se extraen las palabras clave, sus ubicaciones y se calculan sus frecuencias en los documentos para crear un índice invertido en el que buscar posteriormente.
3. **Búsqueda:** El usuario envía una consulta, el buscador separa la consulta en palabras clave y luego recupera documentos que contengan esas palabras clave.
4. **Ranking:** los documentos recuperados se pueden clasificar de forma que se presenten ordenados por relevancia y calidad, para ello se usan algoritmos que pueden tener en cuenta la presencia de palabras clave, historial de usuario, ubicación, entre otros.[19]

Las consultas de palabras clave se pueden clasificar según su especificidad y longitud como consulta de palabras clave de cola corta a larga 1, este concepto es propio del SEO. Cuanto más

específica y larga sea una consulta, menor será el volumen de búsqueda y más se acercará a lo que buscan concretamente los usuarios (figura 1).

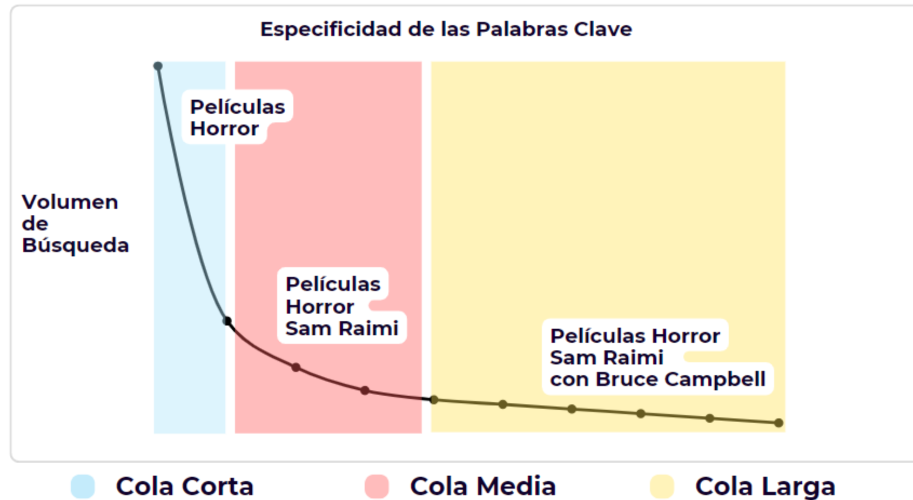

Figura 1: Clasificación de consultas de palabras clave según su longitud de cola.

Extracción de palabras clave y técnicas de ponderación

Para entender bien el proceso que siguen bien las búsquedas por palabras clave es necesario entender las técnicas más usadas para la extracción de palabras clave [6][36]:

- **Métodos basados en la Frecuencia:** estas técnicas identifican palabras clave a partir de su frecuencia en un documento, los métodos más comunes son:
  - **Frecuencia de Términos (TF):** mide la frecuencia de un término en un documento.
  - **Frecuencia de Documento Inversa (IDF):** mide la importancia de un término sobre una colección de documentos.
  - **TF-IDF:** combina las dos anteriores, asigna pesos a los documentos según su frecuencia en el documento y rareza sobre la colección de documentos.

- **Rapid Automatic Keyword Extraction (RAKE):** usa una lista de palabras vacías y signos de puntuación actuando como "delimitadores" para separar el texto en frases, luego analiza la conexión de las palabras en esas frases para darles una puntuación.
- **Métodos basados en grafos:** las frases se representan como grafos donde cada palabra es un nodo y las conexiones entre ellas son los arcos, luego se mide la importancia de cada nodo según su número de conexiones o usando algoritmos más sofisticados como *PageRank* (muy usado en el contexto del SEO).
- **Métodos basados en Machine learning:** a diferencia de los anteriores estos requieren datos de aprendizaje, son más complejos y pueden alcanzar bastante precisión, un enfoque muy común es el de *Conditional Random Fields* (CRF)

Una vez extraídas las palabras clave es necesario ponderarlas. Esto puede mejorar bastante las búsquedas, ya que no todas las palabras son igual de importantes en todos los contextos. Por ejemplo, cuando buscas una película, es de esperar que las palabras clave del título sean más importantes que las palabras clave incluidas en su argumento. Las técnicas de ponderación más comunes son [28]:

- **TF-IDF:** tal y como se mencionó antes, TD-IDF se puede usar para dar peso a las palabras según su frecuencia en el documento y rareza en la colección.
- **Okapi BM25:** este modelo probabilístico extiende TF-IDF con la normalización de la longitud del documento para evitar que los documentos más largos se vean favorecidos injustamente.
- **Clasificación semántica:** se pueden clasificar los resultados según su contexto y semántica, este enfoque nos será más útil para la búsqueda semántica y híbrida.

2.1.3. Limitaciones de las búsquedas por palabras clave

Las búsquedas por palabras clave son efectivas para consultas simples, sobre todo si el usuario sabe exactamente qué es lo que está buscando. Sin embargo este enfoque afronta varios retos [1]:

- **Mal emparejamiento de sinónimos:** los usuarios con frecuencia usan diferentes palabras para expresar lo mismo, lo que puede dificultar la búsqueda de resultados relevantes en buscadores tradicionales.
- **Polisemia:** muchas palabras tienen múltiples significados, las búsquedas por palabras clave pueden devolver resultados irrelevantes debido a la ambigüedad de las palabras.
- **Comprensión del Contexto:** los buscadores tradicionales no tienen la habilidad de entender el contexto de una consulta, lo que lleva a resultados imprecisos y a la pérdida de buenos resultados que no contengan exactamente las mismas palabras.
- **Consultas de Cola Larga:** los buscadores suelen tener problemas al tratar con consultas de cola larga, viéndose reduciendo drásticamente el volumen de búsqueda y perdiendo muchos resultados que podrían ser relevantes.

Para ilustrar esto considérese un usuario buscando "mejores películas de romance". El buscador tradicional devuelve resultados para "películas romance", sin embargo, se dejará resultados que incluyan las palabras como "amor, romántico, pasión...".

Estas limitaciones llevan a la necesidad de construir buscadores más sofisticados con técnicas como las búsqueda booleana, semántica e híbrida.

2.1.4. Búsqueda booleana

La búsqueda booleana no es más que una búsqueda por palabras clave que permite a los usuarios incluir operadores de la lógica booleana en las consultas para delimitar mejor los resultados recuperados [13]. Considera el siguiente ejemplo:

- **Consulta:** "(Horror **OR** Suspense) **NOT** Romance"
- **Resultado:** El resultado incluirá películas de horror o suspense, pero no de romance.

A pesar de ser un avance con respecto a la búsqueda tradicional sigue presentando las mismas limitaciones.

2.2. Búsqueda semántica

Meses después de que Google lanzara el artículo *"BERT: Pre-Training of deep bidirectional transformers for language understanding"*, anunciaron que iban a usarlo para impulsar la búsqueda de Google y que esto representaba: "Uno de los mayores saltos en la historia de la búsqueda". Para no quedarse atrás, Microsoft Bing también declaró: "A partir de abril de este año, utilizamos modelos de transformadores de gran tamaño para ofrecer las mayores mejoras de calidad a nuestros clientes de Bing en el último año."

Esto demuestra claramente la utilidad de estos modelos. Su integración mejora de manera casi inmediata y significativa algunos de los sistemas más desarrollados y mejor mantenidos en los que confían miles de millones de personas en todo el mundo (como Google o Bing). La capacidad que aportan se conoce como *búsqueda semántica* [10] .

Este tipo de búsqueda se basa en el significado y contexto de las palabras contenidas en los documentos y no tanto en la coincidencia exacta de palabras. La semántica, tal y como se define: *"Disciplina que estudia el significado de las unidades lingüísticas y de sus combinaciones"* [11] incorporada a los buscadores, nos permite interpretar la intención del usuario con mayor precisión.

Esto es posible gracias al procesamiento del lenguaje natural, gracias al cual, los ordenadores pueden entender y trabajar con el lenguaje humano. Además estos buscadores han mejorado sustancialmente, estos últimos años, gracias al auge de la inteligencia artificial (IA) y al desarrollo de los LLM.

2.2.1. Procesamiento de lenguaje natural

El procesamiento de lenguaje natural (NLP) es un subcampo de la informática e IA que utiliza el machine learning (ML) para permitir que los ordenadores entiendan y se comuniquen a través de lenguaje humano [25].

Los enfoques del NLP han cambiado significativamente a lo largo de los años: desde sis-

temas basado en reglas, pasando por enfoques estadísticos, hasta llegar al uso del *Deep Learning* (DL). Nosotros vamos a centrarnos en este último, con el cual se han conseguido unos resultados excelentes en la comprensión del lenguaje, posicionándose como actual enfoque dominante del NLP.

2.2.2. Técnicas de NLP

Es importante comprender las técnicas de NLP que serán útiles para procesar los textos, permitiendo su posterior tratamiento computacional y su aplicación en la creación y mejora de sistemas de IR.

Eliminación de palabras vacías

Las palabras vacías (ampliamente conocidas como stop words), es el nombre que reciben aquellas palabras que no tienen significado, como los artículos, pronombres, proposiciones, etc. [También se incluyen aquellas palabras que sean tan frecuentes que no aporten un significado real en un determinado contexto.] [39]

En muchos contextos es conveniente deshacerse de estas palabras, por ejemplo, en el caso de las búsquedas por palabras clave, la inclusión de estas palabras puede generar ruido, ya que no aportan significado y aumentan innecesariamente el numero de datos a procesar afectando a la precisión y a la eficiencia del sistema. No obstante, en tareas más complejas donde la semántica prevalezca, a veces es conveniente conservarlas por su valor contextual.

La forma más común de eliminar estas palabras es mediante el uso de listas de palabras vacías, que contienen los términos que deben de ser eliminados. Estas listas suelen estar predefinidas por el idioma del contexto; por ejemplo, en inglés incluyen palabras como *a, an, the*, entre otras. Si es necesario, estas listas pueden ampliarse con palabras del dominio que aparezcan con demasiada frecuencia y no aporten valor.

Stemming

El *stemming* es un método utilizado para reducir palabras a su raíz o (en inglés) a un *stem.* Por ejemplo, para las palabras bibliotecario, biblioteca, bibliotecas, etc. La raíz es *bibliotec* y las palabras anteriores son sus variantes morfológicas [37].

La aplicación de esta técnica puede mejorar las métricas de los sistemas de IR. Por ejemplo, en el caso de las búsquedas por palabras clave, puede aumentar el número de resultados relevantes obtenidos al emparejar automáticamente las diferentes formas morfológicas de una palabra, incrementando así el volumen de resultados útiles. De nuevo, en tareas donde la semántica prevalezca, sigue siendo interesante no aplicar esta técnica.

El algoritmo más usado para esto es el algoritmo de Porter. Además existen otros métodos basados en análisis lexicográfico y otros algoritmos similares (KSTEM, stemming con cuerpo, métodos lingüísticos...) [37].

Etiquetado gramatical

El etiquetado gramatical (ampliamente conocido como *part-of-speech tagging*) es el proceso de asignar a cada una de las palabras de un texto su categoría gramatical (verbo, sustantivo, etc.). Un ejemplo se puede observar en la figura 2.

Categoría Gramatical	Etiqueta	Ejemplo
Pronombre	PRON	
Verbo	VERB	Me encanta el café
Determinante	DET	PRON VERB DET NOUN
Nombre	NOUN	

Figura 2: Ejemplo básico de etiquetado gramatical

Esta técnica es interesante para aquellas tareas donde se deba de dar más importancia a un

cierto tipo de palabras que a otras o para usarse en conjunto a otras técnicas de NLP.

Las soluciones que se han propuesto para llevar a cabo esta tarea ha evolucionado de aproximaciones lingüísticas mediante sistemas basados en reglas, al ML.

Lematización

La lematización es una extensión del *stemming*, pero más precisa. Consiste en identificar la forma base de una palabra, conocida como lema a partir de su forma flexionada. A diferencia del *stemming* que recorta palabras sin tener en cuenta su contexto, la lematización requiere conocer la categoría gramatical y, en muchos casos, el significado contextual de la palabra en la oración así como el contexto que le rodea, como pueden ser las frases vecinas e incluso el documento completo.

Para reflejar mejor la importancia de esto, vamos a observar dos ejemplos donde la lematización supera al *stemming*:

1. La palabra *Mejor* tiene *Bueno* como lema, el *stemming* falla aquí al ser necesario revisar un diccionario para llegar al lema.
2. La palabra *Canto* se puede interpretar como una forma flexionada de *Cantar* o como un sustantivo, dependiendo del contexto. Por ejemplo: en la frase "Yo canto una canción" actúa como verbo, pero en la frase "El canto de los pájaros es relajante" actúa como sustantivo. A diferencia del stemming, la lematización intenta identificar correctamente el lema considerando el contexto gramatical y semántico de la oración, pudiendo distinguir cuando el lema es cantar o canto.

Los enfoques más aplicados son el uso de diccionarios y de sistemas basados en reglas, como el etiquetado gramatical, aunque sistemas más avanzados recurren al ML que permite una mayor precisión al considerar el contexto lingüístico [38].

Tokenización

La tokenización es el proceso de dividir texto en unidades más pequeñas conocidas como tokens, convirtiéndolo en unidades más manejables para el ordenador. Se puede clasificar en función de cómo se divide el texto [21]:

- **Tokenización de palabras:**
  Entrada: *"La tokenización es una técnica NLP"*
  Salida: *["La", "tokenización", "es", "una", "técnica", "NLP"]*
- **Tokenización de frases:**
  Entrada: *"La tokenización es una técnica NLP. Divide el texto en partes pequeñas"*
  Salida: *["La tokenización es una técnica NLP.", "Divide el texto en partes pequeñas"]*
- **Tokenización de subpalabras:**
  Entrada: *"Tokenización"*
  Salida: *["Toke", "nización"]*
- **Tokenización de caracteres:**
  Entrada: *"Tokenización"*
  Salida:*["T", "o", "k", "e", "n", "i", "z", "a", "c", "i", "ó", "n"]*

Esta técnica es un paso crítico que hace el procesamiento de texto más efectivo en las tareas de NLP [21]:

- **Normalización del texto:** La tokenización reduce el tamaño del texto sin formato, lo que resulta en un análisis estadístico y computacional fácil y eficiente.
- **Extracción de características:** Los tokens permiten crear representaciones numéricas del texto que sirven como características para modelos de ML y DL.
- **Recuperación de información:** La tokenización es otra forma de identificar palabras clave, esencial para nuestra tarea.

- **Adaptación a tareas específicas:** La tokenización se puede personalizar para diferentes tareas NLP, como en resumen de textos, traducción automática, análisis del sentimiento, etc.

La tokenización se puede implementar de numerosas formas, usando desde enfoques más tradicionales, como separar palabras por el espacio o los signos de puntuación que las separa, a tokenizadores más sofisticados como los modelos de lenguaje BERT [21].

Reconocimiento de entidades nombradas

El reconocimiento de entidades nombradas (NER) es una tarea de extracción de información que identifica categorías predefinidas de objetos en un cuerpo de texto. Aunque se pueden crear categorías personalizadas, hay ciertas categorías que se han ido estandarizando con el paso del tiempo, entre ellas están: entidades que hacen referencia a personas, organizaciones, empresas, lugares geográficos, fechas y tiempos, entre otros.

El NER es una tarea crucial dentro del NLP ya que ayuda a comprender el lenguaje y a conocer de qué habla un texto y qué se menciona de forma implícita y explícita.

Las técnicas empleadas han evolucionado desde reglas basadas en lenguaje, a modelos estadísticos y finalmente modelos de DL, aumentando así gradualmente la complejidad y la precisión. Asimismo, existen enfoques híbridos que combinan las fortalezas de distintas metodologías para optimizar los resultados [26].

Vectorización de palabras

La vectorización de palabras (ampliamente conocido como *word embeddings*) es una forma de representar palabras en vectores multidimensionales, donde la distancia y dirección entre vectores refleja la similaridad y relaciones entre las palabras correspondientes

El desarrollo de *embeddings* para representar contenido textual ha jugado un papel crucial en el avance de las aplicaciones de NLP y ML.

Los métodos tradicionales de representar palabras de forma comprensible para las máquinas como la *codificación one-hot*, representan cada palabra como un vector disperso de dimensión igual al tamaño del vocabulario. En estos métodos solo un elemento del vector tiene asignado el valor 1, para indicar la presencia de esa palabra (figura 3). A pesar de la simplicidad de este enfoque, presenta una dimensionalidad excesiva, carece de información semántica y no captura relación entre palabras [12].

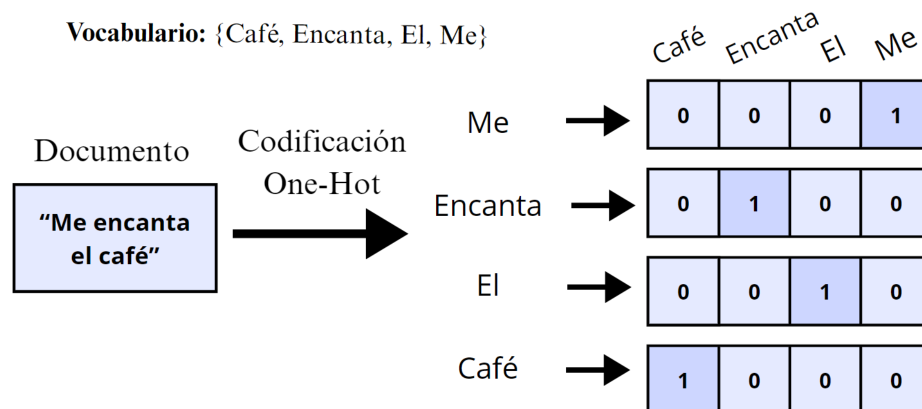

Figura 3: Ejemplo básico de construcción de vectores dispersos con codificación one-hot

Este método tradicional se puede ampliar a la representación del lenguaje en documentos, como una bolsa de palabras (*bag of words*), donde se tokenizan las diferentes frases, se crea un vocabulario a partir de las palabras únicas y se cuentan las ocurrencias de estas palabras en las diferentes frases (figura 4) [10].

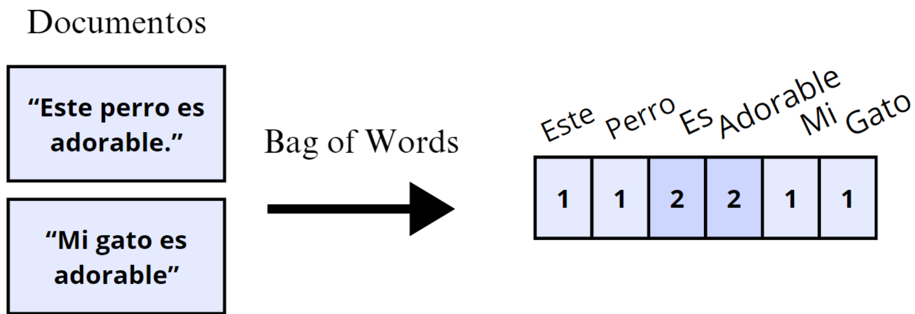

Figura 4: Ejemplo básico de construcción de una bolsa de palabras

A diferencia de estos enfoques, los *word embeddings*, son vectores densos con valores continuos y de menor dimensionalidad (figura 5), que son entrenados usando técnicas de ML, a menudo basadas en redes neuronales, para aprender representaciones que codifiquen el significado semántico y las relaciones entre las palabras.

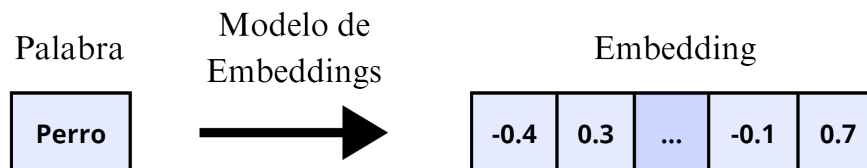

Figura 5: Ejemplo básico de vectorización de una palabra

Un método popular para entrenar modelos de *word embeddings* es Word2Vec, que usa una red neuronal para predecir las palabras circundantes de una palabra objetivo dado un contexto. Otro método ampliamente usado es GloVe (*Global Vectors for Word Representation*), que aprovecha estadísticas globales para crear embeddings [12].

Los embeddings han ido evolucionando para representar diferentes tipos de contenido textual, desde tokens, palabras, frases y hasta documentos [10]. Estos han demostrado ser de gran valor para las tareas NLP, ya que permiten entender y procesar las relaciones semánticas

entre las palabras con mayor precisión que los métodos tradicionales. Entre esas tareas se encuentran la clasificación de texto, NER, IR, generación de texto, etc.

Creación de embeddings

Este proceso requiere el entrenamiento de un modelo sobre una gran cantidad de contenido textual (p.ej. Wikipedia, Google News, etc.) Este se preprocesa mediante tokenización, eliminación de *stop words* y otras tareas de limpieza general.

Una ventana deslizante de contexto se aplica al texto. Para cada palabra objetivo, se consideran las palabras vecinas dentro de una ventana deslizante como contexto. Los modelos de embeddings de palabras están entrenados para predecir una palabra objetivo según su contexto o viceversa. Este entrenamiento conlleva ajustar parámetros del modelo de embeddings para minimizar la diferencia entre las palabras precedidas y las palabras reales en el contexto.

Esto permite a los modelos capturar los diversos patrones lingüísticos y asignar a cada palabra un vector único, que representa la posición de la palabra en un espacio vectorial continuo.

Entre los beneficios derivados de la creación de los embeddings, además de la reducción de la dimensionalidad de las representaciones textuales y la incorporación de semántica y contexto, se encuentran [12]:

- **Hipótesis distribucional:** plantea que las palabras con significados similares tienden a aparecer en contextos similares. Este principio es la base de la mayoría de los modelos de embeddings, que buscan capturar relaciones semánticas a partir de los patrones de coocurrencia entre palabras.
- **Generalización:** los embeddings generalizan bien frente a palabras no vistas durante el entrenamiento o palabras poco frecuentes, ya que aprenden a representarlas en función del contexto. Esto resulta especialmente útil al tratar con vocabularios amplios o cambiantes.

2.2.3. Codificación y decodificación del contexto

Un gran avance en el procesamiento del contexto textual se logró con las Redes Neuronales Recurrentes (RNN). Estas son variantes de las redes neuronales diseñadas para modelar datos secuenciales, al incorporar la información de pasos anteriores como parte de la entrada.

Estas RNNs se usan para dos tareas, *codificar* o representar una secuencia de entrada y *decodificar* o generar una secuencia de salida. Cada paso es autorregresivo: el modelo genera cada palabra basándose en las anteriores.

El paso de codificación está orientado a representar el texto de la forma más informativa posible, generando un embedding que condensa el contexto y que servirá de entrada al decodificador. Para generar esa representación contextual, el codificador recibe como entrada los embeddings de las palabras individuales (figura 6).

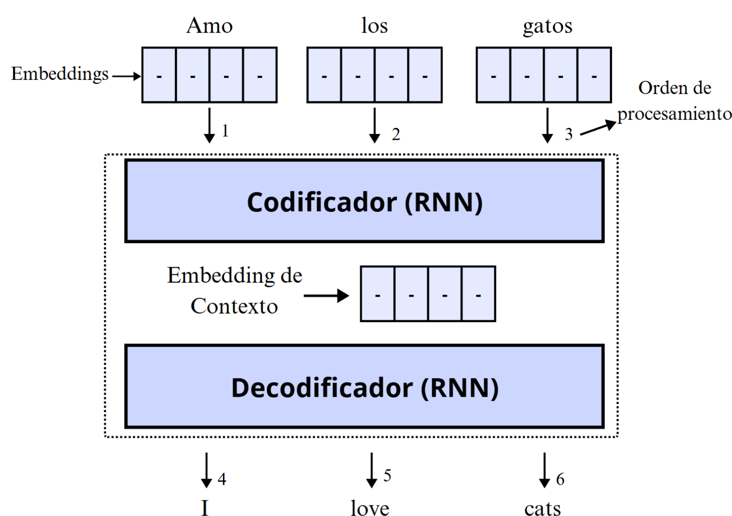

Figura 6: Ejemplo de codificación y decodificación del contexto con RNN

Representar toda la frase con un único embedding puede ser problemático en textos largos. En 2014 se introdujo una solución llamada *Atención* que supuso una gran mejora en la arquitectura. La atención permite al modelo centrarse en ciertas partes de la secuencia de entrada que son más relevantes que otras. Este mecanismo determina selectivamente qué palabras son más importantes dada una frase. [10].

2.2.4. Modelos transformadores

Hasta ahora se han mencionando diferentes modelos de representación de lenguaje, empezando por la bolsa de palabras, seguido por Word2Vec, GloVe y RNN. La evolución de estos modelos de lenguaje se puede apreciar en la figura 7. De estos nos centraremos en los LLM, de gran relevancia en la actualidad, que están marcados por el auge de los modelos transformadores.

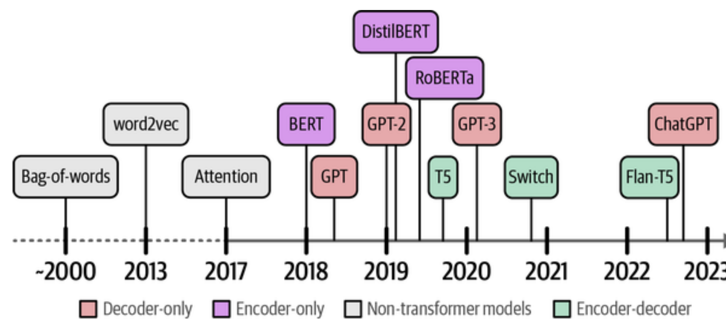

Figura 7: Evolución de los modelos de lenguaje. Imagen tomada del libro *Hands-On Large Language Models*.

Con la publicación del artículo *"Attention is all you need"* en 2017, los autores propusieron una arquitectura neuronal llamada *Transformer*, basada exclusivamente en un mecanismo de atención. Al eliminar las RNN, las secuencias se pueden procesar en paralelo, acelerando así el entrenamiento.

En un transformer, el codificador y decodificador están apilados uno encima de otro. La arquitectura sigue siendo autorregresiva, necesitando consumir cada palabra generada antes de crear una nueva (figura 8).

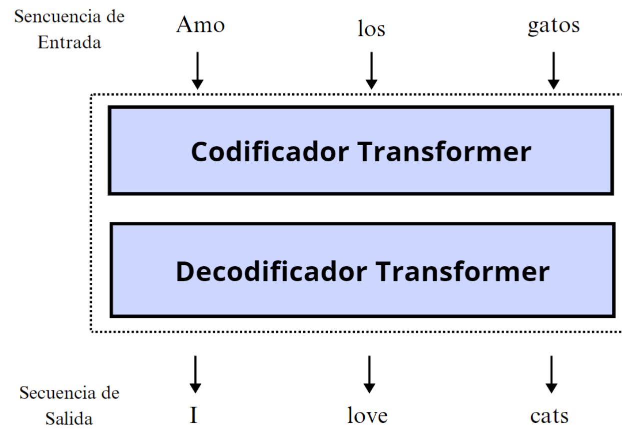

Figura 8: Arquitectura básica de los transformers

Actualmente, tanto el codificador como el decodificador se basan exclusivamente en mecanismos de atención, en lugar de depender de RNN complementadas con atención. El codificador de un Transformer se compone principalmente de dos bloques: self-attention, que procesa la secuencia completa simultáneamente para capturar relaciones contextuales, y una red feedforward que transforma individualmente cada token para enriquecer su representación (figura 9).

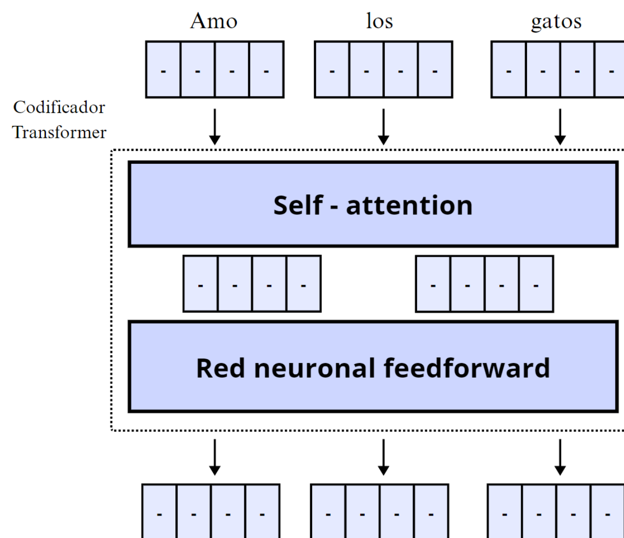

Figura 9: Arquitectura de los codificadores transformers

En el decodificador, la capa de self-attention está enmascarada para impedir que el modelo

acceda a posiciones futuras, garantizando así que la generación sea autorregresiva y evitando la fuga de información durante la creación de la salida.

Además, el decodificador incluye una capa adicional que atiende la salida del codificador, permitiendo enfocarse en las partes relevantes de la secuencia de entrada [10].

Con el paso del tiempo se han ido desarrollando diferentes arquitecturas de transformador, que se pueden clasificar según qué componentes de codificación y decodificación presentan [?]:

- **Solo Codificadores:** Tienen una mejor compresión contextual y representaciones semánticas más fuertes.

  Se usa en clasificación de textos, análisis de sentimientos, NER, búsqueda semántica, etc.

- **Solo Decodificadores:** Generación de texto fluida y coherente, buen aprendizaje autoregresivo.

  Se usa en generación de texto creativo, chatbots, escritura automática, etc.

- **Codificador-Decodificador:** Versátil y multitarea eficiente, a coste de un entrenamiento más costoso e inferencia más lenta

  Se usa en traducción de texto, generación de resúmenes, preguntas y respuestas, etc.

2.2.5. Funcionamiento de un motor de búsqueda semántico

Este tipo de buscadores se basan en el uso de los embeddings, concepto que hemos abordado anteriormente, y convierte la búsqueda en la recuperación de vecinos cercanos (*Aproximate Nearest Neighbour*, ANN) a la consulta del usuario (después de que tanto la consulta del usuario como los documentos se hayan convertido en embeddings).

Recordemos que los embeddings convertían el texto en representaciones numéricas, ubicándolos en un espacio vectorial continuo, donde la distancia y dirección entre vectores refleja la similaridad semántica y la relación contextual entre palabras.

Esta propiedad es la que se usa para construir estos buscadores. En este escenario, cuando un usuario introduce una consulta, se convierte en un embedding de consulta, se proyecta en el mismo espacio que los documentos (figura 10) y luego simplemente se buscan los documentos más cercanos a la consulta en ese espacio, estos serán los resultados de búsqueda.

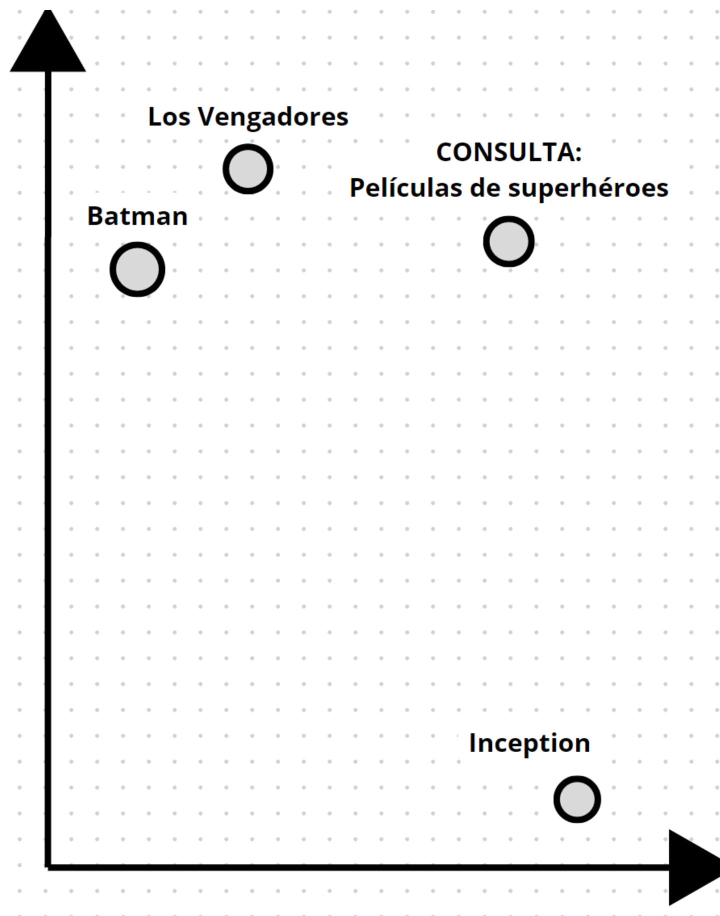

Figura 10: Distancia entre documentos y consulta en el espacio vectorial de embeddings

Ante esto aparecen dos cuestiones [10]:

- **¿Qué distancia semántica implica que los resultados dejan de ser relevantes?** Esto recae en una decisión de diseño, suele ser recomendable tener un umbral de similaridad para filtrar los resultados.
- **¿Son realmente la consulta del usuario y el mejor resultado semánticamente similares?** No siempre, por ello los modelos de lenguaje deben ser entrenados con pares de preguntas y respuestas para ser mejorar en esta tarea.

Por lo general, existen muchos modelos que ya están adaptados para la tarea de IR, es decir, ya están entrenados a partir de pares de consulta y documentos relevantes. Esto permite al modelo aprender representaciones (embeddings) distintas para consultas y documentos, lo que se conoce como *búsqueda semántica asimétrica* [31].

Los motores de búsqueda semánticos siguen el siguiente procedimiento general:

1. **Descubrimiento de datos:** Análogamente a la búsqueda por palabras clave se debe recopilar los datos sobre los que se buscará posteriormente.
2. **Selección del modelo:** Se lleva a cabo la selección de un modelo de lenguaje para generar los embeddings, buscando el equilibrio entre coste computacional y precisión semántica que mejor se ajuste a la aplicación.
3. **División de documentos:** Se debe designar como serán los documentos individuales del conjunto de datos. El tamaño de cada documento dependerá tanto del tipo de datos como del límite máximo de tokens que el modelo puede procesar. Si el documento definido excede ese límite, será necesario aplicar técnicas de chunking para fragmentarlo y aprovecharlo adecuadamente.
4. **Generación y almacenamiento de embeddings:** Los embeddings de los datos deben ser generados y almacenados adecuadamente en una base de datos vectorial.
5. **Indexación:** Al igual que en la búsqueda por palabras clave es recomendable crear un índice a partir de los embeddings almacenados, de forma que se pueda realizar de forma efectiva ANN en el espacio vectorial semántico a partir del embedding de consulta.
6. **Búsqueda:** Se crea el embedding de consulta y se computa la búsqueda semántica en el índice enfrentando el embedding de consulta y los almacenados, a partir de una métrica de similitud semántica, y se recuperan los documentos más similares a la consulta; estos vienen ya ordenados por similitud semántica.

Métricas de similitud semántica

Para poder calcular la similitud semántica entre dos embeddings se suelen usar las siguientes métricas [40]:

- **Distancia Euclidea:** En esencia esta distancia es la longitud del segmento que separa dos puntos, cuanta menor sea la distancia mayor es la similitud.
- **Similaridad Coseno:** Mide el angulo entre dos vectores, cuanto menor sea el ángulo más similares serán los vectores.
- **Producto Escalar:** Es el resultado de multiplicar dos vectores y sumar sus componentes correspondientes. Su valor es proporcional al coseno del ángulo entre ellos y al producto de sus magnitudes. Cuanto mayor sea el producto escalar más similares serán los vectores.

2.2.6. Limitaciones de la búsqueda semántica

A pesar de que la búsqueda semántica ofrece una mejor comprensión de la intención del usuario y mejora la recuperación de información en muchos contextos, aún presenta ciertas limitaciones importantes [34]:

- **Consultas precisas:** En numerosos casos los usuarios desean obtener resultados que coincidan exactamente con los términos de consulta. En este escenario el uso de una búsqueda por palabras clave resulta más adecuada.
- **Vocabularios específicos:** Muchos dominios presentan un vocabulario específico, abreviaciones, anotaciones, símbolos o nombres propios que los modelos semánticos pueden no interpretar correctamente. La búsqueda por palabras clave permite mantener la precisión en estos casos, al no intentar generalizar ni reinterpretar el significado de términos particulares.

- **Control de la ambigüedad:** Hay casos donde el usuario quiere mantener el control sobre las interpretación de su consulta. Por ejemplo, una palabra puede tener múltiples significados, pero el usuario puede estar buscando uno en específico. De nuevo aquí la búsqueda por palabras clave funciona mejor al hacer una búsqueda textual más literal.

2.3. Búsqueda híbrida

La búsqueda híbrida combina los resultados de una búsqueda por palabras clave con los de una búsqueda semántica, aprovechando así la precisión de las palabras clave y la comprensión contextual de la semántica. Esto permite obtener resultados que son a la vez exactos y relevantes según la intención del usuario, adaptándose a consultas específicas o más generales.

2.3.1. Técnicas de fusión

Tras realizar tanto la búsqueda por palabras clave como la semántica, se obtienen dos listas de documentos, cada una ordenada según una puntuación. En cada caso, esta puntuación es calculada según su propio criterio de relevancia. A la lista de puntuaciones se le conoce como *ranking* y a la posición en la que se encuentra el documento como *rank*, de forma que a menor rank, mejor posicionado estará el documento en el ranking.

Para poder combinar ambos rankings de forma coherente, aparecen diferentes técnicas de fusión. Estas técnicas permiten combinar las salidas de distintos sistemas de recuperación, equilibrando sus respectivos rankings para obtener una clasificación final más robusta y representativa.

Suma ponderada

Para aplicar esta técnica de fusión se debe decidir qué peso darle a cada tipo de búsqueda. Una vez determinados esos pesos, solo se debe realizar la suma ponderada entre ambos rankings y ordenar luego ese ranking combinado. Por ejemplo, supongamos que tenemos las siguientes puntuaciones en dos rankings para un documento dado:

$$R_1 = 0{,}8, \quad R_2 = 0{,}6$$

y asignamos a cada ranking los siguientes pesos:

$$w_1 = 0{,}7, \quad w_2 = 0{,}3$$

La suma ponderada para obtener el ranking combinado de ese documento será:

$$R_{combinado} = 0{,}7 \times 0{,}8 + 0{,}3 \times 0{,}6 = 0{,}74$$

Esta técnica puede llegar a ser muy efectiva con una calibración adecuada de las ponderaciones. Sin embargo, lograr esa calibración puede resultar complejo[16].

Fusión por rango recíproco

La Fusión por Rango Recíproco (*Reciprocal Rank Fusión*, RRF) es un método de agregación que combina diferentes listas de puntuación en una única lista, este método sigue la siguiente fórmula:

$$\text{RRF}(d) = \sum_{r \in R} \frac{1}{k + r(d)}$$

Donde:

- *d* es un documento,
- *R* es el conjunto de rankings a fusionar,
- *k* es una constante (normalmente 60),
- *r(d)* es el rank del documento d en el ranking d.

A diferencia de la suma ponderada, este método no se basa en las puntuaciones de relevancia, lo cual resulta especialmente útil cuando dichas puntuaciones no son directamente comparables entre varios rankings. Además, este enfoque no requiere calibrar ponderaciones, lo que simplifica su aplicación. La intuición matemática detrás de esta fórmula es la siguiente:

1. Usar *1/(r(d) + k)* da más peso a los documentos mejor posicionados, favoreciendo aquellos que tienen buenas posiciones en varios rankings.
2. La contribución de cada documento disminuye rápidamente conforme baja su posición, porque la diferencia entre los primeros lugares es más significativa que entre posiciones lejanas.
3. Al sumar los valores recíprocos de diferentes rankings, RRF combina bien los resultados, creando un ranking final más estable y equilibrado.
4. La constante *k* actúa como un factor de suavizado para evitar que un solo ranking influya demasiado, ayudando a equilibrar documentos con posiciones más bajas.

Si bien en teoría se puede asignar cualquier valor a la constante *k*, el valor 60 se ha adoptado comúnmente por motivos empíricos, ya que varios estudios han mostrado que ese valor funciona bien sobre varios conjuntos de datos y tareas de recuperación de información [33]. Por ejemplo, supongamos que queremos calcular el RRF para los rankings que aparecen en la tabla 2.

Posición	**Ranking 1**	**Ranking 2**
1	D1	D2
2	D2	D3
3	D3	D4

Cuadro 2: Ejemplo de dos rankings para calcular RRF

Calculamos los valores para cada documento:

- $RRF(D1) = \frac{1}{60+1} = \frac{1}{61} \approx 0{,}016$

- $RRF(D2) = \frac{1}{60+2} + \frac{1}{60+1} = \frac{1}{62} + \frac{1}{61} \approx 0{,}0161 + 0{,}0164 = 0{,}0325$
- $RRF(D3) = \frac{1}{60+3} + \frac{1}{60+2} = \frac{1}{63} + \frac{1}{62} \approx 0{,}0159 + 0{,}0161 = 0{,}032$
- $RRF(D4) = \frac{1}{60+3} = \frac{1}{63} \approx 0{,}0159$

Reordenadores

Los Reordenadores (*Rerankers*) son modelos de lenguaje entrenados específicamente para asignar puntuaciones de relevancia a pares documentos y consultas. Su objetivo es mejorar la calidad del ranking final [10].

Existen distintos tipos de modelos aplicables a esta tarea, tradicionalmente se usaban modelos entrenados específicamente para reordenar como los modelos *Learning to Rank* (LTR) que predice las puntuaciones a partir de características más simples extraídas de los documentos y las consultas, como la puntuación BM25. Son rápidos y eficientes, aunque con menor precisión [17].

Hoy en día, los rerankers más comunes son los cross-encoders, que reciben como entrada cada consulta y documento, devolviendo una puntuación entre 0 y 1. Estos están entrenados para capturar interacciones detalladas entre ambos textos, lo que les otorga gran precisión, aunque a costa de un alto coste computacional, ya que procesan cada par de forma individual.

Estos rerankers se pueden aplicar en distintos puntos del flujo de búsqueda. Sin embargo, debido a su alto coste computacional, lo más habitual usarlo únicamente al final [10].

Aunque no son una técnica de fusión en sentido estricto, los rerankers pueden aplicarse sobre una lista unificada de resultados provenientes de distintos buscadores. Al reordenarla según su estimación de relevancia, generan un ranking combinado más preciso que el logrado con técnicas de fusión tradicionales. En cualquier caso, el reordenamiento se puede beneficiar de usar métodos de fusión previos antes del reranking.

Un enfoque alternativo, aunque menos común en entornos productivos debido a su elevado coste, es usar LLMs generativos como GPT para reordenar. Esto puede ser aún más preciso que

los cross-encoders, ya que aprovecha la capacidad de comprensión y razonamiento de este tipo de modelos, pero es aún más costoso.

2.3.2. Arquitecturas híbridas

Ahora que sabemos cómo podemos combinar diferentes tipos de búsquedas, el siguiente paso es integrar esa combinación en el flujo completo de búsqueda, dando lugar a lo que se conoce como una arquitectura de búsqueda híbrida. Puede implementarse de distintas maneras, según las necesidades y recursos del sistema:

- **Búsqueda Paralela:** La búsqueda por palabras clave y la semántica se ejecutan en paralelo y luego son fusionados. Es computacionalmente más costoso pero puede dar mejores resultados.
- **Búsqueda Secuencial:** Se selecciona un tipo de búsqueda que se ejecutará primero y luego se ejecuta la otra sobre el primer conjunto de resultados obtenidos. Este enfoque es más eficiente pero puede perderse documentos relevantes si el primer componente de búsqueda no era el adecuado.
- **Índice Híbrido:** Se pueden crear índices de búsqueda híbridos que combinen la información de las palabras clave y la vectorial. Esto puede optimizar la búsqueda pero requiere técnicas de indexación más complejas [30].

3

Desarrollo del Buscador

3.1. Obtención, visualización y tratamiento de los datos

El primer paso para desarrollar este motor de búsqueda de películas, era tener un conjunto de datos (dataset) de películas lo suficientemente grande y completo que permita realizar búsquedas que aprovechen todo el potencial del buscador híbrido.

El dataset sobre el que se trabajó se obtuvo de un recurso público disponible en Kaggle [3]. Este contiene datos en inglés de más de 950 mil películas. Los datos están en formato csv, aprovechando esto se creó un cuaderno de kaggle para poder visualizar adecuadamente los datos de cada csv. A continuación se detalla el contenido original de dataset y las conclusiones obtenidas a partir de diferentes visualizaciones:

movies.csv

Este CSV contiene la información general de las películas:

- **id:** Identificador de la película dentro del dataset.
- **name:** Nombre de la película.
- **date:** Año de lanzamiento de la película.
- **tagline:** Eslogan de la película.

- **description:** Descripción del argumento de la película.
- **minute:** Duración de la película.
- **rating:** Puntuación media de la película entre 0 y 5.

id	name	date	tagline	description	minute	rating
0	10	91159	800714	160814	180383	849091

Cuadro 3: Valores nulos por campos en movies.csv

El dataset contiene un total de 941597 entradas. La tabla 3, muestra cuántos de estos campos contienen valores nulos. Los registros que tenían el nombre y la descripción con valores nulos fueron descartados, ya que estos campos son esenciales para las búsquedas. Además se puede observar que el eslogan de la película es nulo en casi todos los documentos, por lo que a pesar de su posible potencial no fue considerado en la búsqueda semántica.

Tras esta limpieza nos quedamos con 772976 entradas. También se observó que hay descripciones de películas que se repiten (tabla 4).

Cantidad	Únicos	Valor más frecuente	frecuencia
779276	763336	Mexican feature film	893

Cuadro 4: Resumen estadístico del campo *description*

Esto podía llegar a generar ruido en la búsqueda semántica, en especial si estas descripciones no caracterizaban apropiadamente a las películas. Para paliar este problema, se observaron aquellas descripciones que no eran únicas y entre estas eliminaron aquellas descripciones con contenido irrelevante o vacío. A continuación se muestran varios ejemplos de esas descripciones:

["Plot Unknown", "No description found", "Direct to video", " ", ...]

Con esto se redujo el número de entradas a 779276. Dado que los datos están en inglés, el motor de búsqueda semántico que desarrollaremos operará en ese mismo idioma. Para garantizar la robustez de la solución, se eliminaron aquellas películas que no tuvieran su descripción en inglés. Para ello, se hizo uso de la librería de Python *fast-langdetect*. Tras esta última limpieza, el conjunto final quedó en 775793 registros, que será la base sobre la cual se construirá el buscador.

actors.csv

Este CSV contiene información de los actores que participan en cada película.

- **id:** Identificador de la película dentro del dataset.
- **name:** Nombre del actor.
- **role:** Rol del actor en la película.

Una característica clave que se ha podido observar en este csv, es que los actores vienen por defecto en orden de relevancia dentro de la película: primero protagonistas, luego personajes secundarios y así hasta los extras.

Por ejemplo, en el caso de la película Barbie, aparecen primero *Margot Robbie* y *Ryan Gosling* que son los protagonistas y luego el resto de actores con su respectivo orden descendente de relevancia.

El campo *role* se descartó para centrarnos en el propio metadato de los actores.

countries.csv

Este CSV incluye datos relativos a los países de producción de cada película.

- **id:** Identificador de la película dentro del dataset.

- **name:** Código o nombre del país (UK, USA, Ireland, etc.).

Este campo se descartó y se utilizó únicamente como dato informativo, ya que no venía en un formato completamente normalizado (mezclando códigos con nombres propios, lo que dificulta la consistencia en las consultas) y podía generar ruido o ambigüedad geográfica (por ejemplo, confundiendo el país de producción con el país donde transcurre la película). Además, su uso como filtro resultaba demasiado restrictivo.

genres.csv

Este CSV incluye información de los géneros de cada película.

- **id:** Identificador de la película dentro del dataset.
- **name:** Género cinematográfico de la película.

crew.csv

Este CSV incluye información del equipo técnico de cada película.

- **id:** Identificador de la película dentro del dataset.
- **role:** Rol en el equipo técnico (Director, Productor, etc.).
- **name:** Nombre del miembro del equipo técnico.

En este archivo se han observado una gran cantidad de roles dentro del equipo técnico: iluminación, operador de cámara, etc. Para evitar que esto pueda generar ruido y ajustarnos a los miembros de equipo en el que más suelen mostrar interés los usuarios al buscar películas, se han eliminado todas las entradas que no tengan el rol de director, productor, escritor, compositor y cinematógrafo. Esto resultó en una reducción de 4720185 a 2708237 registros.

posters.csv

Este CSV incluye los pósteres de cada película.

- **id:** Identificador de la película dentro del dataset.
- **link:** Enlace a la imagen del póster de la película.

Estos enlaces se fusionaron con los datos de movies.csv y se usarán para mostrar los pósteres en la interfaz de la aplicación. Hay otros archivos en el dataset que no se incluyeron para el desarrollo del buscador. A continuación se detalla brevemente, para cada uno, la decisión de diseño que se ha tomado:

- **languages.csv:** Contiene información acerca de todos los idiomas hablados en cada película. El problema de este archivo es que incluye todos los idiomas, incluso cuando en una película en inglés solo se habla francés en un diálogo aislado, lo que aumenta considerablemente el ruido en el conjunto de búsqueda.
- **releases.csv:** Contiene información relativa al lanzamiento de cada película en diferentes países (país, clasificación de edad, fecha de lanzamiento, tipo de lanzamiento). Se ha considerado que esta información no es relevante para el buscador.
- **studios.csv:** Este archivo contiene información del estudio que ha producido la película. No se ha incluido debido a que aumentaba mucho el tamaño del conjunto de búsqueda y no tantos usuarios buscan películas a partir de este campo.
- **themes.csv:** Este archivo incluye un texto que describe la temática de las películas, aunque suene muy conveniente para la búsqueda semántica, este metadato aparece asociado a muy pocas películas del dataset.

3.2. Almacenamiento y gestión de los datos

Dado que se iba a almacenar los datos en PostgreSQL y ya incluye una opción de importar datos a tablas a partir de CSV, simplemente se crearon las tablas adecuadas para cada CSV

y fueron importados. Para las tablas que están relacionadas con la tabla *movies* por su id, se crearon tablas intermedias siguiendo una relación de muchos a muchos en SQL.

Tras esto el diseño preliminar de la base de datos queda listo para empezar a desarrollar el motor de búsqueda (figura 11).

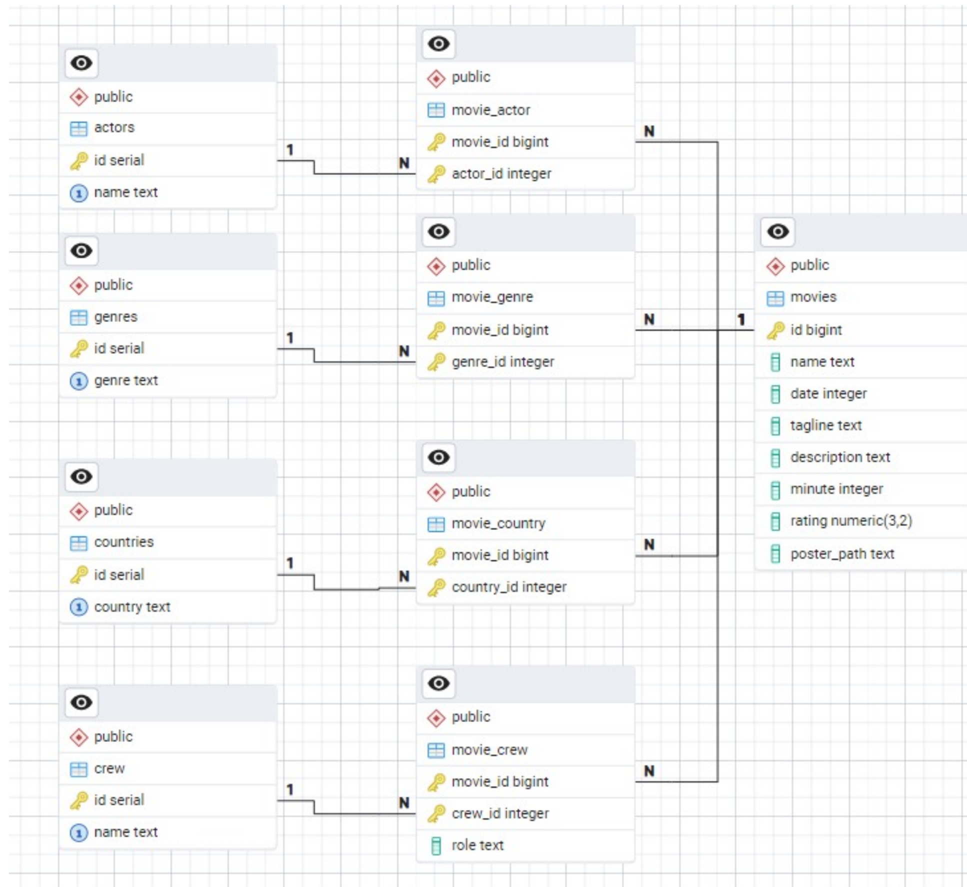

Figura 11: Diseño preliminar de la base de datos.

Antes de empezar a detallar cómo se implementaron los diferentes componentes de búsqueda, es importante definir cuál es el formato de salida que tendrá cada uno de ellos:

```
{
  "query": "The Lord of The Rings",
  "results": [
    {
      "id": 1000067,
      "title": "The Lord of the Rings: The Fellowship of the Ring",
      "releaseDate": 2001,
      "runningTime": 179,
      "rating": 4.37,
```

```
      "posterPath": "https://a.ltrbxd.com/resized/sm/upload/3t/...",
      "description": "Young hobbit Frodo Baggins, after inheriting ...",
      "actors": "Billy Boyd, Cate Blanchett, Dominic Monaghan ...",
      "countries": "New Zealand, USA",
      "directors": "Peter Jackson",
      "writers": "Philippa Boyens",
      "producers": "Barrie M. Osborne, Fran Walsh, Rick Porras...",
      "composers": "Howard Shore",
      "cinematography": "Andrew Lesnie"
    }
    ,
    ...
  ]
}
```

Este formato conserva en todo momento la consulta del usuario (*query*) y la lista de resultados recuperados (*results*) por un determinado componente de búsqueda. Establece una forma de comunicación normalizada y consistente entre los distintos componentes del sistema y entre estos y el *frontend*.

3.3. Evaluación de los buscadores y estrategias de mejora

Antes de desarrollar los diferentes componentes de búsqueda, era fundamental comprender cómo evaluar su calidad. Esto permitió comparar la efectividad de distintos enfoques o configuraciones a lo largo del proceso de desarrollo. Para ello, se utilizan métricas específicas de evaluación en sistemas de IR.

3.3.1. Métricas para evaluar sistemas de IR

Estas métricas se suelen calcular en función de un parámetro k. Este parámetro define el número de resultados a evaluar, ya que muchas veces la evaluación de todos los resultados supone un proceso tedioso y costoso de mantener.

Observar un pequeño subconjunto (generalmente conocido como el top_K) de los resultados en muchas ocasiones ya supone una evaluación representativa del sistema de IR. Un

enfoque muy común y ampliamente usado es el de $k = 10$, lo que supone calcular métricas de evaluación sobre los 10 primeros resultados obtenidos.

Precisión en K

La precisión en K captura el numero de resultados relevantes que se han obtenido en los K primeros resultados.

$$\text{Precisión@}K = \frac{\text{Número de resultados relevantes en K}}{K}$$

Esta métrica se centra en lo correctos que son los resultados recuperados. Sin embargo, no captura la calidad del ranking, de forma que si los resultados más relevantes se encuentran al final del ranking, esta métrica no quedará penalizada [8].

Recall en K

El Recall en K captura el número de resultados relevantes que se han obtenido frente al total de resultados relevantes que existen para esa consulta.

$$\text{Recall@}K = \frac{\text{Número de resultados relevantes en K}}{\text{Número total de resultados relevantes}}$$

Esta métrica es muy útil ya que refleja qué proporción de resultados relevantes se han recuperado aunque, de nuevo, no captura la calidad del ranking [8].

Recall@K no se consideró para las evaluaciones en este trabajo, ya que es difícil determinar para cada búsqueda que se realiza cuál es el total de resultados del dataset que son relevantes realmente.

Promedio de precisión en K

El promedio de precisión en K (*Average Precision at K*, AP@K) se calcula como el promedio de precisión en todas las posiciones relevantes dentro de K.

$$\text{AP@}K = \frac{1}{N}\sum_{k=1}^{K}\text{Precision}(k)\cdot\text{rel}(k)$$

donde:

- *N* Es el número total de documentos relevantes para un usuario particular.
- *rel(k)* Mide la relevancia del documento en la posición k, es un valor binario donde 1 significa que el documento es relevante y 0 lo contrario.

Esta métrica sí penaliza los rankings donde los documentos relevantes no están al inicio. Esto se debe a que calcula la precisión solamente en las posiciones donde existen documentos relevantes, lo que hace que los errores en las primeras posiciones del ranking tengan un impacto acumulativo en la métrica final [8].

Media de promedios de precisión en K

La media de promedios de precisión en K (*Mean Average Precision at K*) se calcula como la media de promedios de precisión en K sobre la evaluación de diferentes usuarios o sobre diferentes búsquedas [8].

$$\text{MAP@}K = \frac{1}{U}\sum_{u=1}^{U}\text{AP@}K_u$$

donde:

- *U* Es el conjunto de evaluaciones de diferentes usuarios o el conjunto búsquedas evaluadas.

- MAP es 1 cuando todos los documentos relevantes están clasificados en las primeras posiciones del ranking, sin resultados irrelevantes antes que ellos.
- MAP es 0 cuando no hay ningún documento relevante recuperado en el top-K.
- MAP toma valores entre 0 y 1 en los demás casos, siendo mayor cuanto antes aparecen los documentos relevantes y más precisas son sus posiciones en el ranking.

Ganancia acumulada en K

La ganancia acumulada en K (*Cumulative Gain at K*, CG) es una medida de la relevancia total de un ranking. Suma las puntuaciones de relevancia sobre una lista:

$$\text{CG@}K = \sum_{k=1}^{K} rel(k)$$

Esta métrica no tiene en cuenta la calidad del ranking [8].

Ganancia descontada en K

La ganancia descontada en K (*Discounted gain at K*, DGC) introduce un descuento logarítmico, para asignar menos ganancia a los documentos que aparecen en peores posiciones del ranking [8].

$$\text{DCG@}K = \sum_{k=1}^{K} \frac{rel(k)}{log_2(i+1)}$$

Ganancia normalizada y descontada en K

La ganancia normalizada y descontada en K (*Normalized Discounted Gain at K*, NDGC@K) normaliza el DGC dividiendo por el ranking ideal (IDGC@K), en el que todos los documentos aparecen ordenados por orden de relevancia [8].

$$\text{NDCG@}K = \frac{NDGC@K}{IDGC@K}$$

- NDGC es 1 cuando los resultados están en el orden de relevancia ideal.
- NDGC es 0 cuando no hay resultados relevantes en el top-K.
- NDGC está entre 0 y 1 el, en el resto de casos. A mayor valor, mejor es el ranking.

Es importante mencionar que esta métrica admite tanto relevancia binaria como numérica: por ejemplo, 5 máxima relevancia - 1 mínima relevancia [8].

También se puede calcular el NDGC@K sobre varias evaluaciones de usuarios o búsquedas:

$$\text{NDCG@}K = \frac{1}{U}\sum_{u=1}^{U} NDGC@K$$

Rango recíproco en K

El rango recíproco en K (*Reciprocal Rank at K*, RR@K) mide cuánto tarda en aparecer un resultado relevante en el ranking.

$$\text{RR@}K = \frac{1}{\text{Posición del primer resultado relevante}}$$

- RR es 1 cuando aparece un resultado relevante en la primera posición.
- RR es 0 cuando no hay resultados relevantes en el top-K.
- RR está entre 0 y 1 el, en el resto de casos. A menor valor, más tarda en aparecer un resultado relevante en el ranking.

También se puede calcular la media de rango recíproco en K (*Mean Reciprocal Rank at K*, MRR@K) haciendo la media de RR sobre varias evaluaciones de usuarios o búsquedas [8]:

$$\text{MRR@}K = \frac{1}{U}\sum_{u=1}^{U} RR@K$$

3.3.2. Diseño de las evaluaciones

Para evaluar los diferentes buscadores, se definieron diferentes categorías de búsqueda y se hicieron evaluaciones para cada una de ellas. Estas categorías son las siguientes:

Habrá consultas en las que la importancia recaiga plenamente en la búsqueda por palabras clave, a este tipo de búsquedas las llamaremos **búsquedas exactas**:

Consulta: *"The Shinning"*

En algunos casos se requerirá en igual medida la potencia de ambos buscadores, a este tipo de búsquedas las llamaremos **búsquedas exploratorias**:

Consulta: *"Movie directed by Stanley Kubrick with axes, snow and blood"*

En otros se requerirá una comprensión contextual y semántica completa, a este tipo de búsquedas las llamaremos **búsquedas creativas**:

Consulta: *"Movie where a family have to live in a hotel during winter, and the father becomes a psychopath"*

Para cada una de esas categorías se definieron siete búsquedas diferentes a evaluar, las cuales se enumeran a continuación:

- **Búsqueda exacta:** ["Harry Potter", "Movies with Brad Pitt and Leonardo DiCaprio", ...]
- **Búsqueda exploratoria:** ["Action Films set in prison", "Space and futuristic Films", ...]
- **Búsqueda creativa:** ["Movies with melancholy tone", "Film about self-discovery", ...]

Las métricas seleccionadas fueron: **MRR@10, NDGC@10, MAP@10, tiempo de respuesta medio (TRM) y promedio del número de resultados obtenidos (PNR)**.

Evaluaciones con LLM

Había un gran problema al que hacer frente en este paso: Cómo conseguir evaluaciones para cada tipo de búsqueda y para cada conjunto de resultados obtenidos. El proceso de evaluación puede ser muy costoso y no contaba con los medios que cuentan las grandes empresas, como contratar evaluadores humanos o diseñar sistemas de evaluación y, procesos como realizar las evaluaciones de forma manual requeriría de una cantidad de tiempo muy elevada.

Para poner solución a esto, y basándonos en la idea que hay detrás de las evaluaciones de sistemas RAG (*Retrieved Augmented Generation*) [7], se automatizaron las evaluaciones usando una IA generativa como evaluador. Concretamente se hizo uso de *Gemini Flash 2.0* como evaluador. Debido a que los modelos de IA Generativa dan respuestas no deterministas y pueden presentar algo de subjetividad, y para intentar garantizar la validez de sus evaluaciones, fueron repetidas todas 10 veces, por lo que solo se consultó la media esas evaluaciones y se usó la desviación típica como medida de la fiabilidad de esas evaluaciones, valor que se adjuntó sobre cada métrica con un $\pm$.

Para usar el LLM como evaluador se siguieron las buenas prácticas recomendadas de *prompting*, proporcionando al modelo instrucciones generales de evaluación. Los *prompts* fueron diseñados para ser específicos y precisos, dejando clara la tarea a realizar, y se utilizaron en modo *zero-shot*, es decir, sin entrenamiento previo [9].

Además a las instrucciones generales del *prompt*, se le añadió una indicación extra para cada categoría de evaluación:

- **Búsqueda exacta:** Instrucciones para darle importancia a las coincidencias por palabras clave en el título, actores, géneros y equipo técnico. Las coincidencias parciales de palabras clave serán valoradas pero peor que las coincidencias exactas. La relevancia semántica se considerará un factor secundario y no deberá dar más puntuación que las

coincidencias por palabras clave.

- **Búsqueda exploratoria:** Instrucciones para que las coincidencias temáticas tengan más importancia. Los resultados deben encajar en la descripción general de la película en cuanto argumento, temática y géneros. Se recuerda que que el título incluya palabras que están en la consulta no es motivo para aumentar la puntuación de la evaluación.
- **Búsqueda creativa:** Instrucciones para darle importancia a la coincidencia conceptual. Los resultados deben capturar la esencia de la consulta. De nuevo se recuerda el hecho de que el título incluya palabras que están en la consulta no es motivo para aumentar la puntuación de la evaluación.

Con objeto de mantener consistencia en las respuestas del LLM, se hizo uso de la librería Pydantic junto con LangChain de forma que así se podían definir, a partir de modelos de pydantic, el formato las respuestas esperadas e integrarlo adecuadamente en los prompts.

Para realizar estas evaluaciones, el LLM debía asignar a las 10 primeras películas una puntuación de relevancia entre 1 y 5, en función de como se ajustan los resultados a lo buscado. La lista de evaluaciones que devolvía el LLM quedaba de la siguiente manera:

$$[\text{Película 1}, \ldots, \text{Película K}] \quad \Rightarrow \quad \textit{Evaluación del LLM} \quad \Rightarrow \quad [5, 5, 1, 3, \ldots, 5]$$

Luego solo había que calcular y guardar las métricas a partir de esas listas. Para calcular el MAP, que solo admite relevancia binaria, se consideraron relevantes los resultados con una puntuación mayor a 2.

Evaluaciones con usuarios reales

Después de realizar las evaluaciones con LLM, durante la etapa de desarrollo, se diseñó un formulario, con dos preguntas generales para poder recopilar, de forma general, las preferencias de uso en los buscadores de películas de una muestra de usuarios de aplicaciones de este tipo, y también preguntas para conseguir evaluaciones de los resultados del buscador final y poder compararlos con los obtenidos con los LLM.

Se consiguieron recopilar un total de 27 respuestas. La primera pregunta general que se realizó estaba relacionada con el número de resultados que miran los usuarios cuando hacen búsquedas. La segunda fue acerca de la importancia de que las películas estén o no evaluadas previamente.

¿Cuántos resultados sueles revisar cuando haces una búsqueda? (Nota: '10-20' significa que normalmente miras los 10 primeros resultados, y a veces puedes llegar a ver hasta 20).
27 respuestas

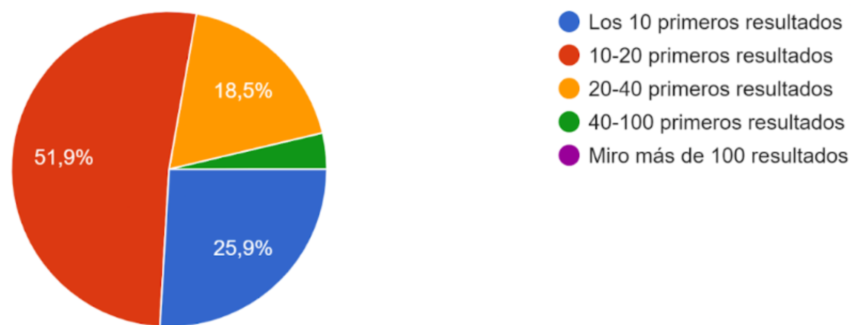

Figura 12: Número de resultados revisados por usuario

¿Es importante para ti que las películas tengan una valoración? (Nota: Por ejemplo, que veas que una película tiene una nota media de 4.3/5)
27 respuestas

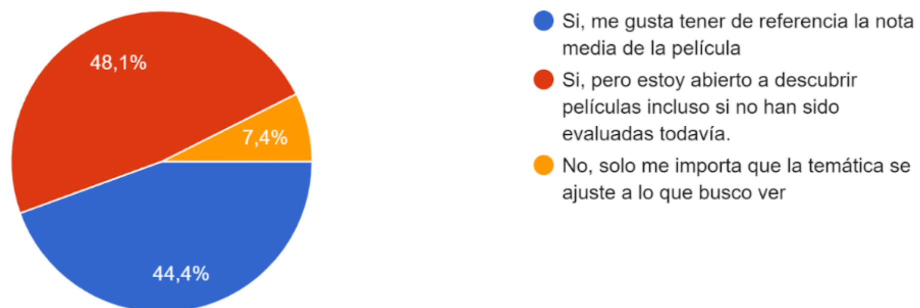

Figura 13: Importancia de las películas evaluadas por usuario

En las figuras 12 y 13 se puede observar cómo la mayoría de los usuarios rara vez miran más de 40 resultados y, por lo general, suelen preferir que las películas estén evaluadas. Esta

información que se consiguió recopilar a partir del formulario fue muy valiosa para tomar decisiones de diseño en el desarrollo de los buscadores.

3.4. Desarrollo de la búsqueda por palabras clave

Para buscar contenido textual a partir de palabras clave, PostgreSQL te ofrece los comandos *like*, *ilike* o expresiones regulares. No obstante, todos estos métodos carecen de técnicas de NLP que mejoren las condiciones de búsqueda.

Para ello PostgreSQL ofrece la Búsqueda por Texto Completo (*Full Text Search*, FTS), que ofrece un conjunto de herramientas que añaden un mínimo de "inteligencia" a las búsquedas. A pesar de que se queda lejos de comprender las consultas del usuario, es capaz de encontrar palabras que están más cerca en significado que en su escritura.

En el núcleo del FTS se encuentra la configuración FTS, que define las reglas bajo las cuales se realizarán las búsquedas y los emparejamientos. Esta configuración hace referencia a uno o varios diccionarios. En el caso del diccionario inglés que incorpora PostgreSQL, incluye reglas definidas para emparejar palabras a partir de su raíz e identificar *stop words* [29].

3.4.1. Extracción de palabras clave

FTS normaliza las columnas de texto deseadas convirtiéndolas en unos vectores llamados *ts_vectors* que reducen el texto original a un conjunto de esqueletos de palabras (palabras raíz) llamados lexemas. Cada lexema es almacenado con la posición en el texto y su importancia.

Este proceso concretamente consiste en tokenizar el texto en palabras diferentes, convertirlas a minúsculas, aplicar *stemming* y eliminar *stop words*.

Un *ts_vector* no es más que una lista ordenada de distintos lexemas, en la que cada uno de estos va seguido de su posición en el texto y su peso. La sintaxis general es la siguiente:

[‘palabra1’:1D,3B ‘palabra2’:2A ...]

Los números indican la posición de la palabra en el texto y las letras (A, B, C, D) indican la importancia de esa palabra en la búsqueda [23].

3.4.2. Técnicas de ranking y ponderación

Para clasificar los resultados, FTS incluye funciones de ranking para dar puntuación a los resultados: *ts_rank* y *ts_rank_cd*, basadas en el conocido algoritmo *Okapi BM25*. Ambas consideran la frecuencia de los lexemas y los pesos para la ponderación. *ts_rank_cd*, además, tiene en cuenta la posición del lexema buscado en el texto de forma que si los lexemas se encuentran más próximos entre sí, el resultado quedará mejor clasificado.

Estas funciones de ranking además tienen en cuenta la ponderación asignada con las etiquetas D, C, B y A, que son, respectivamente: 0.1, 0.2, 0.4 y 1.0.

A partir de estas funciones es posible definir funciones de ranking personalizadas, componiéndolas, normalizándolas o incorporando factores externos.

La puntuación asignada por estas funciones de ranking puede ser normalizada, por ejemplo, en función del tamaño del documento. Un documento que contenga el lexema buscado debería tener una mayor puntuación de relevancia si contiene 5 palabras que si contiene 500.

Estas normalizaciones se pueden concatenar entre ellas; entre estas destacan las siguientes [23]:

- **Normalización 1:** divide la puntuación por $1 + \log(\text{tamaño del documento})$, penalizando los documentos con muchas palabras.
- **Normalización 2:** divide la puntuación por el tamaño del documento. Penaliza los documentos largos de forma más estricta que la normalización 1.
- **Normalización 4:** divide la puntuación por la media armónica de las distancias entre las ocurrencias de los lexemas (solo válido con *ts_rank_cd*). Penaliza cuando los lexemas aparecen dispersos.

- **Normalización 32:** transforma la puntuación aplicando $\frac{\text{rank}}{\text{rank}+1}$, escalándola al rango $[0, 1)$. Es una transformación cosmética que no cambia el orden de los resultados.

3.4.3. Funciones de búsqueda

Para las búsquedas se usa *tsquery* que se construye de forma análoga al *ts_vector* pero además permite componer *búsquedas booleanas* además de otras funcionalidades y personalizaciones del texto.

PostgreSQL amplía la búsqueda booleana con operadores como el de proximidad <->, que da prioridad a los documentos donde los términos aparecen más cerca. Permite definir la distancia entre palabras: <-> indica que deben estar juntas, *<2>* que puede haber una palabra entre medias, y así sucesivamente.

Esta *tsquery* se compara con los diferentes *ts_vector* usando el operador @@, devolviendo un booleano que indica si coinciden o no. Posteriormente, a partir del conjunto de documentos que coinciden, se aplica la función de ranking y se reordenan obteniendo el resultado final de la búsqueda.

Para convertir el texto de entrada del usuario a *tsquery*, FTS ofrece diferentes funciones [23]:

- **phraseto_tsquery:** Trata la consulta del usuario para buscarla como una frase exacta. Para ello normaliza la entrada del usuario a partir de la configuración FTS, elimina los signos de puntuación y pone operadores de proximidad entre palabras clave.
- **plainto_tsquery:** Normaliza la entrada del usuario a partir de la configuración FTS, elimina los signos de puntuación y separa las palabras clave con el operador booleano AND (&).
- **websearch_to_tsquery:** Normaliza la entrada del usuario a partir de la configuración FTS, elimina los signos de puntuación y separa las palabras clave intentando imitar el comportamiento de algunas herramientas de búsqueda web. Por ejemplo, el texto "or" lo cambia por el operador booleano OR (|) , "-" por el operador booleano NOT (!), etc.

- **to_tsquery:** Normaliza la entrada del usuario a partir de la configuración FTS, espera que las palabras ya vengan separadas por operadores booleanos válidos.

3.4.4. Indexación y optimización

FTS usa concretamente GIN (*General Inverted Index*). Tiene un índice invertido compuesto por un diccionario de lexemas donde la clave corresponde a un lexema y su valor es una lista de postings que indica los documentos donde aparece esa palabra.

Para mejorar el rendimiento de inserciones, PostgreSQL mantiene una lista de pendientes (*pending list*), que acumula temporalmente las entradas antes de insertarlas en el índice principal. Esto reduce el coste inmediato de las escrituras, aunque el mantenimiento del índice sigue siendo relativamente costoso.

Para acelerar el acceso al índice, GIN utiliza internamente estructuras similares a B-trees en sus páginas, pero el índice en sí no es un B-tree tradicional (figura 14).

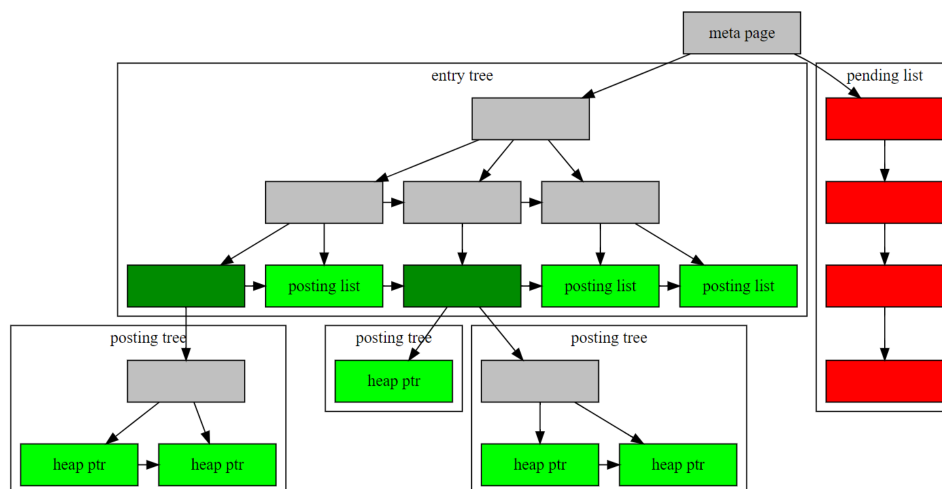

Figura 14: Interior de un índice GIN

La implementación de estos índices nos permiten crear un índice invertido para FTS, que a pesar de ser costoso de actualizar, puede mejorar la velocidad de las consultas de minutos a mili segundos [24].

PostgreSQL también admite los índices GIST (*Generalized Search Tree*), pero estos no han sido considerados ya que suelen ser mejor para datos dinámicos, siendo, por el contrario, algo más lentos en las búsquedas) [22].

3.4.5. Detalles de la implementación

Al implementar FTS nos encontramos con un problema: los metadatos sobre los que se quiere buscar se encontraban en tablas separadas por una relación de muchos a muchos. Para resolverlo, se encontraron dos soluciones: crear un índice independiente por cada tabla o crear un índice compuesto.

Dado que el objetivo es realizar siempre un tipo de búsqueda (la de películas), la opción más simple y eficiente era la creación de un índice compuesto. Para ello se recurrió a la creación de una vista materializada. En esta se incluye el id de la película y únicamente los metadatos necesarios para realizar la búsqueda; estos son:

- **Metadatos de búsqueda:** nombre, género, descripción, país, actores, equipo técnico.
- **Metadatos de filtrado:** puntuación, fecha de lanzamiento y duración.

Los metadatos de filtrado no se utilizan directamente en la búsqueda, sino que se usarán como filtros adicionales a la búsqueda.

Durante la creación del índice GIN se debía construir un *ts_vector* a partir de la concatenación de los *ts_vector* de cada metadato de búsqueda. Al construir estos *ts_vectors*, además se especificó la siguiente ponderación:

- **Ponderación A:** Nombre.
- **Ponderación B:** Género, actores, equipo técnico.
- **Ponderación D:** Descripción.

De esta forma, la búsqueda por palabras clave dará más valor al título de las películas y algo menos al resto de los metadatos, dejando el metadato descripción como el menos relevante, ya que ese metadato no es el punto fuerte de la búsqueda por palabras clave y puede generar mucho ruido en las búsquedas.

La función de ranking que se empleó es la siguiente:

$$\text{Rank}(d, q) = 0{,}7 \cdot \text{ts_rank_cd}(d, q, 32) + 0{,}3 \cdot \frac{\text{COALESCE}(\text{rating}(d), 0)}{5{,}0}$$

donde:

- *d* es un documento,
- *q* es una consulta del usuario,
- *t_rank_cd(d,q,32)* es la función de ranking de FTS con normalización 32,
- *COALESCE(rating(d), 0)* es la puntuación de la película en el dataset, 0 si falta el valor.

De este modo, las puntuaciones están a la misma escala (escala [0,1)] y son comparables, además, se valora positivamente la calificación media de las películas. La función de búsqueda que se ha usado es *to_tsquery*, ya que es la que más flexibilidad nos da al poner operadores booleanos donde queramos, además de que el resto de funciones no se ajustaban bien a nuestro caso de uso.

El formato que se elegió para las *tsquery* es separar las palabras clave con el operador OR (|), ya que si usamos el operador AND (&), a pesar de ser la implementación más común, sufrirá mucho con las consultas de cola larga, reduciendo drásticamente el número de resultados obtenidos. Nuestro objetivo precisamente es obtener un buen número de películas que estén ordenadas por relevancia, de forma que puedan aportar valor a la búsqueda híbrida. Por lo tanto, el formato que se ha usado para las consultas de usuario se ve de la siguiente forma:

Entrada: *"Harry Potter"* $\Rightarrow$ **Salida:** *"Harry | Potter"*

Dado que nuestro buscador va a permitir buscar películas usando lenguaje natural, será muy común que los usuarios escriban textos como “Películas dirigidas por X”, “Películas de Acción en las que actúe X”, etc. Hay palabras en estas descripciones naturales de las películas que no aportan información relevante en la búsqueda por palabras clave como son “Películas, Dirigida, etc.”. Para evitar que generen ruido, se han definido unas *stop words* personalizadas para nuestro dominio, comprobando previamente en IMDB que cada palabra seleccionada para ser *stop word* no fuera el título de una película relevante. Entre las *stop words* definidas están:

Stop words de películas: {“movie”, “film”, “motion”, “genre”, “cinematography”, “directed”, ...}

Es importante recalcar que estas *stop words* se aplican solo a esta búsqueda por palabras clave, ya que estas palabras sí serán importantes para la búsqueda semántica e híbrida.

Las búsquedas por palabras clave se evaluaron en la categoría de *búsquedas exactas*, ya que es la más representativa para este buscador.

MRR@10	**MAP@10**	**NDCG@10**	**TRM**	**PNR**
$0{,}5238 \pm 0{,}0$	$0{,}5092 \pm 0{,}011$	$0{,}8283 \pm 0{,}009$	1.3151	500.0

Cuadro 5: Primera evaluación del buscador de palabras clave

El diseño inicial de este componente de búsqueda presentó las métricas que aparecen en la tabla 5. Se puede observar que ordena bien los resultados relevantes (82 % de NDGC). Sin embargo, tiene una precisión del 50 % y no ofrece resultados relevantes hasta la posición 5 (52 % de MRR). El resultado obtenido era muy mejorable, por ello se revisaron los resultados de búsqueda y se observó lo siguiente:

Al buscar “Harry Potter” aparecían una gran cantidad de resultados que presentaban con mucha frecuencia la palabra “Harry” por separado y lo mismo con “Potter”. Sin embargo, el usuario quiere resultados que contengan ambas palabras juntas.

Para asegurarnos de que ambas palabras estén juntas se modificó el formato de las *tsquery* para poner el operador de proximidad <-> entre los nombres propios. Para detectar estos nombres propios se usó el modelo *en_core_web_sm*, un modelo ligero de Spacy para hacer NER en inglés. El nuevo formato de *tsquery* queda de la siguiente manera:

Entrada: *"Harry Potter and magic"* $\Rightarrow$ **Salida:** *"(Harry<->Potter) | and | magic"*

MRR@10	**MAP@10**	**NDCG@10**	**TRM**	**PNR**
$0,8333 \pm 0,0$	$0,8603 \pm 0,013$	$0,9344 \pm 0,005$	0.2056	278.42

Cuadro 6: Evaluación del buscador por palabras clave tras aplicar NER con <−> como separador de nombres propios

Observando las métricas obtenidas (tabla 6) es notable que tras incorporar un operador más limitante que OR, como es <->, se reduce la cantidad de resultados a 278.42. Además se ha reducido bastante el tiempo de respuesta y mejorado mucho las métricas de calidad. Otra prueba que se realizó fue cambiar la ponderación para que los metadatos tuvieran peso A y solo la descripción tuviera peso D:

- **Ponderación A:** Nombre, género, actores, equipo técnico.
- **Ponderación D:** descripción

MRR@10	**MAP@10**	**NDCG@10**	**TRM**	**PNR**
$0,9591 \pm 0,046$	$0,9326 \pm 0,038$	$0,9527 \pm 0,003$	0.2184	278.42

Cuadro 7: Evaluación del buscador por palabras clave tras aplicar NER con <−> como separador de nombres propios y nuevas ponderaciones

Este cambio mejoró aún más las métricas de calidad (tabla 7), lo que implica que darle la puntuación máxima de relevancia a los metadatos de búsqueda y la mínima a la descripción trae resultados más relevantes.

Antes de quedarnos con esta versión del buscador, se realizaron unas últimas pruebas sobre este. En primer lugar, se añadieron las normalizaciones 1 y 4, con el objetivo de comprobar si la normalización por tamaño de documento y el uso de la media armónica de la distancia entre palabras contribuyen a recuperar mejores resultados en nuestro caso de uso. Posteriormente, se sustituyó el operador de proximidad <-> por &:

Prueba	MRR@10	MAP@10	NDCG@10	TRM	PNR
Normalización 1	$0{,}9286 \pm 0{,}0$	$0{,}9288 \pm 0{,}027$	$0{,}9465 \pm 0{,}005$	0.2114	92.29
Normalización 4	$0{,}9286 \pm 0{,}0$	$0{,}9281 \pm 0{,}028$	$0{,}9486 \pm 0{,}006$	0.1966	92.29
Normalización 1 | 4	$0{,}9286 \pm 0{,}0$	$0{,}9107 \pm 0{,}036$	$0{,}9467 \pm 0{,}008$	0.2089	92.29
Separador &	$0{,}9327 \pm 0{,}071$	$0{,}8818 \pm 0{,}053$	$0{,}9615 \pm 0{,}004$	0.2236	307.14

Cuadro 8: Evaluación del buscador por palabras clave con distintas configuraciones de normalización y usando & como separador de nombres propios

A la vista de los resultados (Tabla 8), podemos concluir que estas pruebas no mejoraban la versión anterior del buscador.

Una vez que ha quedado definido el diseño del componente de búsqueda por palabras clave y hemos creado el índice GIN, el componente quedó completo y funcional (figura 15), devolviendo el formato de salida JSON definido anteriormente.

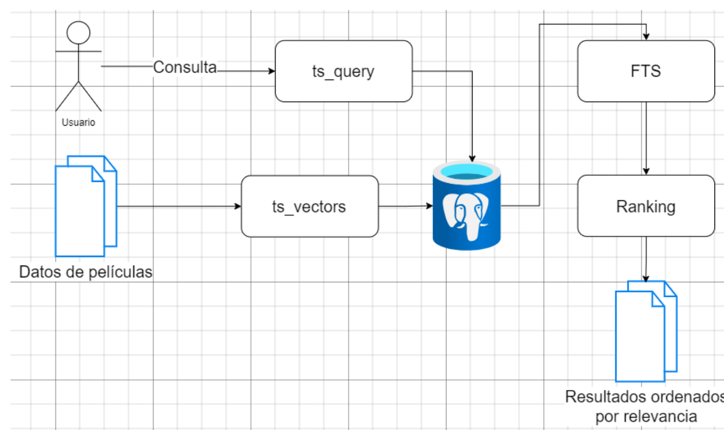

Figura 15: Diagrama de flujo del componente de búsqueda por palabras clave.

3.5. Desarrollo de la búsqueda semántica

El próximo paso era crear y almacenar los embeddings de los documentos para poder computar adecuadamente la búsqueda semántica. Es necesario buscar un LLM que esté entrenado para la tarea de IR y que se ajuste al tamaño y dominio de nuestros datos.

3.5.1. Selección del modelo y almacenamiento de embeddings

Para buscar el modelo adecuado se recurrió al *MTEB* (*Massive Text Embedding Benchmark*) [27], una herramienta de *HuggingFace*, empresa referente de la comunidad de IA, que ofrece una manera estandarizada de evaluar y clasificar los modelos de embeddings para diversas tareas, idiomas, etc. Para buscar el modelo más adecuado se accedió a la clasificación de modelos y se filtraron aquellas más relacionadas con la IR en inglés. Los modelos encontrados aparecían ordenados a partir de los siguientes datos:

- Zero-shot (% de rendimiento del modelo sin haber sido entrenado con ejemplos para tareas específicas).
- Tamaño del modelo (Millones de parámetros).
- Uso de memoria (MB).
- Dimensión de los embeddings generados.
- Máximo de tokens de entrada soportados.
- Métricas de desempeño generales en diferentes tareas (Clasificación, clustering, IR, etc.).

Estos embeddings se crearon a partir de las descripciones de las películas. Para determinar cuantos tokens de entrada debía soportar el modelo, se analizó la distribución del número de palabras en la descripción de las películas (figura 16).

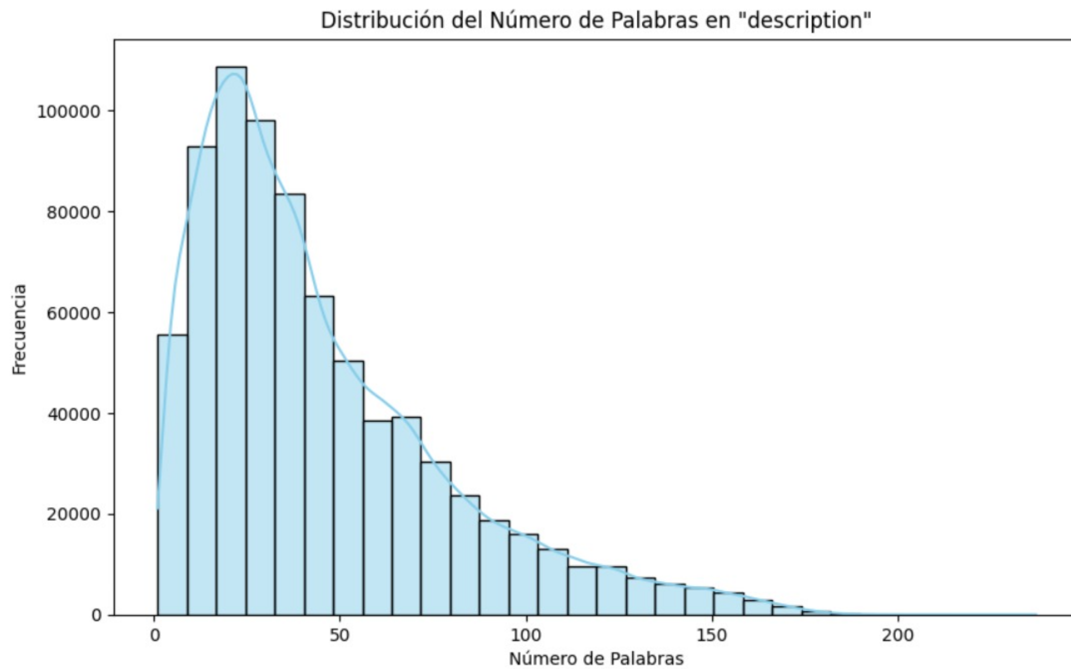

Figura 16: Distribución del número de palabras en el campo description.

Para saber cuántos tokens son esas palabras en inglés, se siguió la siguiente regla general de conversión [4]:

$$1 \text{ token} \approx \frac{3}{4} \text{ palabras}$$

Para 300 palabras:

$$\text{número de tokens} \approx \frac{4}{3} \times 250 \text{ palabras} = 333{,}33 \text{ tokens}$$

Dado que la mayoría de los modelos que aparecen en el MTEB soportan por lo menos 512 tokens de entrada y nuestros documentos tenían como máximo y en casos aislados 333 tokens, no hubo problemas con el tamaño de los documentos y no fue necesario recurrir a técnicas de chunking.

Para tratar de hacer la búsqueda semántica lo más rápida posible, se intentó buscar un equilibrio entre modelos ligeros, que a la vez se encuentren en una posición alta en la clasi-

ficación y tuvieran un *zero-shot* alto para garantizar un buen funcionamiento en el dominio de las películas, sin entrenamiento previo del modelo. Por esta razón, se eligieron los modelos que aparecen en la tabla 9

Posición en el ranking	Modelo	Zero-shot	Uso de Memoria	Tamaño	Dimensión	Máx. Tokens
28	snowflake-arctic-embed-l	100 %	1274MB	335M	1024	512
34	snowflake-arctic-embed-m-v1.5	100 %	415MB	109M	768	512

Cuadro 9: Datos de los modelos elegidos (según MTEB).

Se seleccionaron dos modelos de tamaño medio, uno con embeddings de 1024 dimensiones y otro de 768, con el objetivo de evaluar si realmente el incremento en la dimensión conlleva una mejora significativa en la precisión. Esta comparación permite valorar si el mayor coste de modelos más grandes se justifica frente a alternativas más ligeras para resolver este problema.

Otra comparación que se realizó y que observaremos posteriormente, es la inclusión de los metadatos en el cómputo de embeddings, de forma que se compruebe adecuadamente la necesidad del componente de búsqueda por palabras clave, para efectuar búsquedas exactas. Para ello se unificaron los metadatos en una columna de texto que describía la película de forma natural:

- **Sin metadatos:** *"Argumento"* $\Rightarrow$ *embedding*
- **Con metadatos:** "La película X va de *Argumento*, los géneros cinematográficos son *Géneros*, etc." $\Rightarrow$ *embedding*

No obstante, al crear estas columnas de texto se observó un problema: la distribución de palabras presentaba valores atípicos muy pronunciados, llegando a haber documentos de hasta 500 palabras (figura 17). Esto supone que habrá documentos que superen el límite de 512 tokens de los modelos.

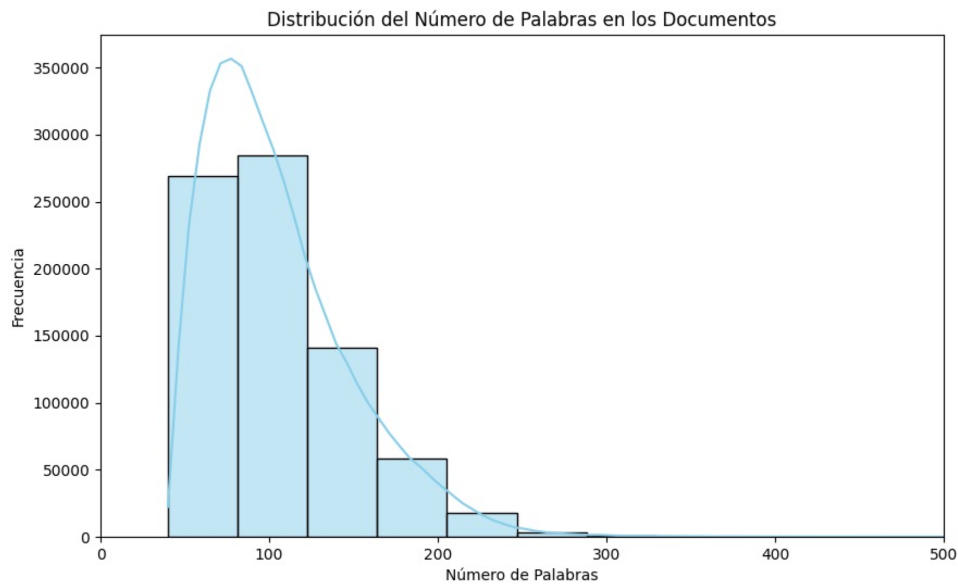

Figura 17: Distribución del número de palabras en los documentos con metadatos.

Se comprobó que esto se debía a que los campos de equipo técnico y actores incluían un gran número de datos por película en varias ocasiones. La aparición de una gran cantidad de nombres de personas puede llegar a generar ruido en las búsquedas por palabras clave. Por lo tanto, además de reducir el numero de tokens de entrada de los embeddings, al recortar esos metadatos, se consiguió mejorar el desempeño de las búsquedas por palabras clave.

Aprovechando que, como vimos anteriormente, esos metadatos ya vienen ordenados por orden descendente de relevancia y, además simulando el comportamiento de los usuarios que identifican las películas a partir de los actores y miembros del equipo técnico mas importantes, se tomaron las siguientes decisiones de diseño:

- Reducción de actores por película a los 10 primeros que aparecen.
- Reducción del equipo técnico por película a los 5 primeros que aparecen por rol (director, productor, etc.)

Se puede observar que tras este recorte, los valores atípicos se estabilizan, llegando a haber documentos de hasta 300 palabras (figura 18).

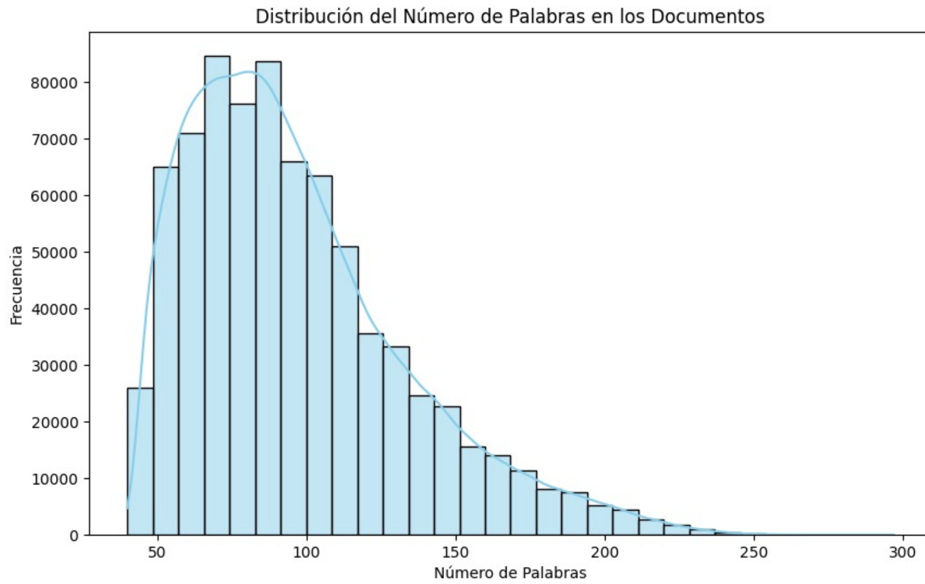

Figura 18: Distribución del número de palabras en los documentos con metadatos recortados.

$$\text{número de tokens} \approx \frac{4}{3} \times 300 \text{ palabras} = 400 \text{ tokens}$$

Esta cantidad ya era admisible para nuestros modelos. Una vez seleccionado los modelos y embeddings a producir, quedaba computarlos. Para ello, se utilizó un servidor especializado para IA con dos GPU del laboratorio del grupo de Investigación y Aplicaciones en Inteligencia Artificial de la E.T.S. de Ingeniería Informática. Gracias a este servidor, se pudo crear un Jupyter Notebook para computar los diferentes embeddings usando la librería Sentence Transformers. Tras tan solo unas cuantas horas de cómputo, se obtuvieron los embeddings de tamaño 1024 y los de tamaño 768 con y sin metadatos.

Para cada tipo de embedding se generó un CSV con las columnas [*movie_id, embedding*]. Usando la extensión *pgvector* de PostgreSQL se pudieron crear nuevas tablas para cada tipo de embedding e importar los CSV de datos (figura 19).

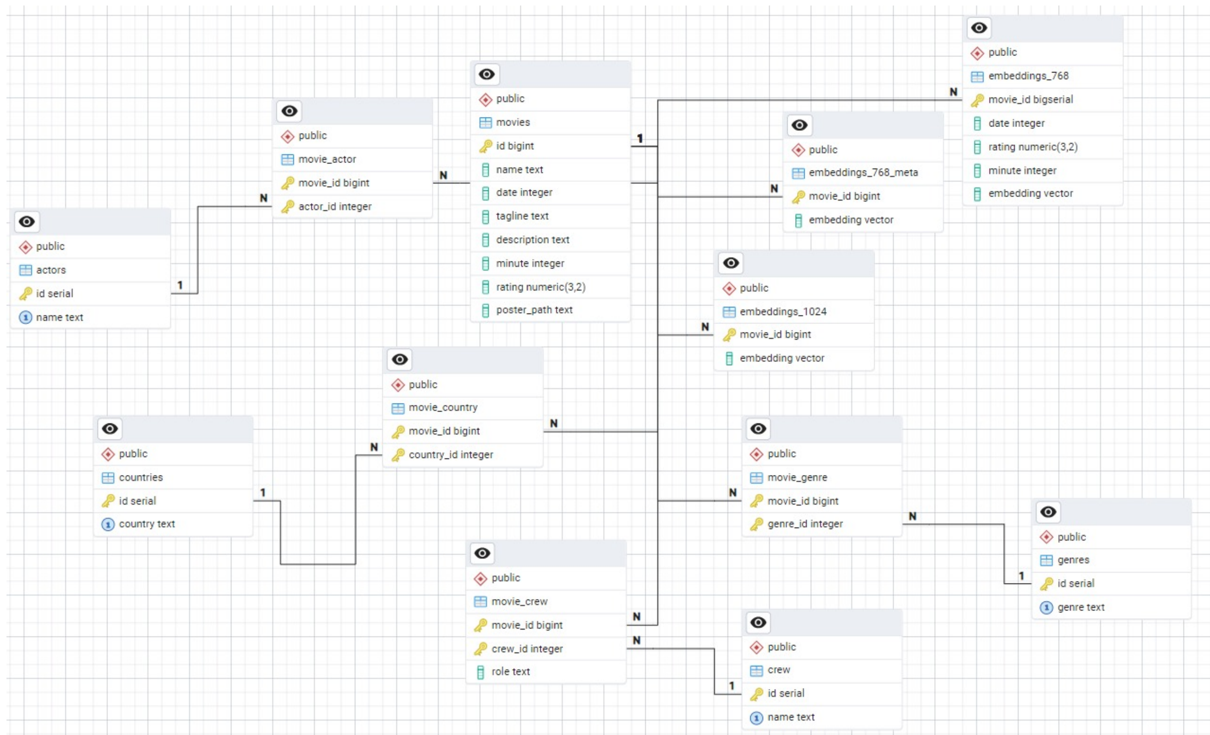

Figura 19: Diseño de la base de datos con embeddings.

3.5.2. Indexación y optimización

Pgvector ofrece dos métodos de indexación de los vectores [5]:

- **HNSW (Hierarchical Navigable Small Worlds):** Construye un grafo multicapa, tiene tiempos de construcción más lentos y usa más memoria que IVVFLAT, pero ofrece un mejor rendimiento en cuanto al equilibrio entre *recall* y velocidad.

- **IVVFLAT (Inverted File with Flat Compression):** Divide el espacio vectorial en listas y busca el subconjunto de listas más cercanas al embedding de la consulta. Estos índices se construyen más rápidamente, pero ofrecen un peor rendimiento en cuanto al equilibrio entre *recall* y velocidad.

Se eligió la creación de índices HNSW, ya que al no tener más de 1M de datos en el dataset y estar compuestos por documentos relativamente pequeños en comparación con otros casos de uso, podía merecer la pena sacrificar algo de tiempo de construcción a cambio de mejor

precisión en las búsquedas. Además, según Pinecone, proveedor líder de bases de datos vectoriales, *los grafos HNSW se encuentran entre los índices con mejor desempeño para las búsquedas semánticas*[2].

Este índice compone un algoritmo de búsqueda ANN basado en grafos de proximidad. En este tipo de grafos los nodos se enlazan en función de su cercanía. Para entender cómo funciona este índice vamos a ver brevemente sus fundamentos.

Lista de probabilidad de omisión

El concepto de lista de probabilidad de omisión (*Probability Skip List*) fue introducido en 1990 por *William Pugh*. Este permite la búsqueda rápida en un array ordenado, mientras usa una lista enlazada para una inserción más rápida de elementos.

Las skip lists funcionan construyendo una jerarquía de capas de listas enlazadas. En la primera se observan enlaces que *omiten* muchos de los nodos intermedios. A medida que bajas en las capas, el número de *omisiones* de cada enlace disminuye.

Para buscar en las listas de omisión se empieza en la primera capa y se busca el nodo correcto (el de menor valor al buscado). Si un nodo es mayor al buscado ya sabemos que nos hemos pasado y debemos retroceder al nodo anterior (figura 20). HNSW hereda el mismo formato en capas[2].

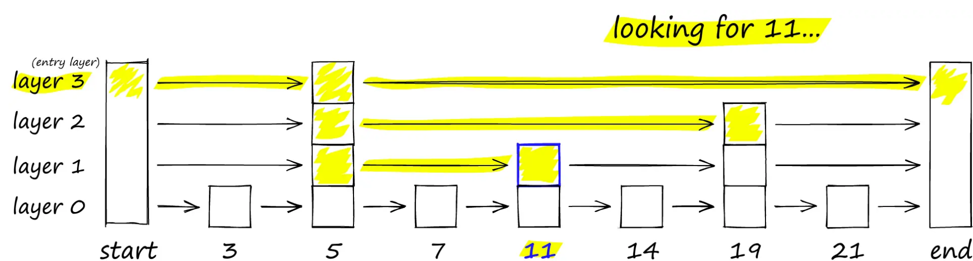

Figura 20: Ejemplo de probability skip list. Imagen obtenida de Pinecone.io

Navigable small worlds

Hierarchical Navigable Small Worlds (NSW) es un tipo de búsqueda vectorial que se introdujo en diferentes artículos publicados en 2011. Este construye los grafos de proximidad con enlaces de largo y corto alcance, permitiendo que los tiempos de búsqueda se reducen a una complejidad logarítmica o polilogarítmica.

Al empezar la búsqueda se comienza por un punto de entrada pre-definido. Este está conectado con varios nodos cercanos. Se identifica cuál de estos nodos está más cerca de la consulta del usuario y se desplaza ahí. Este proceso de búsqueda avariciosa (*greedy*) se repite hasta que estemos en un nodo cuyo nodo más cercano a la consulta sea el propio nodo de consulta. Ese nodo previo será el mínimo local y la *condición de parada* (figura 21).

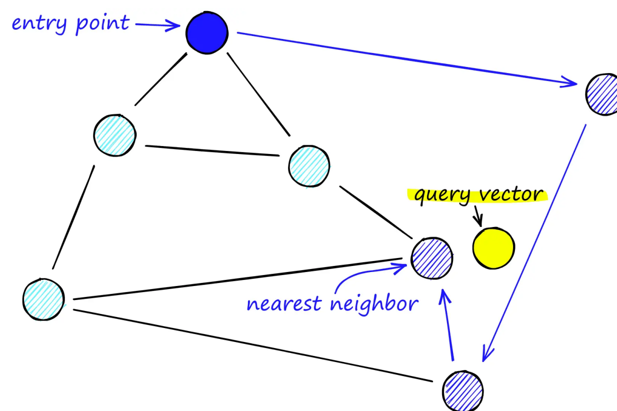

Figura 21: Ejemplo de NSW. Imagen obtenida de Pinecone.io

La ruta que se sigue por el grafo se compone de dos fases. El "zoom-out" en el que pasamos por nodos de menor grado (el grado es el número de conexiones del nodo) y el "zoom-in" donde se atraviesan nodos de mayor grado.

Debido a la condición de parada, es posible encontrar un mínimo local y parar prematuramente en la fase de zoom-out. Para minimizar esto se puede aumentar el grado medio de los nodos, pero esto también aumentará la complejidad del grafo.

Otro enfoque es empezar directamente en nodos de mayor grado, esto es esencial para HNSW[2].

Hierarchical Navigable small worlds

HNSW es una evolución de NSW que incluye el concepto de la jerarquía de capas de Pugh, de manera que en las capas superiores los enlaces son más lejanos y en las inferiores son más cercanos.

De esta forma, el punto de entrada se encuentra en la primera capa, dando inicio a la fase de zoom-out. Posteriormente se busca el mínimo local siguiendo la estrategia greedy descrita anteriormente y se pasa al mismo nodo encontrado en la capa inferior. Se continúa así sucesivamente hasta encontrar al mínimo local de la última capa (figura 22) [2].

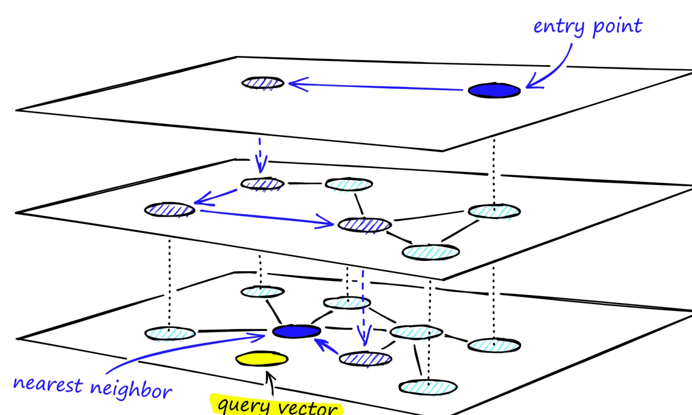

Figura 22: Ejemplo de HNSW. Imagen obtenida de Pinecone.io

3.5.3. Detalles de la implementación

Una vez se almacenaron los datos, ya se podía implementar la búsqueda semántica. Para ello las consultas del usuario se convierten a embeddings usando el método *encode* de *Sentence Transformers* y poniendo la etiqueta *prompt_name="query"* para identificar este embedding como de consulta.

De los diferentes operadores de similitud que ofrece pgvector, se eligió el de la distancia coseno (<=>), ya que es el más utilizado para este tipo de tareas y el que ha demostrado ofrecer mejores resultados en IR. Con esto ya se podían hacer búsquedas semántica por fuerza bruta. No obstante ya que buscamos ofrecer la mejor experiencia al usuario, vamos a optimizar esta

búsqueda con un índice HNSW para reducir el tiempo de respuesta de esta búsqueda. Para construir un índice HNSW pgvector te deja configurar los siguientes parámetros:

- **m:** Define el grado máximo que pueden tener los nodos del grafo. Un mayor valor de *m* incrementa la precisión de la búsqueda, aunque también aumenta la complejidad del grafo y, en consecuencia el consumo de memoria y el coste computacional.
- **ef_construction:** Define el tamaño de la lista dinámica de nodos candidatos que se consideran en cada paso de la construcción del grafo. Cuanto mayor sea su valor, más exhaustiva será la exploración del espacio de búsqueda para establecer las conexiones, lo que da lugar a un grafo de mayor calidad y, por tanto, a búsquedas más precisas. Sin embargo, también incrementa significativamente el tiempo necesario para construir el índice.

Con respecto a la construcción del índice se hicieron varias pruebas, concluyendo con los parámetros *m=32* y *ef_construction=64*, ya que son los mejores parámetros de construcción que fue posible ajustar sin sacrificar demasiado tiempo de construcción (tabla 10).

m	ef_construction	Tiempo de construcción	Observaciones
16	64	30 min	–
32	64	1 h	–
>32	–	>3 h	Tiempo de construcción excesivo
–	>64	>3 h	Tiempo de construcción excesivo

Cuadro 10: Comparativa del tiempo de construcción del índice HNSW a partir de los parámetros de construcción

Las evaluaciones para la búsqueda semántica se decidió hacerlas sobre el desempeño medio en todas las categorías, con el fin de reflejar cómo funciona la semántica en cada una de ellas. La primera evaluación realizada fue la comparación la búsqueda semántica con embeddings de tamaño 768 haciendo búsqueda usando fuerza bruta y con índice:

Modelo	MRR@10	MAP@10	NDCG@10	TRM	PNR
Fuerza bruta	$0{,}7130 \pm 0{,}011$	$0{,}7257 \pm 0{,}015$	$0{,}8876 \pm 0{,}005$	2.258	500.0
Con índice	$0{,}6683 \pm 0{,}018$	$0{,}7073 \pm 0{,}016$	$0{,}9256 \pm 0{,}007$	0.1135	40.0

Cuadro 11: Evaluación del buscador semántico con y sin índice HNSW

Viendo los resultados (tabla 11) se puede apreciar como el tiempo de respuesta medio se redujo de 2.25 a 0.11 segundos. El promedio de número de resultados se reduce ya que *ef_search* es por defecto 40. Usar el índice ha reducido el MAP y el MRR, y ha aumentado el NDCG levemente, por lo que usar el índice no afecta en gran medida a la precisión y calidad de ranking.

Posteriormente, para ver la diferencia entre hacer uso de un modelo más pesado, se realizó una comparación mediantebúsqueda por fuerza bruta entre el modelo de embeddings de tamaño 768 y el de 1024.

Modelo	MRR@10	MAP@10	NDCG@10	TRM	PNR
Embedddings 1024	$0{,}8368 \pm 0{,}0025$	$0{,}7749 \pm 0{,}0106$	$0{,}9103 \pm 0{,}0073$	2.3875	500.0
Embeddings 768	$0{,}7130 \pm 0{,}0115$	$0{,}7257 \pm 0{,}0152$	$0{,}8876 \pm 0{,}0046$	2.2575	500.0

Cuadro 12: Resultados del buscador semántico con embeddings de dimensión 1024 y 768

En la tabla 12 se puede observar una leve mejora de las métricas y el tiempo de respuesta medio apenas se ha visto afectado. Sin embargo, el cómputo de estos embeddings y la indexación de estos eran más del doble de costoso de computar, además de requerir un mayor consumo de memoria para su almacenamiento. Por ello se optó por mantener el modelo de embeddings de tamaño 768 para el buscador.

Una vez elegido el tamaño del modelo de embeddings, se realizó una evaluación en la categoría de búsquedas exactas para verificar si los embeddings se pueden beneficiar o no del uso de los metadatos:

Modelo	MRR@10	MAP@10	NDCG@10	TRM	PNR
Buscador palabras clave	$0{,}9591 \pm 0{,}046$	$0{,}9326 \pm 0{,}038$	$0{,}9527 \pm 0{,}003$	0.2184	278.42
Embeddings 768 + metadatos	$0{,}8643 \pm 0{,}023$	$0{,}8024 \pm 0{,}025$	$0{,}9394 \pm 0{,}003$	2.7399	500.0
Embeddings 768	$0{,}5762 \pm 0{,}0$	$0{,}5762 \pm 0{,}005$	$0{,}8589 \pm 0{,}005$	2.4212	500.0

Cuadro 13: Resultados del buscador semántico con metadatos frente al buscador de palabras clave y al modelo bruto en búsquedas exactas

Es cierto que la inclusión de los metadatos a los embeddings ha mejorado las métricas en este tipo de búsquedas (tabla 13). Sin embargo, las búsquedas por palabras clave siguen ofreciendo mejores métricas y, en cualquier caso, será más eficiente y fácil de mantener que los embeddings con metadatos. Por esta razón, se decidió quedarse con el buscador semántico sin metadatos.

Dado que, por defecto, el índice HNSW tiene *ef_search = 40*, siempre devuelve 40 resultados. A mayor *ef_search* se recuperan más resultados relevantes a costa de un mayor coste computacional. Esto se puso a prueba y los resultados se muestran en la tabla 14:

Modelo	MRR@10	MAP@10	NDCG@10	TRM	PNR
HNSW (ef = 40)	$0{,}6683 \pm 0{,}0176$	$0{,}7073 \pm 0{,}0159$	$0{,}9256 \pm 0{,}0066$	0.1135	40.0
HNSW (ef = 100)	$0{,}7222 \pm 0{,}0232$	$0{,}7444 \pm 0{,}0212$	$0{,}9190 \pm 0{,}0079$	0.1851	100.0
HNSW (ef = 200)	$0{,}7502 \pm 0{,}0226$	$0{,}7446 \pm 0{,}0186$	$0{,}9173 \pm 0{,}0047$	0.2839	200.0

Cuadro 14: Resultados del buscador semántico con índice HNSW para diferentes valores de *ef*

Se puede observar que efectivamente a mayor valor de *ef_search* se traduce en un mayor desempeño general y más resultados se obtienen, a coste de mayor tiempo de búsqueda. El punto óptimo parece *ef = 100* ya que a partir de este la mejora de métricas no es tan apreciable.

Una vez se definió el diseño del componente de búsqueda semántica y creado el índice HNSW, el componente quedó completo y funcional (figura 23), devolviendo siempre el conjunto de resultados ordenados en formato JSON.

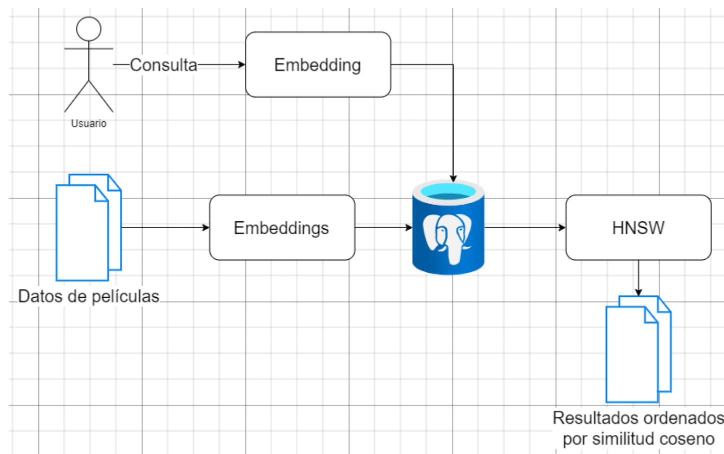

Figura 23: Diagrama de flujo del componente de búsqueda por palabras clave.

Tras finalizar la implementación de los componentes de búsqueda por palabras clave y búsqueda semántica, se puede observar una comparación entre ambos enfoques en la figura 24. En ella se muestran tres gráficas de tipo radar, cada uno correspondiente a un tipo de búsqueda diferente: exacta, exploratoria y creativa. Cada gráfico representa los valores de tres métricas de evaluación —MAP@10, MRR@10 y NDCG@10— que permiten comparar el rendimiento del buscador por palabras clave (en azul) frente al buscador semántico (en rojo).

Se puede observar como en las búsquedas exactas destaca el buscador por palabras clave, y en las creativas, el buscador semántico, mientras que en las exploratoria están más empatados, al ser un tipo de búsqueda que aprovecha la fortaleza de ambas búsquedas.

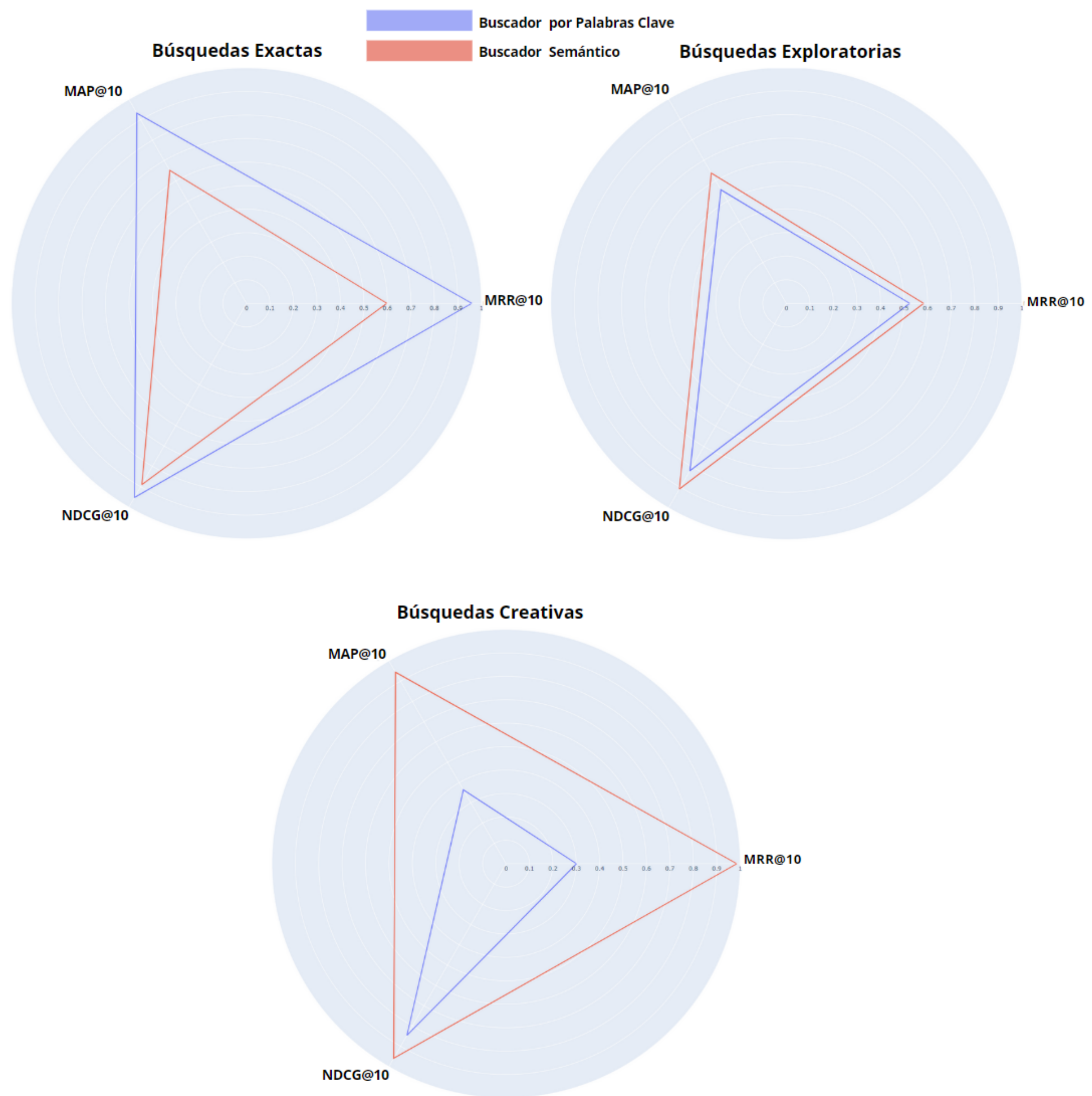

Figura 24: Desempeño de las búsquedas semántica y por palabras clave por categoría de búsqueda.

3.6. Desarrollo de la búsqueda híbrida

Aprovechando las capacidades de Python y PostgreSQL se puso a prueba la arquitectura de búsqueda en paralelo, haciendo los componentes de búsqueda asíncronos para poder ejecutarlos a la vez y, posteriormente, fusionar sus resultados.

3.6.1. Implementación de RRF

Se implementó el algoritmo RRF para la fusión de resultados. No obstante, recordando su funcionamiento, este algoritmo trata por igual a los documentos recuperados por ambos buscadores, recompensando aquellos que aparecen en ambas listas y asignando menor puntuación a medida que descienden en el ranking. Esto plantea una cuestión importante: ¿deberían tener siempre el mismo peso los resultados de ambos buscadores? En muchos casos, la respuesta es no.

RRF ponderado

Para hacer frente a este problema aparece la suma ponderada. Sin embargo, no se puede aplicar a nuestras búsquedas, ya que las puntuaciones de ranking de los componentes de búsqueda (ts_rank y la distancia coseno) no son directamente comparables.

Un concepto que se ha explorado es el de RRF ponderado, que pretende simular la suma ponderada. Se puede ponderar la contribución de cada buscador siguiendo una determinada estrategia. Una opción sencilla sería definir pesos fijos, por ejemplo, 70 % de importancia para la búsqueda semántica y 30 % para la búsqueda por palabras clave. Sin embargo, esto limita la capacidad de adaptarse a diferentes tipos de búsqueda.

Se puede implementar una estrategia de ponderación dinámica, que ajuste los pesos en función de la consulta que introduce el usuario o de la propia puntuación que tienen los resultados en cada buscador. No obstante, en los experimentos realizados no se logró al obtener ponderaciones óptimas de esas características.

Hay diferentes enfoques más avanzados que se pueden seguir para lograr esto, como entrenar un modelo, que dado una consulta estime el valor numérico de ponderación para un tipo de búsqueda. Sin embargo, por limitaciones de tiempo, se optó por una solución más sencilla pero funcional: utilizar un modelo generativo preentrenado para inferir de forma heurística la ponderación. Esto operaba de la siguiente manera[32]:

Consulta: *"The Shinning"*

$$PonderacionClave = Prediccion(Consulta) \rightarrow 0$$

$$PonderacionSemantica = 1 - PonderacionClave \rightarrow 1 - 0 = 1$$

3.6.2. Implementación de Crossencoder

Para la implementación de modelos *crossencoder* como técnica de fusión, primero se llevó a cabo el enfoque simple en el que se creó un conjunto de documentos únicos a partir de los recuperados, sin realizar una fusión propiamente dicha, y se reordenan usando un crossencoder.

Primero se usó *ms-marco-MiniLM-L-6-V2*[20], pero, dado que es un modelo ligero en comparación con los que hay disponibles hoy en día, y que el equipo con el que se trabajaba tardaba en hacer las inferencias con este modelo, se tuvo que recurrir a otro enfoque de *crossencoder* para poder alcanzar más precisión y a la vez que la velocidad de inferencia sea superior.

Cohere rerank

Cohere, empresa ampliamente reconocida en el sector de la inteligencia artificial, pone a disposición de los usuarios el uso de sus LLM vía API. Uno de los modelos que ofrece es Cohere rerank 3.5, un modelo basado en *crossencoders* que se adapta a diferentes tipos de datos y ofrece una gran precisión y velocidad de inferencia.

Para poder usar esta API simplemente es necesario conseguir una API Key y luego hacer una llamada al endpoint POST *https://api.cohere.com/v2/rerank*, al que se debía adjuntar la consulta del usuario y los documentos recuperados. Tras esto la API responde por el campo *results* con una lista de los ids de documentos ordenados y sus relevancias calculadas.

Esta API limita a 4096 tokens por documento, por lo que no tendremos problema, y además, dado que los documentos recuperados son datos estructurados (tal y como se definieron en el formato de salida), se deben pasar en formato YAML por recomendación propia de la API[15].

3.6.3. Combinación de fusiones

Una vez exploradas esas técnicas concluimos haciendo una combinación de estas. Anteriormente al usar los *crossencoder* se realizaba un reordenamiento de todos los documentos, lo cual supone un coste computacional elevado ya que, por lo general, le llegaban unos 500 documentos. Para mitigar esto se tomó la siguiente decisión de diseño: Llevar a cabo RRF sobre los documentos recuperados, ya que es una técnica eficiente y da lugar a una fusión robusta y, posteriormente, hacer reranking con cohere pero solamente sobre los 40 primeros documentos para aumentar la precisión aún más sobre ese subconjunto. Usando esta estrategia se conseguía lo siguiente:

1. Centrar el coste computacional donde el usuario centra su atención, es decir, el usuario promedio que emplea una aplicación de búsqueda de películas no mira resultados más allá de los primeros 10-40 resultados. En caso de que un usuario desee mirar más allá de estos resultados, tendrá una fusión RRF que es menos precisa pero sigue siendo una solución robusta.

2. Dar una fusión más robusta y ordenada al reranking puede lograr que ofrezca un mejor desempeño que pasarle un conjunto desordenado de películas.

3. Reducción considerable del tiempo de búsqueda, mejorando la experiencia del usuario.

3.6.4. Detalles de la implementación

La evaluación de las búsquedas híbridas se realizaron sobre el desempeño medio en todas las categorías de búsqueda. Primero se puso a prueba el RRF con el RRF ponderado:

Modelo	**MRR@10**	**MAP@10**	**NDCG@10**	**TRM**	**PNR**
Híbrido RRF	$0{,}7804 \pm 0{,}012$	$0{,}7454 \pm 0{,}01$	$0{,}9086 \pm 0{,}006$	0.6331	621.8
Híbrido RRF Ponderado	$0{,}7839 \pm 0{,}013$	$0{,}7699 \pm 0{,}017$	$0{,}9134 \pm 0{,}005$	1.9063	515.5

Cuadro 15: Resultados del buscador híbrido con RRF y RRF ponderado

En la tabla 15, se puede observar cómo el calculo de la ponderación claramente ha aumentado el tiempo de cómputo y, sin embargo, apenas se ha visto beneficiado en cuanto a métricas de desempeño.

Posteriormente, se aplicó crossencoders frente a la API de Cohere (tabla 16).

Modelo	MRR@10	MAP@10	NDCG@10	TRM	PNR
Híbrido Cohere	$0{,}9524 \pm 0{,}0$	$0{,}9044 \pm 0{,}006$	$0{,}9505 \pm 0{,}009$	4.6371	516.6
Híbrido Crossencoder	$0{,}7010 \pm 0{,}036$	$0{,}6910 \pm 0{,}022$	$0{,}9047 \pm 0{,}006$	2.4973	460.5

Cuadro 16: Resultados del buscador híbrido usando Cohere y Crossencoder

En los resultados obtenidos en la tabla 16 se puede apreciar que el modelo simple de *crossencoder* no ha conseguido superar al RRF, pero el modelo de Cohere ha conseguido superar a todos los modelos anteriores sustancialmente, a costa de un peor tiempo de respuesta medio.

Hasta este momento se han realizado todas las pruebas de los componentes de búsqueda poniendo como límite de cantidad de resultados por componente de búsqueda a 500.Recordando la idea de que los usuarios solo suelen mirar los primeros resultados de búsqueda, se redujo el límite a 100 resultados por componente de búsqueda, y también, con objeto de evitar que Cohere haga un reranking de todos los documentos, se probó a hacer evaluaciones de la búsqueda híbrida en la que se aplica RRF y luego Cohere solo a los 40 primeros documentos obteniendo los resultados que aparecen en la tabla 17.

Modelo	MRR@10	MAP@10	NDCG@10	TRM	PNR
Híbrido Cohere	$0{,}9524 \pm 0{,}00$	$0{,}9044 \pm 0{,}006$	$0{,}9505 \pm 0{,}009$	4.6371	516.6
Híbrido RRF + Cohere	$0{,}9191 \pm 0{,}027$	$0{,}9078 \pm 0{,}015$	$0{,}9470 \pm 0{,}004$	3.1482	192.5

Cuadro 17: Resultados del buscador híbrido Cohere y su versión con RRF

Gracias a esto se consiguió reducir el tiempo de respuesta en un segundo y medio sin afectar demasiado a las métricas de desempeño. Una vez definido el diseño del componente de búsqueda híbrida, el componente quedó completo y funcional (figura 25), devolviendo siempre el conjunto de resultados ordenados en formato JSON.

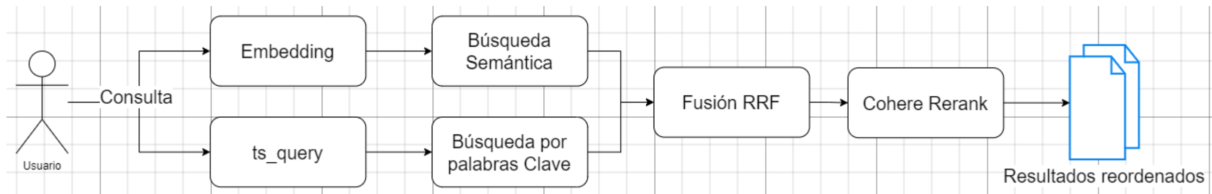

Figura 25: Diagrama de flujo del componente de búsqueda híbrida.

Tras finalizar la implementación del componente de búsqueda híbrida, se puede observar una comparación entre este enfoque y las búsquedas que lo componen en la figura 26. En ella se muestran tres gráficas análogas al de la figura 24 que permiten comparar el rendimiento del buscador por palabras híbrido (en azul) frente al buscador por palabras clave (en rojo) y el buscador semántico (verde).

Se puede observar como la búsqueda híbrida ha conseguido aprovechar los puntos fuertes de ambos buscadores, ofreciendo un gran rendimiento en todos los tipos de búsqueda.

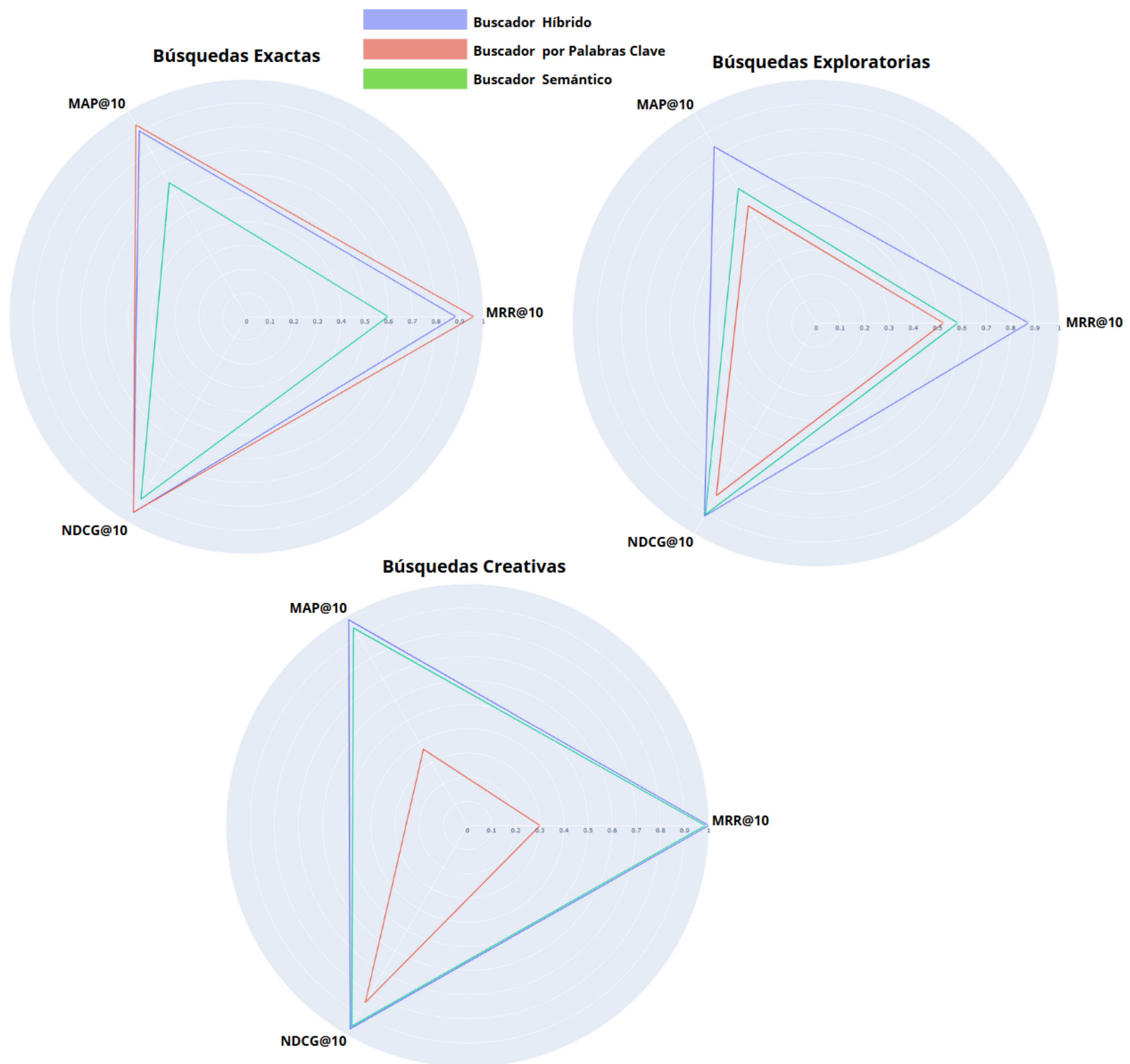

Figura 26: Desempeño de las búsquedas por categoría de búsqueda.

3.6.5. Evaluación final con usuarios reales

Retomando el formulario diseñado previamente, se llevó a cabo una evaluación del componente final de búsqueda híbrida con usuarios reales. Para ello, se implementó una lógica de ramificación en el formulario basada en las preferencias del usuario respecto a la presencia de valoraciones en las películas.

Concretamente, si el usuario indicaba que consideraba esencial que las películas estuvieran evaluadas, se le dirigía a una rama del formulario en la que los resultados de búsqueda

mostrados correspondían únicamente a películas con un filtro de que tengan evaluaciones. En cambio, si esta condición no era importante para el usuario, se le mostraban los resultados completos sin aplicar dicho filtro.

Para hacer menos ardua la evaluación de los usuarios se redujo el número de resultados a evaluar por categoría a 2 y el top_k a 5. Los resultados obtenidos fueron los siguientes: Con 12 usuarios que consideraban esencial que las películas estuvieran evaluadas, las evaluaciones obtenidas fueron muy buenas (tabla 18).

Categoría de búsqueda	**MRR@5**	**MAP@5**	**NDCG@5**
Exactas	1.0000	0.9555	0.9754
Exploratorias	1.0000	0.9346	0.9669
Creativas	0.9792	0.9422	0.9701

Cuadro 18: Resultados de evaluaciones humanas por categoría con búsquedas con el filtro de películas evaluadas

Con 15 usuarios que estaban abiertos a ver películas sin evaluar las evaluaciones obtenidas fueron también muy buenas (tabla 19).

Categoría de búsqueda	**MRR@5**	**MAP@5**	**NDCG@5**
Exactas	1.0000	0.9805	0.9849
Exploratorias	0.8558	0.8541	0.9349
Creativas	0.9423	0.9121	0.9806

Cuadro 19: Resultados de evaluaciones humanas por categoría con búsquedas sin el filtro de películas evaluadas

A la vista de los resultados podemos concluir una satisfacción general de los usuarios con los resultados obtenidos.

4

Desarrollo de la Aplicación Web

4.1. Desarrollo del Frontend

Una vez desarrollada la búsqueda híbrida, ya se pudo empezar a preparar la interfaz de la aplicación web, como una aplicación de una sola página (*Single Page Aplication*, SPA) compuesta por un buscador que se conecta por API al buscador híbrido.

Se ha utilizado ReactJS como framework, estructurando el frontend en distintos componentes de React —uno para la barra de búsqueda, otro para la lista de resultados, entre otros—. La lógica de interacción entre los componentes se gestionó mediante hooks como useState, permitiendo actualizar dinámicamente la interfaz en función del estado de la aplicación. El diseño base del buscador se muestra en la figura 27.

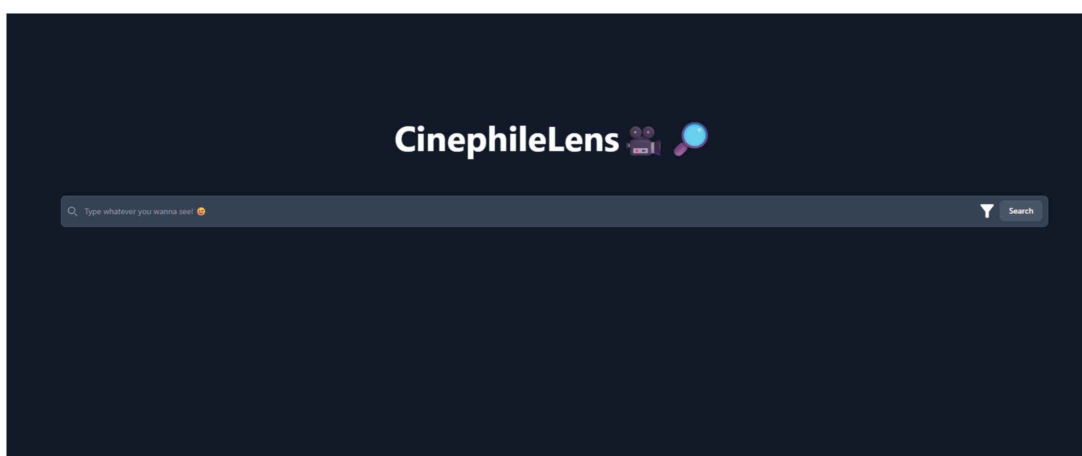

Figura 27: Diseño buscador por defecto.

Cuando se realiza una búsqueda, aparecerán las películas en diseños de tarjetas, cada una dando una vista previa de la película, indicando su título, año de lanzamiento y la imagen del póster (figura 28).

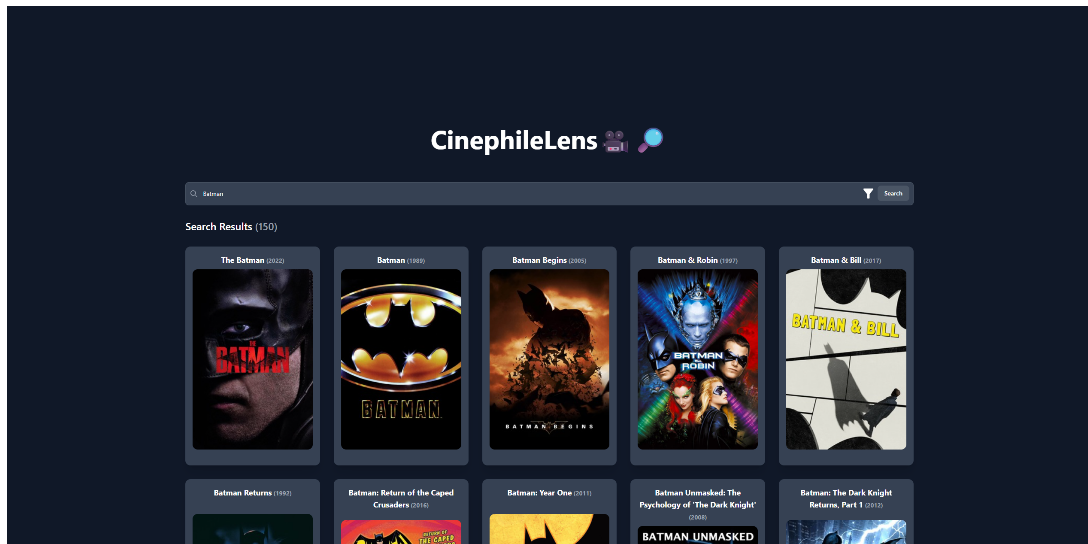

Figura 28: Diseño de la página tras buscar.

Al pulsar una tarjeta lleva a una vista detallada donde aparecen todos los metadatos relativos a esa película (figura 29).

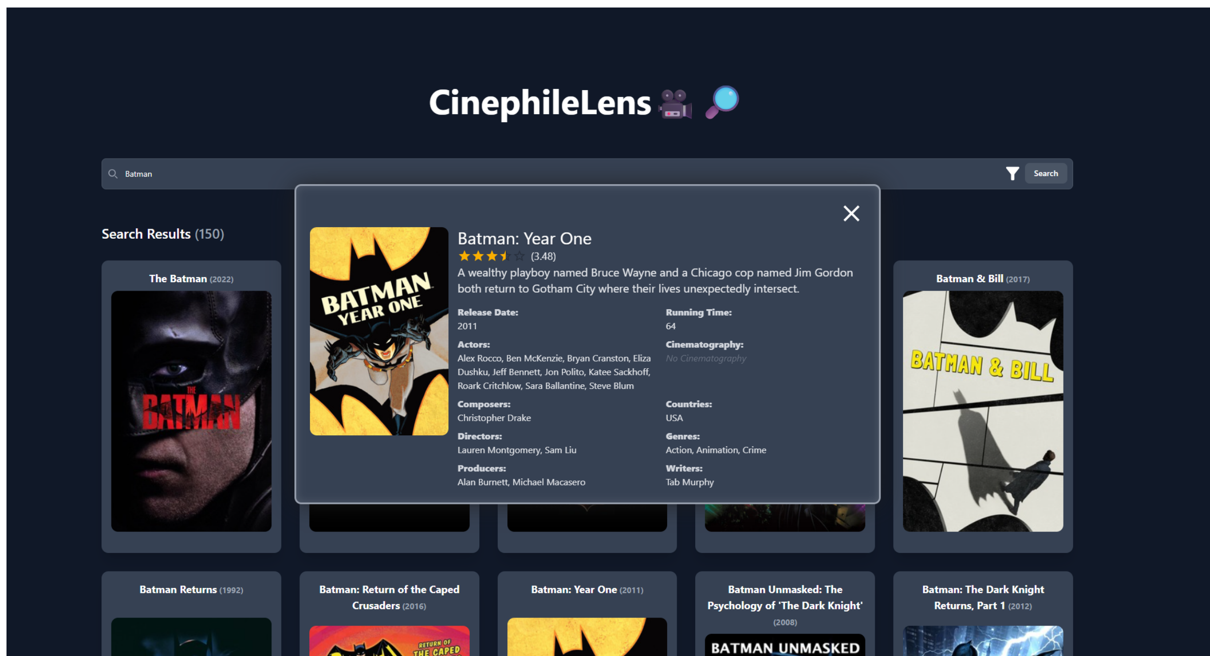

Figura 29: Diseño de la vista detallada de las películas.

4.1.1. Microservicios

Se implementó la lógica del buscador híbrido como un backend con FastAPI, permitiendo conectar las consultas del usuario del frontend con el buscador híbrido y su flujo completo de procesamiento.

Filtros

Los metadatos que no se contemplaron para las búsquedas, se han implementado como filtros en el backend de forma que el usuario tenga aún más libertad a la hora de elegir que quiere ver (figura 30).

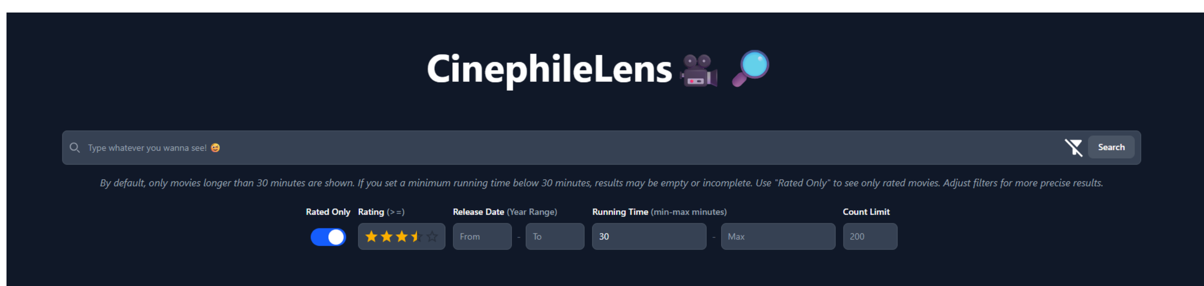

Figura 30: Diseño del buscador con filtros..

- **Rating:** Permite al usuario filtrar por puntuaciones entre 1 y 5, en intervalos de 0.5. Por ejemplo, si filtra por puntuación 3, solo le saldrán películas que tengan una puntuación mayor o igual a 3.
- **Rated only:** Checkbox usado para determinar si los resultados devueltos tienen una evaluación o no. Por defecto está activado ya que la mayoría de usuarios prefieren que estén evaluadas, pero el usuario tiene libertad de cambiarlo y explorar.
- **Release date:** Permite especificar el rango de años en el que salió la película.
- **Running time:** Permite especificar el rango de duración en minutos de la película. Por defecto tiene un rango inferior de 30 para que solo salgan películas o largometrajes (al durar más de 30 minutos), pero el usuario tiene libertad de cambiarlo y explorar.

- **Count limit:** Permite especificar el límite máximo de resultados devueltos por el buscador. Por defecto está a 200 para cubrir la necesidad general de los usuarios, basándonos en la idea de que los usuarios rara vez ven más de 100 resultados.

Se llevó a cabo una medición sobre cómo afectaba el filtro por defecto de rated only al buscador (tabla 20).

Modelo	MRR@10	MAP@10	NDCG@10	TRM	PNR
Sin Filtro	$0{,}9191 \pm 0{,}027$	$0{,}9078 \pm 0{,}015$	$0{,}9470 \pm 0{,}004$	3.1482	192.5
Con Filtro de Evaluados	$0{,}9341 \pm 0{,}003$	$0{,}9194 \pm 0{,}015$	$0{,}9511 \pm 0{,}003$	1.2328	101.3

Cuadro 20: Comparación de desempeño general del buscador híbrido RRF con y sin filtro de películas evaluadas

La aplicación del filtro ha mejorado levemente las métricas y reducido en dos segundos el tiempo de búsqueda y, por lo general, sigue recuperando cerca de 100 documentos. Esto permite concluir que la aplicación del filtro es beneficiosa.

4.2. Funcionalidades adicionales de la aplicación

Tras el desarrollo de la aplicación se decidió llevar a cabo nuevas funcionalidades para mejorar aún más la herramienta.

4.2.1. Corrector

Hasta ahora todas las evaluaciones se realizaron con consultas escritas sin fallos ortográficos, pero, ¿cómo podemos mitigar los malos resultados que puedan salir por una consulta mal escrita? La solución implementada fue la creación de un corrector personalizado, que funciona a partir de un diccionario de frecuencias de palabras. Su funcionamiento es el siguiente [14]:

1. Se selecciona una palabra que no sea una *stop word*, para comprobar si se le puede asignar una corrección.

2. Se define un umbral de frecuencia, en nuestro caso 60, de forma que si la palabra tiene más de esa frecuencia en el diccionario se considera que es una palabra normal del dominio y no se sugiere correcciones.

3. Si se va a dar sugerencias de corrección a una palabra, se genera el conjunto de palabras resultante de editar un número determinado de letras en esa palabra. Editar implica eliminar, añadir y editar letras, en mi caso elegí la edición de hasta dos letras. Por ejemplo:

   Entrada: *"Mad"* ⇒ **Salida:** *{Ma, M, Mu, Mada, Mad, ...}*

4. A partir del diccionario de frecuencias, se calcula, para cada palabra, la probabilidad de que sea la seleccionada para ser una sugerencia de corrección a partir de la siguiente fórmula:

   $$P(w) = \frac{\text{count}(w)}{\sum_{w' \in V} \text{count}(w')}$$

   donde:

   - $P(w)$ es la probabilidad de la palabra w,
   - count(w) es el número de apariciones de la palabra w en el diccionario
   - V es el vocabulario total (todas las palabras en el diccionario),

5. La palabra con mayor probabilidad es presentada al usuario y ya el debe decidir si aceptar o no la sugerencia.

Este diccionario de palabras y frecuencias específico de nuestro dominio lo obtenemos de nuestro propio dataset. Para ello FTS permite crear una tabla de palabras y sus frecuencias en los documentos usando la operación *ts_stat* con los lexemas obtenidos a partir de crear *ts_vectors* con la configuración 'simple' sobre las palabras clave de nuestro dataset (título, géneros, descripción, etc.).

Luego solo tenemos que hacer una llamada a la tabla para crear el diccionario de palabras en el dataset y sus frecuencias. La configuración simple, a diferencia de la configuración en

inglés no aplica stemming ni elimina stop words, de forma que deja las palabras tal y como aparecen, lo que nos conviene para estas correcciones [35].

Esta corrección de consultas se añade como endpoint a la API del backend desarrollado, de forma que cuando un usuario deja de escribir, manda una petición POST a la API y si hay sugerencias de corrección aparecerán en la página sobre el buscador (figura 31).

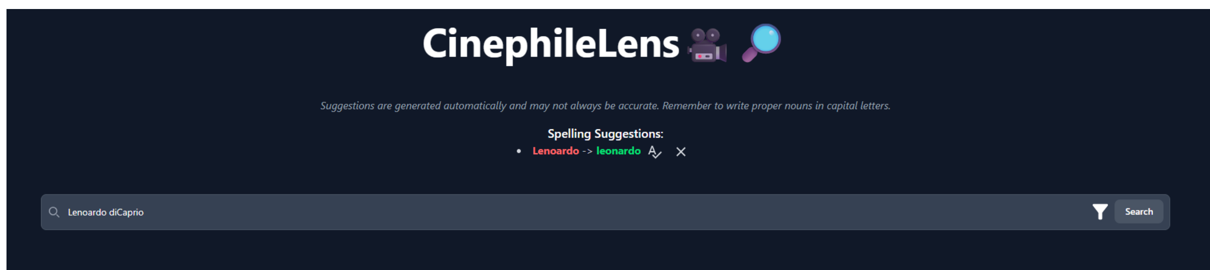

Figura 31: Vista del corrector en la aplicación

4.2.2. RAG

Una de las grandes tendencias recientes es la creación de sistemas de recuperación aumentada por generación (RAG). Estos sistemas surgen como una evolución de los chatbots, pasando de permitir que la IA responda directamente a partir de los datos con los que fue entrenada, a un enfoque en el que primero se realiza una recuperación de información, y luego la IA responde basándose en dicha información recuperada.

Con esto se consigue reducir las alucinaciones de la IA generativa, aumentar la precisión de las respuestas y añadir más garantía a la fiabilidad de los datos y contexto a sus respuestas. Estos sistemas, entre otras cosas, permiten interactuar con los datos obtenidos al hacer una búsqueda. La IA generativa puede ayudar a justificar las respuestas, a explicarlas, etc.

Este concepto se ha llevado a nuestra aplicación, de forma que, aplicando las buenas prácticas de prompting que fueron usadas para las evaluaciones. Se han conectado las búsquedas a llamadas a Gemini Flash 2.0 para ofrecer una mejor experiencia al usuario y constituyendo un sistema RAG. Concretamente se ha usado la IA generativa para lo siguiente:

1. Se envía a la IA generativa la consulta y los diez primeros resultados, para que haga

una reseña general de esos primeros resultados obtenidos usando un lenguaje natural y amigable. La idea es que el usuario pueda entender el contexto de los datos y qué resultados pueden ser de su interés (figura 32).

2. Se manda a la IA generativa la consulta, la película a la que el usuario haya solicitado reseña tras pulsar un botón de generar reseña y la posición en el ranking en el que está esa película. A partir de esto la IA ofrece una reseña de la película usando un lenguaje natural y amigable, para ayudar al usuario a entender el contexto de esa película y a decidir si le interesa o no el resultado.

   La posición de la película en el ranking se adjunta para que, en caso de que el usuario pida la reseña de un mal resultado, pero esté en mala posición en el ranking, pueda justificarle, de forma amable, que es normal que ese resultado sea malo y aún así explicárselo por si le llegara a interesar (figura 33).

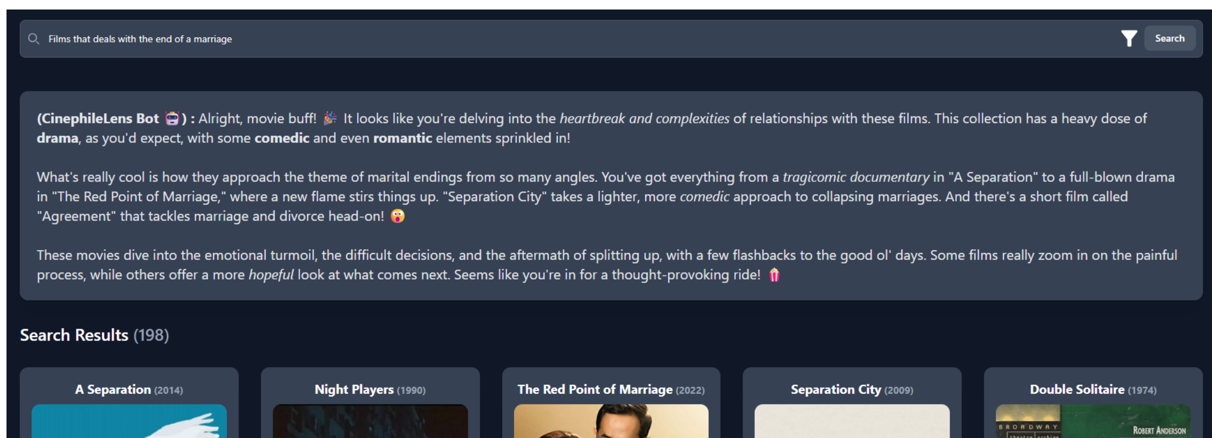

Figura 32: Vista de la reseña general del RAG

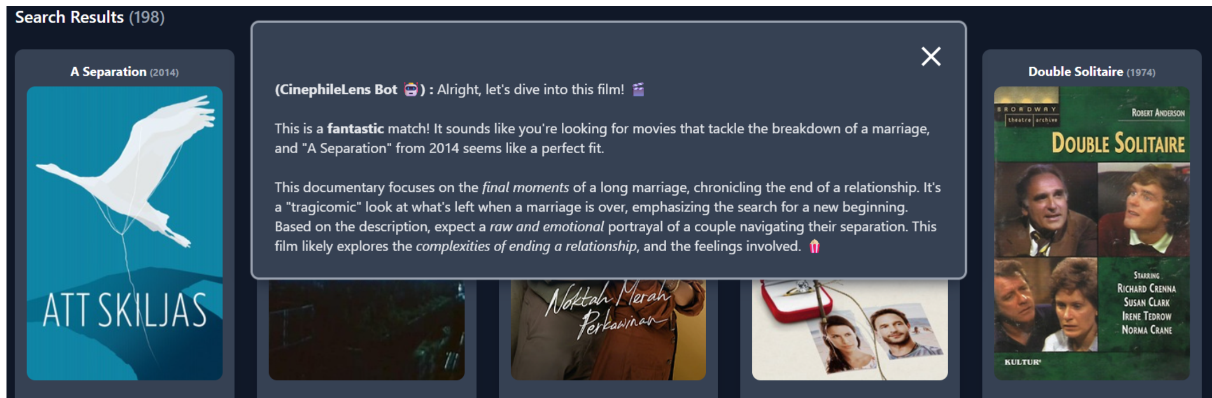

Figura 33: Vista de la reseña del RAG de una película específicada

La lógica de ambas reseñas se han añadido como endpoint a la API. Al endpoint de la reseña general se le llama cuando se obtienen los resultados de búsqueda y al de la reseña específica cuando el usuario pulsa el botón de generar reseña sobre una película (figura 34).

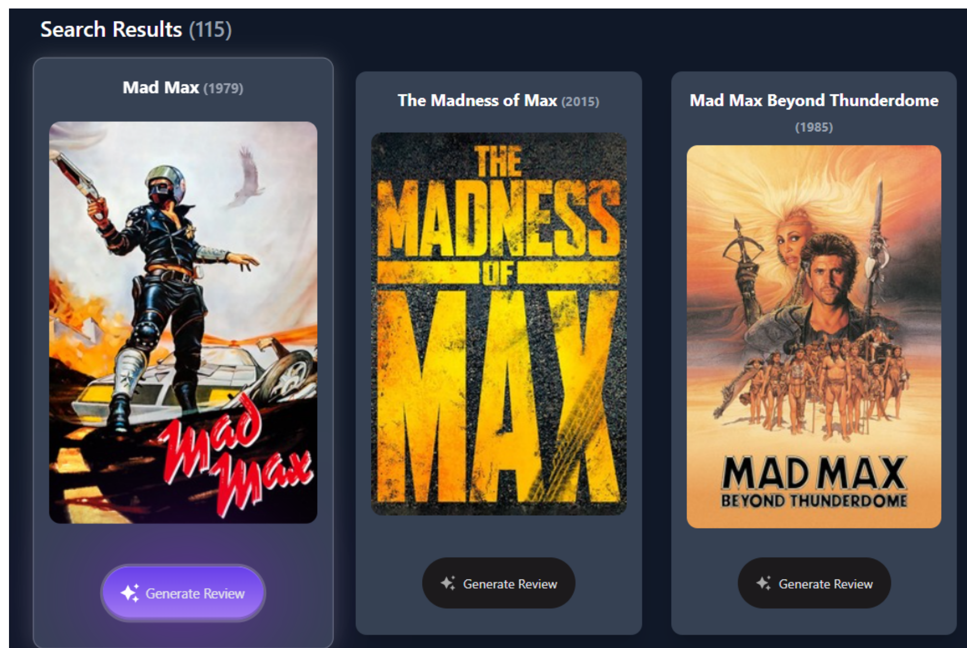

Figura 34: Vista del botón de generar reseña personalizada

5

Conclusiones y Líneas Futuras

5.1. Conclusiones

Este proyecto ha abarcado desde el estudio de la base de los motores de búsqueda y las técnicas de NLP tradicionales, hasta las capacidades vanguardistas de los motores de búsqueda semánticos y las técnicas de reordenación impulsadas por los LLM basados en transformadores.

Este conocimiento ha permitido desarrollar enfoques de buscadores híbridos que combinan lo mejor de los métodos tradicionales y modernos. Al integrar estos enfoques con mecanismos de evaluación inteligentes y una comprensión clara de las necesidades del usuario —obtenidas mediante la recopilación de datos y evaluaciones con usuarios reales— se ha logrado construir un buscador de películas que otorga una notable autonomía a quienes lo utilizan.

Se ha podido integrar esta poderosa herramienta en un frontend conectado a un backend a través de una API desarrollados con framework modernos y una base de datos PostgreSQL, garantizando la escalabilidad y preparación para despliegue de la aplicación. Incluso se han integrado en la aplicación web diversas funcionalidades de mejoras como la integración de un corrector de consultas personalizado y un RAG que permite a los usuarios tener reseñas de los resultados que obtienen al buscar.

5.2. Líneas Futuras

Durante el desarrollo de este trabajo se ha puesto de manifiesto la gran variedad de configuraciones que pueden explorarse para refinar los motores de búsqueda, así como la relevancia que tienen los datos en el rendimiento de estos sistemas. Además los nuevos avances de la IA están permitiendo crear sistemas de IR cada vez más complejos y eficaces. Por esta razón, aún habiendo conseguido buenos resultados, hay diferentes líneas futuras que se podrían considerar para este proyecto:

- **Contemplar más categorías de búsqueda:** A pesar de habernos centrado en tres categorías estrechamente relacionadas con los diferentes componentes de búsqueda, se podría experimentar con nuevas categorías, como búsquedas que relacionen metadatos entre sí (lo cual se vería altamente beneficiado por una base de datos de grafos), o búsquedas que requieran la incorporación de nuevos datos como personajes, elementos del guion, entre otros.
- **Realizar ajuste fino:** A pesar de que tanto el reranker como el modelo de embeddings nos ha dado buenos resultados en este caso de uso, hacerle ajuste fino a ambos modelos con consultas y documentos del dominio puede llevar a una mejora considerable de la calidad del buscador.
- **Mejorar el RAG:** Aunque se ha desarrollado un RAG básico a partir del buscador, se podría profundizar en su desarrollo convirtiendo la aplicación en un agente de IA, aplicando técnicas avanzadas de RAG para generar respuestas más precisas y personalizadas, o incorporando mecanismos específicos para evaluar la calidad de las respuestas generadas.
- **Mejorar los componentes de búsqueda:** Se podría trabajar en mejoras de estos, como por ejemplo, incorporar un modelo NER más preciso para el reconocimiento de entidades en consultas por palabras clave, utilizar modelos de embeddings más avanzados, explorar nuevas técnicas de fusión de resultados, arquitecturas híbridas o integrar otros modelos de reordenación que mejoren la calidad del ranking final.

Referencias

[1] The limitations of keyword search, 2017. Copyright Clearance Center.

[2] Faiss hnsw index explained, n.d. Pinecone.

[3] Letterboxd movies dataset, n.d. Kaggle, publicado por gsimonx37.

[4] Openai tokenizer tool, n.d. OpenAI, herramienta oficial para visualizar el tokenizado.

[5] pgvector: Open-source vector similarity search for postgres, n.d. GitHub - pgvector.

[6] Understanding tf-idf for machine learning, n.d. Capital One.

[7] Confident AI. Llm-as-a-judge simply explained: A complete guide to run llm evals at scale. https://www.confident-ai.com/blog/why-llm-as-a-judge-is-the-best-llm-evaluation-method, 2025.

[8] Evidently AI. A complete guide to ranking and recommendations metrics. Evidently AI guide, 2025.

[9] Evidently AI. Guía de ingeniería de prompt. Evidently AI guide, n.d.

[10] Jay Alammar and Maarten Grootendorst. *Hands-On Large Language Models: Build and deploy LLM-powered applications.* O'Reilly Media, Sebastopol, CA, 2024.

[11] R. Asale and Rae. Semántico, semántica, n.d. Diccionario de la lengua española - RAE.

[12] Joel Barnard. What are word embeddings?, 2024. IBM Think.

[13] Shauntee Burns. What is boolean search?, 2011. The New York Public Library blog.

[14] Ruchita Chimu. Autocorrect using nlp, 2020. Medium.

[15] Cohere. Cohere api reference — rerank (v2), 2024. Documentación oficial de la API de Rerank.

[16] Elastic. Elasticsearch hybrid search, 2023. Elastic Labs.

[17] Elastic. Learning to rank (ltr), 2025. Elastic.

[18] Elastic. ¿qué es la recuperación de información?, n.d. Elastic.

[19] W. M. Experts et al. How search engines work, 2024. SEO.com.

[20] Hugging Face. cross-encoder/ms-marco-minilm-l6-v2, n.d. Modelo de Hugging Face basado en Cross-Encoder para tareas de ranking.

[21] GeeksforGeeks contributors. Nlp | how tokenizing text, sentence, words works, 2025. GeeksforGeeks learning portal.

[22] PostgreSQL Global Development Group. Gist and gin index types. En PostgreSQL 9.1 Documentation, 2011.

[23] PostgreSQL Global Development Group. 12.3 controlling text search. En PostgreSQL 17 Documentation, 2025. Full Text Search Controlling Official Documentation.

[24] PostgreSQL Global Development Group. 64.4 gin indexes. En PostgreSQL 17 Documentation, 2025.

[25] IBM. ¿qué es el pln (procesamiento del lenguaje natural)?, 2025. IBM.

[26] IBM Think contributors. What is named entity recognition?, 2023. IBM Think.

[27] et al. Kenneth Enevoldsen. Mmteb: Massive multilingual text embedding benchmark. *arXiv preprint arXiv:2502.13595*, 2025.

[28] O. Kopp. The most important ranking methods for modern search engines, 2025. Kopp Consulting.

[29] Regina O. Obe and Leo S. Hsu. *PostgreSQL: Up and Running*. O'Reilly Media, 3 edition, 2017.

[30] Pinecone. Hybrid search, n.d. PineCone.

[31] Nils Reimers and Iryna Gurevych. Semantic search with sentence transformers, 2023. Sentence-Transformers Documentation.

[32] Shubham Sarkar. Hybrid search in rag: Concept of weighted reciprocal rank fusion (rrf) — part 1, 2024. Medium.

[33] Deval Shah. Mathematical intuition behind reciprocal rank fusion (rrf) — explained in 4 mins, 2024. Medium.

[34] Team Timescale and Hervé Ishimwe. Combining semantic search and full-text search in postgresql (with cohere, pgvector, and pgai), 2024. Timescale Blog, publicado el 15 de noviembre de 2024.

[35] The PostgreSQL Global Development Group. Postgresql: Text search functions and operators — statistics, 2024. Documentación oficial de PostgreSQL sobre estadísticas en búsqueda textual.

[36] Anthony Técher. Keyword extraction: Understanding search algorithms, 2020. SEO-Quantum.

[37] Wikipedia contributors. Stemming, 2021. Wikipedia, La Enciclopedia Libre.

[38] Wikipedia contributors. Lemmatization, 2024. Wikipedia.

[39] Wikipedia contributors. Palabra vacía, 2024. Wikipedia, La Enciclopedia Libre.

[40] Sudhir Yelikar. Understanding similarity or semantic search and vector databases, 2023. Medium.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga