Sistema de Recuperación de Información sobre Wikipedia

Práctica 4 - Recuperación de Información

Jesús J. Cantero López

Curso 2024/25

Índice

1. Introducción

En esta práctica se desarrolla un **sistema completo de Recuperación de Información (RI)** que permite buscar en una colección de más de 10 GB de artículos de Wikipedia. El sistema está compuesto por:

- **Backend**: API REST desarrollada con FastAPI que implementa el pipeline completo de RI
- **Frontend**: Interfaz web desarrollada con React, TypeScript y TailwindCSS
- **Corpus**: Artículos de Wikipedia en castellano, catalán y portugués (+10 GB)

El modelo de recuperación utilizado es el **modelo vectorial con TF-IDF** y **similitud del coseno**, implementando un índice invertido persistente que permite búsquedas eficientes sobre millones de documentos.

1.1. Objetivos

1. Implementar un backend con FastAPI que procese documentos, construya el índice e implemente la búsqueda
2. Desarrollar un frontend en React para enviar consultas y visualizar resultados
3. Utilizar una colección documental real de más de 10 GB
4. Documentar el proceso, problemas encontrados y soluciones aplicadas

2. Entorno de trabajo

2.1. Backend (Python)

Se ha utilizado **Python 3.11** con Anaconda para gestionar el entorno virtual.

```
conda create -n P4_Final_RI python=3.11
conda activate P4_Final_RI
pip install -r backend/requirements.txt
```

Las principales librerías utilizadas son:

- `fastapi`: Framework web asíncrono para la API REST
- `uvicorn`: Servidor ASGI para ejecutar FastAPI
- `nltk`: Procesamiento de lenguaje natural (stopwords, stemming)
- `pydantic`: Validación de datos y modelos

2.2. Frontend (JavaScript/TypeScript)

Se utilizó **Node.js 18+** con el siguiente stack:

- `Vite`: Bundler moderno (más rápido que create-react-app)
- `React 18`: Framework de interfaz de usuario
- `TypeScript`: Tipado estático para JavaScript
- `TailwindCSS`: Framework CSS utility-first
- `Lucide React`: Iconos

2.3. Contenerización

El sistema puede ejecutarse con **Docker** y **Docker Compose** para facilitar el despliegue:

```
docker-compose up
```

3. Estructura del proyecto

La estructura de carpetas y archivos del proyecto es la siguiente:

```
P4_RI/
|-- backend/                    (API FastAPI)
|   |-- main.py                 (punto de entrada de la API)
|   |-- config.py               (configuración centralizada)
|   |-- preprocessing.py        (tokenización, stopwords, stemming)
|   |-- indexing.py             (cálculo TF-IDF, índice invertido)
|   |-- build_index.py          (indexación de castellano)
|   |-- build_index_ca.py       (indexación de catalán)
|   |-- build_index_pt.py       (indexación de portugués)
|   |-- wikipedia_loader.py     (carga de artículos JSON)
|   |-- persistent_index.py     (índice persistente en disco)
|   |-- requirements.txt
|-- frontend/                   (Interfaz React)
|   |-- src/
|   |   |-- App.tsx             (componente principal)
|   |   |-- api.ts              (cliente API)
|   |   |-- types.ts            (tipos TypeScript)
|   |-- package.json
|-- datos/                      (corpus de Wikipedia)
|   |-- extracted_es/           (artículos castellano, ~5.8 GB)
|   |-- extracted_ca/           (artículos catalán, ~1.9 GB)
|   |-- extracted_pt/           (artículos portugués, ~2.8 GB)
|   |-- index/                  (índices generados)
|       |-- es/                 (índice castellano)
|       |-- ca/                 (índice catalán)
|       |-- pt/                 (índice portugués)
|-- doc_practica4/              (documentación)
|-- Dockerfile                  (imagen Docker)
|-- docker-compose.yml          (orquestación de servicios)
```

3.1. Descripción de Archivos del Backend

- `main.py`: API FastAPI con todos los endpoints de búsqueda y pipeline
- `preprocessing.py`: Funciones de tokenización, eliminación de stopwords y stemming
- `indexing.py`: Cálculo de TF, IDF, TF-IDF y construcción del índice invertido
- `build_index.py`: Script de indexación batch para castellano
- `build_index_ca.py`: Script de indexación batch para catalán
- `build_index_pt.py`: Script de indexación batch para portugués
- `wikipedia_loader.py`: Iterador sobre archivos JSON de WikiExtractor
- `persistent_index.py`: Clase para guardar/cargar el índice en disco

3.2. Ejecución del Sistema

3.2.1. Opción rápida: Docker

La forma más sencilla de ejecutar el sistema es mediante Docker. Desde el directorio raíz del proyecto:

```
docker compose up
```

Esto levantará automáticamente el backend y el frontend, dejando la plataforma lista para usar.

3.2.2. Opción manual

```
# 1. Instalar dependencias del backend
cd backend
pip install -r requirements.txt

# 2. Construir índices (por idioma, este paso ya está realizado)
python build_index.py       # Castellano (~3 horas)
python build_index_ca.py    # Catalán (~41 min)
python build_index_pt.py    # Portugués (~48 min)

# 3. Ejecutar backend
uvicorn main:app --reload

# 4. Instalar y ejecutar frontend
cd frontend
npm install
npm run dev
```

4. Colección Documental

4.1. Elección de la Fuente

Se eligió **Wikipedia** como fuente de datos por las siguientes razones:

- **Variedad temática**: Artículos de ciencia, historia, cultura, geografía, etc.
- **Experiencia realista**: Simula un buscador real que los usuarios conocen
- **Metadatos ricos**: Cada artículo incluye título, URL y contenido estructurado
- **Licencia libre**: CC BY-SA permite uso académico
- **Descarga sencilla**: Dumps oficiales disponibles en Wikimedia

4.2. Obtención de los Datos

Los datos se obtuvieron de los dumps oficiales de Wikimedia:

- Castellano: `https://dumps.wikimedia.org/eswiki/latest/`
- Catalán: `https://dumps.wikimedia.org/cawiki/latest/`
- Portugués: `https://dumps.wikimedia.org/ptwiki/latest/`

La extracción se realizó con **WikiExtractor**, que convierte el XML de Wikipedia a archivos JSON con una línea por artículo:

```
{"id": "12", "url": "https://es.wikipedia.org/wiki/...",
 "title": "Título", "text": "Contenido del artículo..."}
```

4.3. Estadísticas del Corpus

Idioma	Artículos	Términos únicos	Tiempo
Castellano	1,924,610	2,657,342	188.6 min
Catalán	743,723	1,574,860	41.1 min
Portugués	1,057,354	1,603,916	47.6 min
Total	**3,725,687**	-	**4.6 horas**

Cuadro 1: Estadísticas del corpus indexado.

El tamaño total de los datos extraídos es de **10.5 GB**, superando el requisito mínimo de 10 GB.

5. Pipeline de Preprocesamiento

El preprocesamiento de documentos y consultas sigue el siguiente pipeline:

5.1. Análisis Léxico

Normalización del texto: conversión a minúsculas y limpieza de caracteres especiales.

```
1 def normalize_text(text: str) -> str:
2     text = text.lower()
3     text = re.sub(r'[^\w\s]', ' ', text)
4     return text
```

Listing 1: Normalización de texto

5.2. Tokenización

Extracción de palabras mediante expresiones regulares:

```python
def tokenize_text(text: str) -> list[str]:
    text = normalize_text(text)
    tokens = re.findall(r'\w+', text)
    return tokens
```

Listing 2: Tokenización

5.3. Eliminación de Stopwords

Filtrado de palabras vacías usando NLTK. Se soportan tres idiomas:

```python
def remove_stopwords_tokens(tokens, language="spanish"):
    stop_words = set(stopwords.words(language))
    return [t for t in tokens if t not in stop_words]
```

Listing 3: Eliminación de stopwords

5.4. Stemming

Reducción de palabras a su raíz usando el algoritmo Snowball:

```python
def lemmatize_or_stem_tokens(tokens, language="spanish"):
    stemmer = SnowballStemmer(language)
    return [stemmer.stem(t) for t in tokens]
```

Listing 4: Stemming con Snowball

6. Modelo de Recuperación

6.1. Modelo Vectorial TF-IDF

El sistema utiliza el **modelo vectorial** con ponderación **TF-IDF** (Term Frequency - Inverse Document Frequency). Cada documento y consulta se representa como un vector en un espacio de términos.

6.1.1. Cálculo de TF (Term Frequency)

La frecuencia normalizada del término t en el documento d:

$$\mathrm{TF}(t,d) = \frac{f(t,d)}{|d|} \tag{1}$$

Donde $f(t,d)$ es el número de ocurrencias del término y $|d|$ es el número total de tokens en el documento.

6.1.2. Cálculo de IDF (Inverse Document Frequency)

Mide la importancia del término en la colección. Se usa IDF suavizado:

$$\mathrm{IDF}(t) = \log \frac{N+1}{df_t + 1} + 1 \tag{2}$$

Donde N es el número total de documentos y df_t es el número de documentos que contienen el término.

6.1.3. Peso TF-IDF

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t) \tag{3}$$

```
# Calcular TF
tf = count / n_tokens

# Calcular IDF suavizado
idf[term] = log((doc_count + 1) / (df_t + 1)) + 1.0

# Peso TF-IDF
tfidf = tf * idf[term]
```

Listing 5: Cálculo de TF-IDF en el índice

6.2. Similitud del Coseno

Para calcular la relevancia de un documento respecto a una consulta, se utiliza la **similitud del coseno**:

$$\text{sim}(q, d) = \frac{\vec{q} \cdot \vec{d}}{||\vec{q}|| \times ||\vec{d}||} = \frac{\sum_{i=1}^{n} q_i \times d_i}{\sqrt{\sum_{i=1}^{n} q_i^2} \times \sqrt{\sum_{i=1}^{n} d_i^2}} \tag{4}$$

Las normas de los documentos se precalculan durante la indexación para acelerar las búsquedas.

```
def search(self, query: str, top_k: int = 10):
    # Preprocesar consulta
    query_tokens = preprocess_document(query, self.language)

    # Calcular vector TF-IDF de la consulta
    query_vec = {}
    for term, count in Counter(query_tokens).items():
        if term in self.idf:
            tf = count / len(query_tokens)
            query_vec[term] = tf * self.idf[term]

    # Calcular similitud con cada documento
    scores = defaultdict(float)
    query_norm = sqrt(sum(w*w for w in query_vec.values()))

    for term, weight in query_vec.items():
        if term in self.inverted_index:
            for doc_id, doc_weight in self.inverted_index[term]:
                scores[doc_id] += weight * doc_weight

    # Normalizar por normas de documentos
    for doc_id in scores:
        scores[doc_id] /= (query_norm * self.doc_norms[doc_id])

    # Ordenar por score descendente
    return sorted(scores.items(), key=lambda x: x[1], reverse=True)[:
       top_k]
```

Listing 6: Cálculo de similitud coseno

6.3. Índice Invertido

El índice invertido es la estructura de datos principal que permite búsquedas eficientes. Para cada término, almacena una lista de documentos que lo contienen junto con su peso TF-IDF:

```
{
  "madrid": [("doc_123", 0.045), ("doc_456", 0.032), ...],
  "españa": [("doc_789", 0.067), ("doc_123", 0.041), ...],
  ...
}
```

Los postings están ordenados por peso descendente para optimizar las búsquedas.

7. API REST (Backend)

7.1. Endpoints Implementados

La API FastAPI expone los siguientes endpoints:

Método	Endpoint	Descripción
GET	`/`	Información de la API
GET	`/stats`	Estadísticas del índice cargado
GET	`/search?q={query}`	Búsqueda de documentos
GET	`/document/{id}`	Obtener documento por ID
POST	`/lexical_analysis`	Normalización de texto
POST	`/tokenize`	Tokenización
POST	`/remove_stopwords`	Eliminación de stopwords
POST	`/lemmatize`	Stemming/Lematización
POST	`/analyze_query`	Pipeline completo de análisis

Cuadro 2: Endpoints de la API REST.

7.2. Ejemplo de Respuesta de Búsqueda

```
GET /search?q=historia+de+españa&lang=es&top_k=5

{
  "query": "historia de españa",
  "language": "es",
  "results": [
    {
      "id": "12345",
      "title": "Historia de España",
      "url": "https://es.wikipedia.org/wiki/Historia_de_España",
      "score": 0.847,
      "snippet": "La historia de España abarca el período..."
    },
    ...
  ],
  "total_results": 5,
  "search_time_ms": 234
}
```

8. Interfaz de Usuario (Frontend)

El frontend está desarrollado con **React** y **TypeScript**, utilizando **TailwindCSS** para los estilos.

8.1. Características

- **Selector de idioma**: Permite elegir entre castellano, catalán y portugués
- **Barra de búsqueda**: Campo de texto con botón de búsqueda
- **Indicador de carga**: Spinner mientras se realiza la búsqueda
- **Lista de resultados**: Muestra los 20 primeros resultados relevantes con título, score, snippet y enlace a Wikipedia
- **Estadísticas dinámicas**: Los indicadores de número de documentos, vocabulario y estado del índice se actualizan automáticamente después de cada consulta en cada idioma
- **Gestión de errores**: Mensajes cuando el backend no está disponible

8.2. Capturas de Pantalla

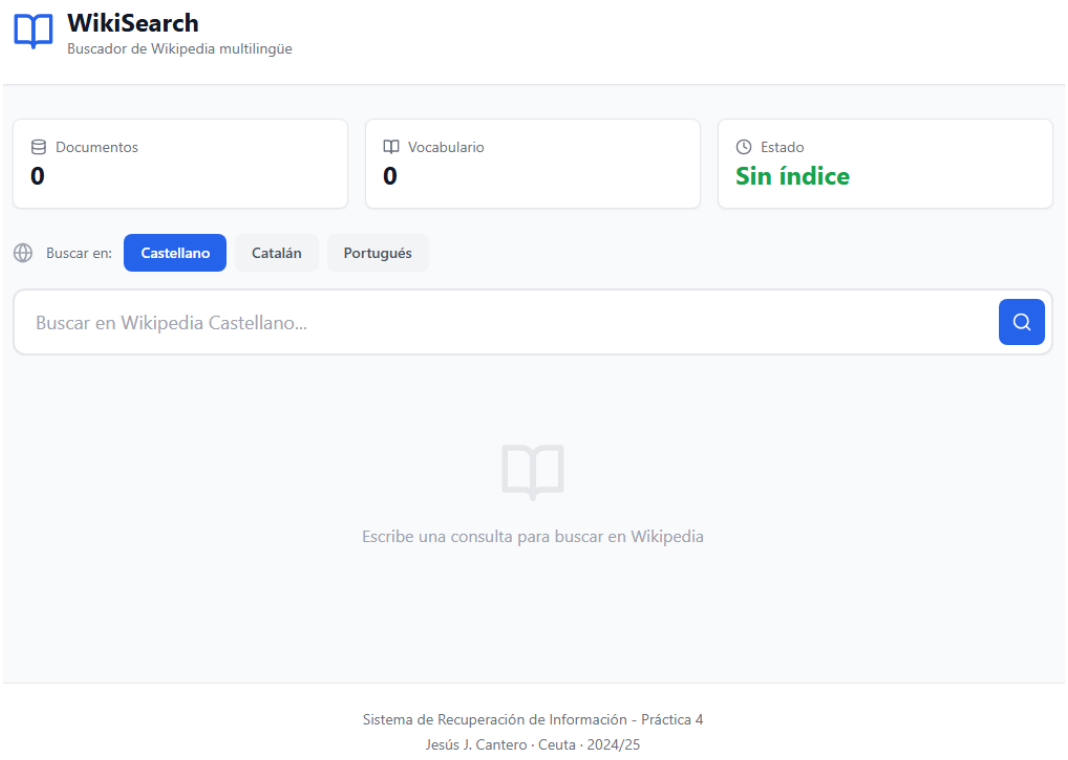

Figura 1: Interfaz principal del buscador WikiSearch.

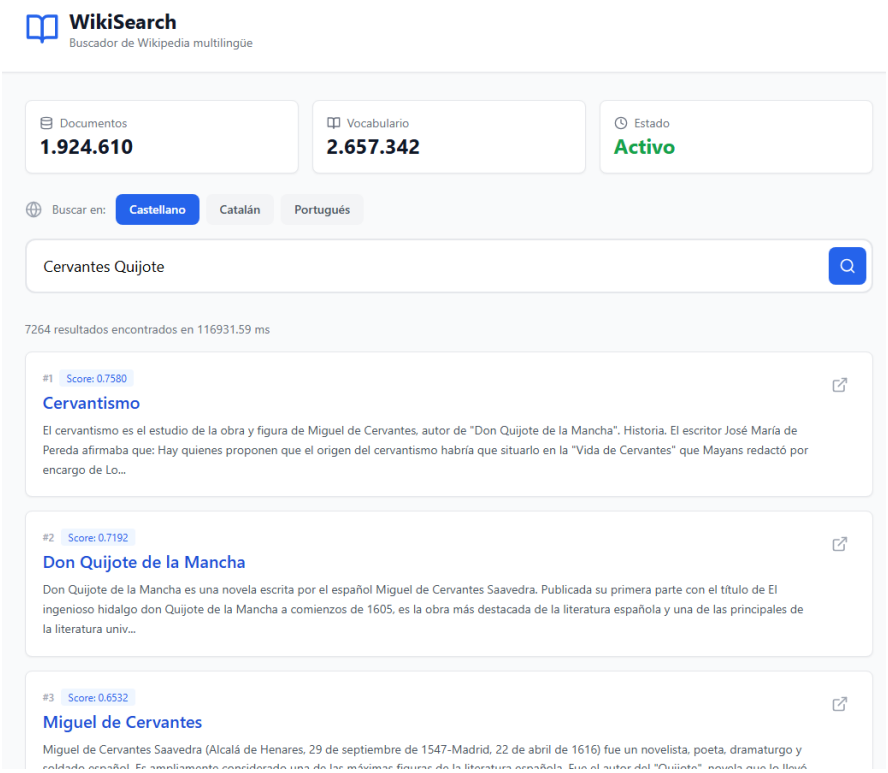

Figura 2: Primera búsqueda en castellano (carga inicial del índice).

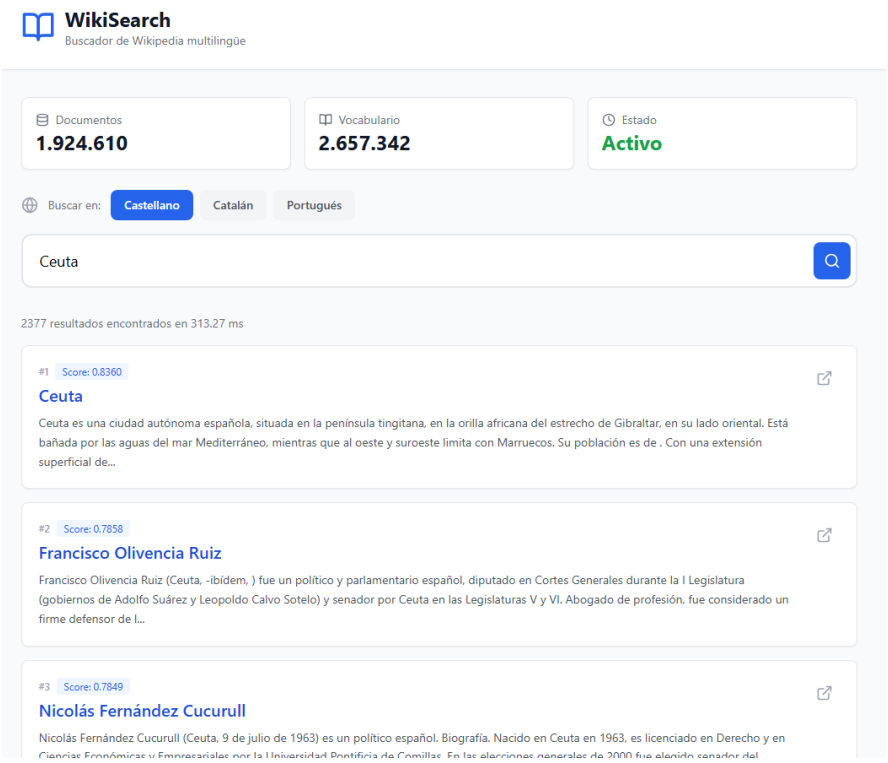

Figura 3: Segunda búsqueda en castellano con índice ya cargado (313.27 ms).

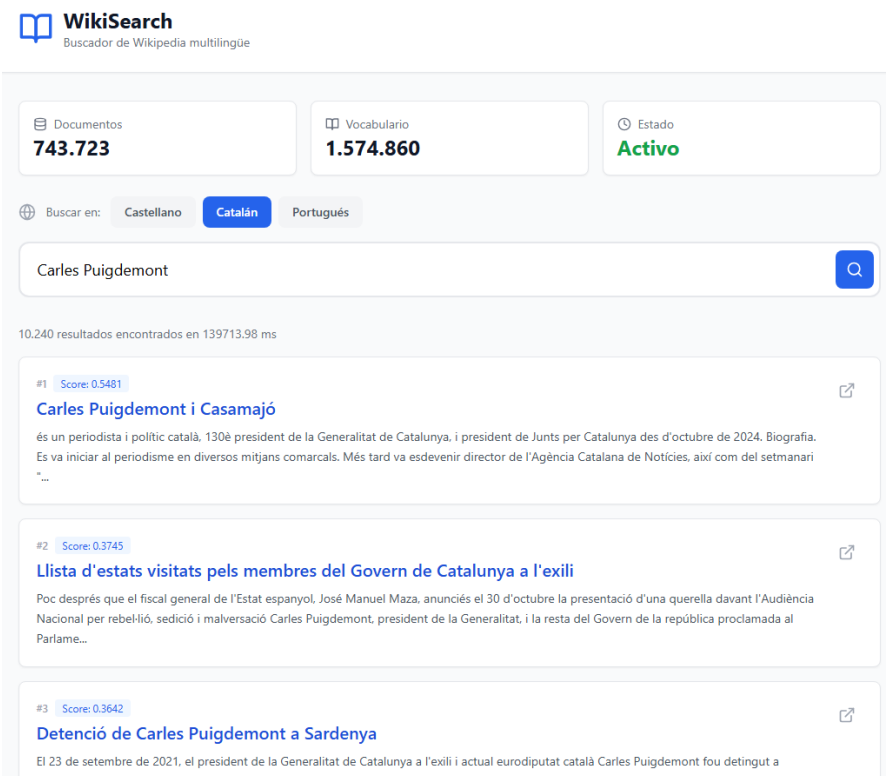

Figura 4: Primera búsqueda en catalán (carga inicial del índice).

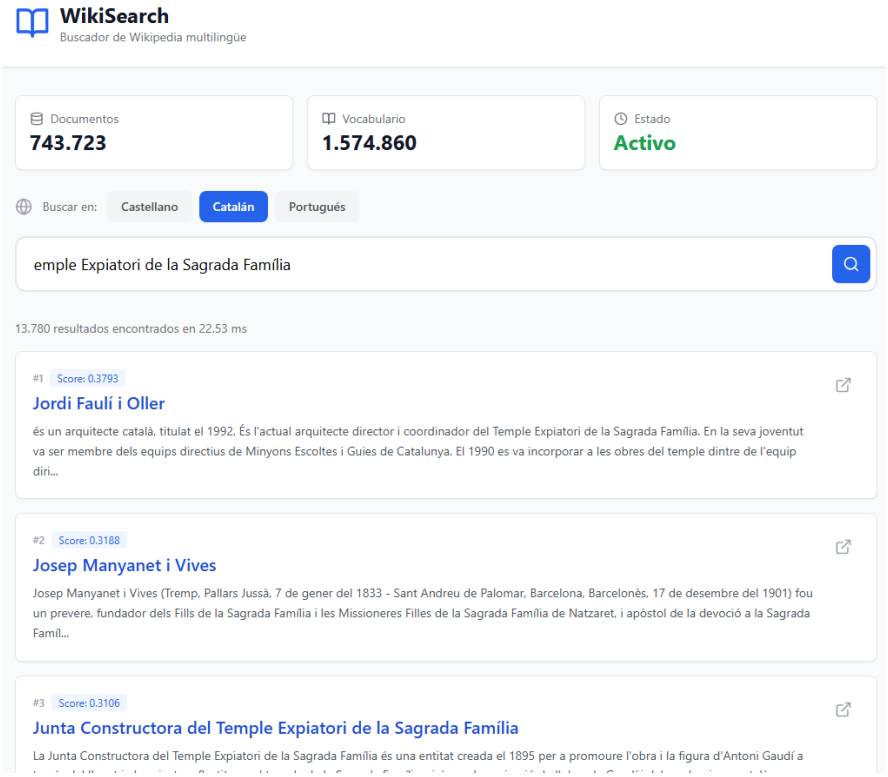

Figura 5: Segunda búsqueda en catalán con índice ya cargado (22.53 ms).

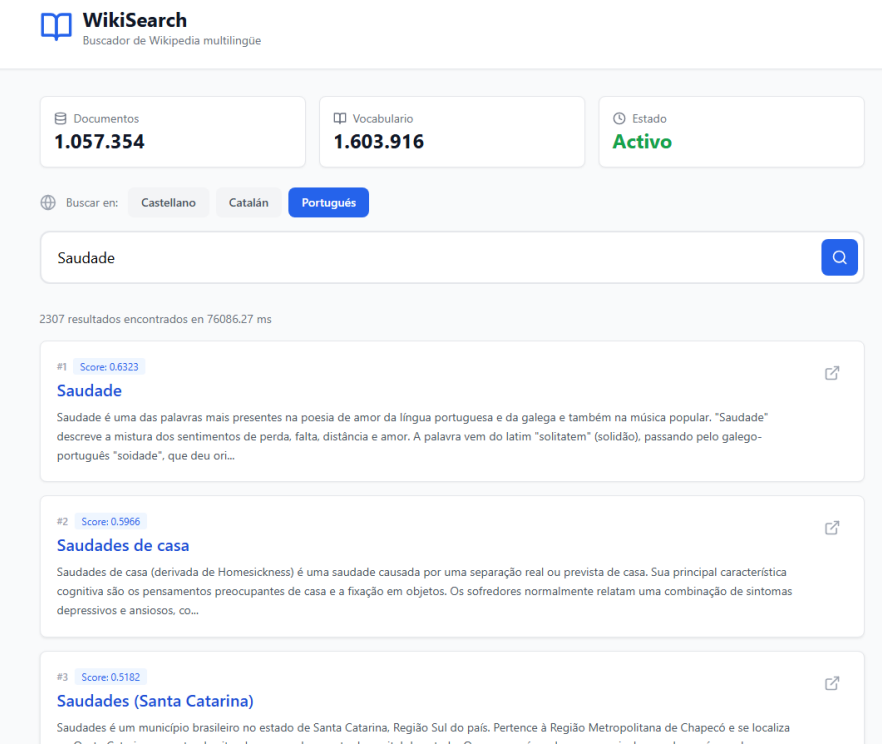

Figura 6: Primera búsqueda en portugués (carga inicial del índice).

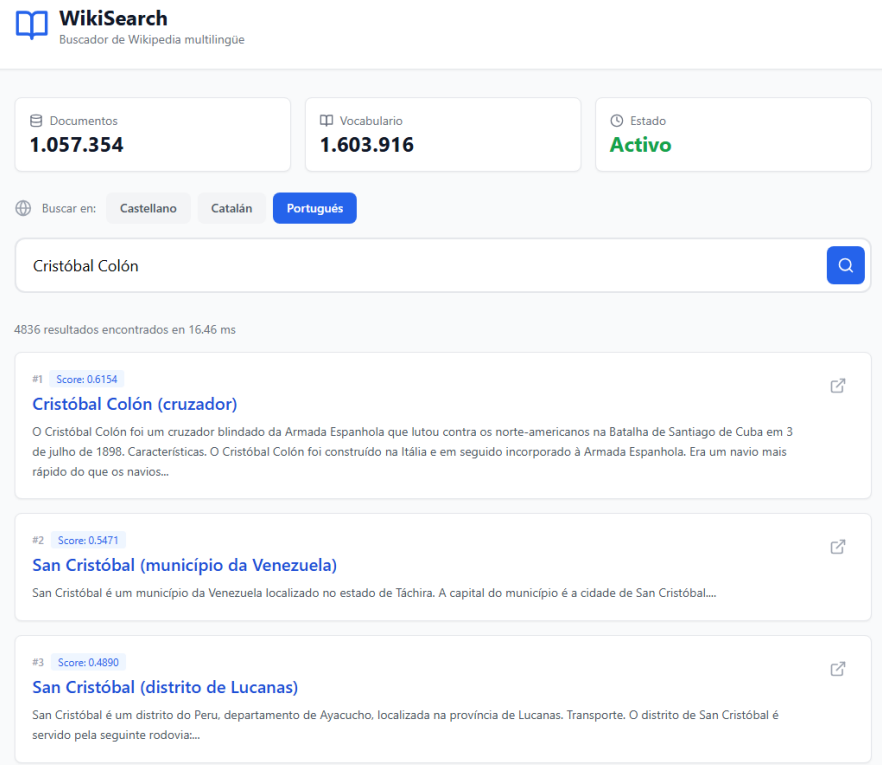

Figura 7: Segunda búsqueda en portugués con índice ya cargado (16.46 ms).

Las figuras anteriores demuestran la diferencia de rendimiento entre la primera búsqueda (que incluye la carga del índice desde disco) y las búsquedas posteriores (que utilizan el índice ya cargado en memoria). Los tiempos de respuesta se reducen drásticamente: de varios segundos en la carga inicial a apenas **313 ms** en castellano, **22 ms** en catalán y **16 ms** en portugués. Esta mejora evidencia la eficiencia del índice invertido en memoria para búsquedas en tiempo real.

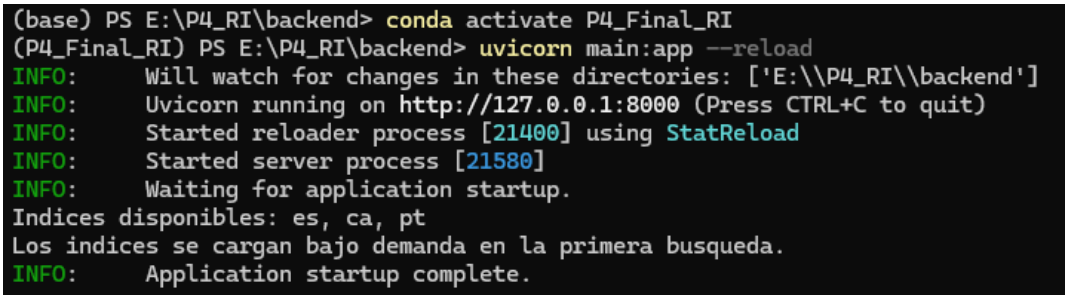

Figura 8: Consola del backend FastAPI en ejecución.

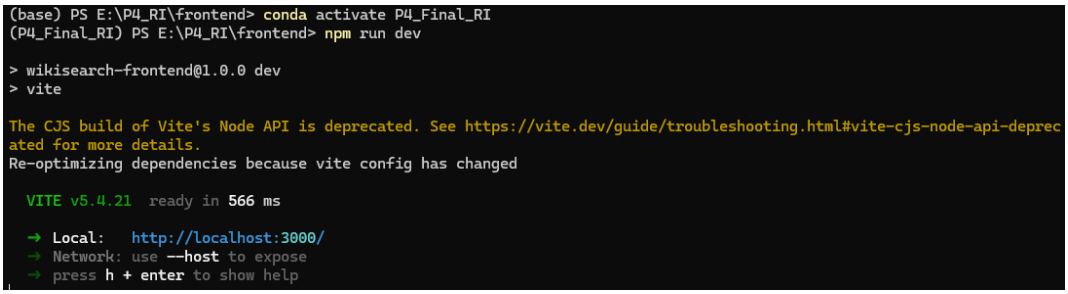

Figura 9: Consola del frontend Vite/React en ejecución.

Figura 10: Backend y Frontend ejecutado en Docker.

9. Problemas Encontrados y Soluciones

Durante el desarrollo del sistema se encontraron varios problemas técnicos que requirieron soluciones específicas.

9.1. Error de Multiprocessing en Windows

Problema: WikiExtractor fallaba con el error `ValueError: cannot find context for 'fork'`.
Causa: WikiExtractor usa `fork` para multiprocessing, que solo existe en Linux/Mac.
Solución: Usar WSL (Windows Subsystem for Linux) para la extracción con 4 procesos paralelos.

9.2. MemoryError durante Indexación

Problema: El script de indexación fallaba con `MemoryError` al procesar más de 3 millones de documentos.
Causa: El índice invertido completo no cabía en RAM (20-40 GB necesarios).
Solución: Dividir la indexación en scripts separados por idioma, cada uno guardando en su propio directorio (`index/es/`, `index/ca/`, `index/pt/`).

9.3. Alto Consumo de RAM al Cargar Índices

Problema: Cargar el índice de castellano consume 18 GB de RAM en pico.
Causa: El archivo del índice invertido (.pkl) contiene 2.6 millones de términos con millones de postings.
Solución: Los índices se cargan bajo demanda (no al iniciar el servidor). Para Docker, se recomienda usar los índices de catalán o portugués, que son más ligeros.

9.4. Mezcla de Idiomas en Resultados

Problema: Búsquedas en castellano devolvían resultados en portugués y catalán.
Causa: El índice original fue construido antes de separar los scripts por idioma, mezclando documentos.
Solución: Reconstruir el índice de castellano usando solo `extracted_es`.

10. Conclusiones

10.1. Logros del Proyecto

Se ha desarrollado un sistema completo de Recuperación de Información que cumple con todos los requisitos de la práctica:

- **Backend funcional**: API REST con FastAPI que implementa el pipeline completo de RI
- **Frontend moderno**: Interfaz web con React, TypeScript y TailwindCSS
- **Corpus significativo**: Más de 10 GB de artículos de Wikipedia en 3 idiomas
- **Índice eficiente**: Índice invertido con TF-IDF y similitud coseno
- **Despliegue con Docker**: Sistema containerizado para fácil despliegue

10.2. Estadísticas Finales

Métrica	Valor
Total de artículos indexados	3,725,687
Tamaño del corpus	10.5 GB
Tiempo total de indexación	4.6 horas
Idiomas soportados	3 (ES, CA, PT)
Endpoints de la API	9

Cuadro 3: Estadísticas finales del sistema.

10.3. Lecciones Aprendidas

- **Escalabilidad**: Trabajar con millones de documentos requiere estrategias de memoria cuidadosas
- **Separación por idiomas**: Mantener índices separados simplifica el mantenimiento y reduce el consumo de RAM
- **WSL para Windows**: Herramientas de Linux como WikiExtractor funcionan mejor en WSL que en Windows nativo
- **Docker**: Facilita el despliegue pero tiene limitaciones de memoria que afectan a índices grandes

11. Referencias

- Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.
- FastAPI Documentation: `https://fastapi.tiangolo.com/`
- React Documentation: `https://react.dev/`
- Wikimedia Downloads: `https://dumps.wikimedia.org/`
- NLTK Documentation: `https://www.nltk.org/`