27-5-2025

Buscador Web

Proyecto de prácticas (SAR)

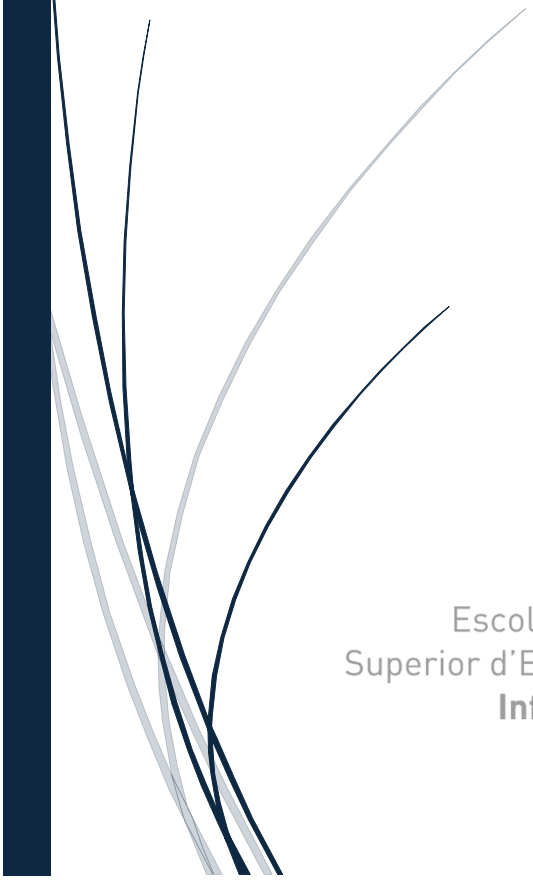
Jorge Rodríguez González

Julián Cussianovich Porto

Germán Soria Bustos

William Atef Tadrous

GRUPO 3CO11



Escola Tècnica Superior d'Enginyeria Informàtica

etsinf

Índice

1 Objetivo de la práctica

En esta práctica, nuestro objetivo es completar la librería de un programa de **indexación** y **recuperación** de artículos mediante consultas, capaz de devolver los resultados más relacionados con esta de manera rápida y eficiente. Consiste en dos partes principales y una ampliación, que se detallan a continuación. Estas son:

- El **indexador**: indexa los archivos JSON del directorio indicado y guarda el orden de procesamiento para su uso posterior.
- El **recuperador**: dada una **consulta**, recorre el índice generado y devuelve los artículos donde aparezcan las palabras o 'tokens' de la consulta.
- La búsqueda semántica: representa un proceso distinto, ya que no busca solo los tokens de la consulta, sino que **compara** la distancia semántica entre esta y los artículos, utilizando distintos modelos.

Este programa está estructurado en tres archivos principales: "SAR_Indexer.py", "SAR_Searcher.py", y "SAR_Lib.py"; los dos primeros corresponden al Indexador y Recuperador respectivamente, y el último es la librería que modificamos para **implementar** la **funcionalidad**. También hemos prestado mucha atención a que los algoritmos utilizados en este proyecto sean eficaces y de complejidad adecuada, ya que los conjuntos de documentos que puede procesar este programa podrían contener miles de artículos por documento JSON.

2 Organización

Para la colaboración y gestión de tareas en este proyecto, hemos utilizado **GitHub**. Esta plataforma nos ha permitido llevar un **control de versiones** del código, facilitar el trabajo en paralelo de los distintos miembros del equipo y organizar las tareas pendientes y sus respectivas asignaciones.

3 Funcionalidades básicas

A continuación, veremos cómo se han implementado las funciones básicas. Será en el apartado 3 donde mostraremos las modificaciones y adiciones que se han realizado sobre estas funcionalidades para la búsqueda semántica.

3.1 Indexación

El proceso de **indexación** es fundamental en cualquier sistema de recuperación de información, ya que permite organizar y preparar los datos para realizar búsquedas eficientes. La implementación de la función ***index_file()*** ha sido realizada por Jorge Rodríguez y Julián Cussianovich.

Objetivo: La función *index_file()* procesa un fichero de artículos *JSON* para crear un índice invertido, mapeando palabras a los artículos que las contienen (opcionalmente con

posiciones) y guardando metadatos. Para esto, asignamos un docid único a cada documento y recorremos sus líneas. Guardamos la información para cada artículo. Si es posicional, para cada token modificamos o creamos su *posting list* posicional; y si no es posicional, solo añadimos o creamos *posting list* de *artids*. Siempre añadimos los nuevos artículos o posiciones al final de las listas anteriores, para posteriormente realizar búsquedas de forma más eficiente.

Coste Asintótico $O(N)$: El coste asintótico para procesar un fichero individual es $O(N)$:, donde N representa su tamaño total (en caracteres o tokens). Esta complejidad lineal surge porque:

1. La función itera una vez por cada artículo del fichero.
2. Para cada artículo, operaciones como *self.parse_article* (procesamiento *JSON*) y *self.tokenize* (división en palabras) consumen tiempo proporcional al tamaño del artículo o al número de sus tokens.
3. La indexación de estos tokens en *self.index* (ya sea posicional o no) también es lineal respecto al número de tokens del artículo, ya que las inserciones en diccionarios y listas son $O(1)$ en promedio.

Al sumar estos costes por cada artículo, el total es proporcional al tamaño acumulado de todos los artículos y tokens del fichero, resultando en $O(N)$.

Eficiencia: La eficiencia de *index_file* radica en su procesamiento lineal del fichero y el uso de estructuras de datos (diccionarios/conjuntos) con operaciones promedio $O(1)$, lo que es óptimo para la indexación.

3.2 Recuperación

El recuperador realizará entonces la tarea de, dado un **índice** adecuado y una ***'query'* o consulta**, encontrar los artículos que contengan los ***'tokens'* o palabras** de la consulta y devolverlos en orden. Se debe ejecutar **estrictamente después** del indexador. Para esto hemos desarrollado una serie de funciones para trabajar con *posting lists* y resolver consultas.

3.2.1 Operaciones con posting lists

Las funciones descritas a continuación han sido implementadas por Jorge Rodríguez y Germán Soria.

Operaciones no posicionales:

1. ***And_posting***: La función *and_posting* calcula la **intersección** de dos *posting list*, *p1* y *p2* (ordenadas), devolviendo los identificadores de artículo (*artid*) comunes en ambas mediante un eficiente recorrido lineal. El coste asintótico de esta operación es $O(N_1 + N_2)$, donde N_1 es la longitud de la lista *p1* y N_2 es la longitud de la lista *p2*. Esta complejidad se justifica porque, en el peor de los casos, cada puntero recorre su lista respectiva por completo una sola vez. En cada paso de comparación, al menos un puntero avanza, asegurando que el número total de

operaciones sea proporcional a la suma de las longitudes de las listas. En caso de que los elementos de las *posting lists* comparados sean iguales, se añaden al resultado.

2. ***Minus_posting***: calcula la **diferencia** entre dos *posting list.* Lo hace recorriendo las 2 *posting lists* ordenadas, devolviendo los *artid* que están en *p1* y no en *p2*: si son iguales (los elementos de ambas *posting lists*), avanza; y si son diferentes (uno menor que otro), añade al resultado. Finalmente se añaden al resultado los elementos restantes de *p1* si los hubiera. De la misma forma que *and_posting*, presenta un coste lineal dependiente de la longitud de las listas.
3. ***Reverse_posting***: obtiene el **complemento** de una *posting list.* Como los 2 anteriores, presenta un coste asintótico lineal $O(M + N_p)$, con M igual a la cantidad de artículos indexados y N_p igual a la longitud de la *posting list* de entrada. De forma similar a la función *minus_posting*, primero obtiene todos los *artid* que se han indexado (*self.articles.keys()*), y hace un recorrido por la *posting list* y todos los *artid*, devolviendo los que no están en la *posting list*.

Todos estos algoritmos de tipo *merge* se han implementado siguiendo las directrices del contenido visto al final del tema 1 de la asignatura: 'Introducción a la recuperación de Información.'

Operaciones posicionales:

La función ***get_positionals()*** devuelve los *artid* donde una **secuencia** de *terms* aparece consecutivamente, usando intersecciones posicionales para frases.

Para calcular esto, en caso de que la secuencia *terms* tenga más de un término, hace llamadas a la función *interseccion_posicional_con_punteros()*.

Interseccion_posicional_con_punteros() es una **nueva función** que hemos añadido para calcular las *posting lists* y posiciones en las que un término sigue a otro. El proceso es similar al de un *and_posting*: Recorre las *posting lists* ordenadas, y para cada coincidencia, busca en las listas de posiciones (ordenadas también) dos posiciones contiguas de forma que la posición de la *posting list 1* sea igual a la de la *posting list 2* menos 1. En caso de una consulta con n términos, se llama a esta función n-1 veces. Para implementarla se ha recurrido al **algoritmo de intersección posicional** visto al final del tema 2 de la asignatura: 'Índices invertidos. Términos y consultas', con una distancia entre términos máxima (**tolerancia**) $k = 1$.

Es una de las **implementaciones más eficientes** (sin tener en cuenta técnicas más avanzadas como *Skip pointers*).

Con todo esto, el coste asintótico de *get_positionals()* es aproximadamente $\boldsymbol{O(k \cdot n \cdot l)}$, donde $\boldsymbol{k}$ es el número de términos, $\boldsymbol{n}$ el número de documentos en las posting lists y $\boldsymbol{l}$ la cantidad promedio de posiciones por documento; tratándose de un coste lineal compuesto o pseudo-lineal.

La función ***get_posting()*** **obtiene la** *posting list* **para un término dado**. Si el índice es posicional (*self.positional*), llama a *self.get_positionals()* con el término.

En caso contrario (índice no posicional), recupera la lista directamente de *self.index*. Si el término no se encuentra, retorna una lista vacía.

3.2.2 Resolución de consultas

La función ***solve_query()*** (implementada por William Atef) es el motor principal para **procesar las consultas** de los usuarios, encargada de interpretar una cadena de consulta booleana y devolver la lista de *artids* que la satisfacen. Esta función maneja:

- Términos individuales.
- Frases exactas entre comillas (ej: "inteligencia artificial"), que deben aparecer consecutivamente.
- El operador *NOT* para exclusión.
- Un *AND* implícito entre los demás componentes de la consulta.

Proceso de Resolución de Consultas:

1. **Tokenización**: Inicialmente, la consulta (*query*) se descompone en una lista de tokens. Esto permite separar frases completas (entre comillas) de términos sueltos y operadores como *NOT*.
2. **Procesamiento Iterativo**: La función itera sobre estos tokens:
    - **Operador *NOT***: Si se encuentra *NOT*, se procesa el siguiente token (término o frase). Su *posting list* se invierte usando *self.reverse_posting()*.
    - **Frases entre comillas**: Para frases exactas, se utiliza *self.get_positionals()* para obtener los *artid* donde la secuencia de términos aparece consecutivamente.
    - **Términos individuales**: Para un término suelto, *self.get_posting()* recupera su *posting list*.
    - **Combinación de Resultados (*AND* implícito)**: La *posting list* del token actual (o su versión invertida por *NOT*) se combina con los resultados acumulados de tokens anteriores mediante *self.and_posting()*. Si es el primer elemento procesado, establece el resultado base.
3. **Retorno**: La función devuelve una tupla conteniendo la *posting list* final con los *artid* resultantes y un diccionario vacío, previsto para compatibilidad futura.

3.2.3 Mostrar resultados

La función *solve_and_show()* (implementada por William Atef) ejecuta una consulta, muestra los **resultados formateados** y devuelve el número total de hallazgos**.** Primero, llama a *self.solve_query()* para obtener los *artid* correspondientes a la *query*. Informa del total de artículos recuperados y luego muestra una selección de estos (todos si *self.show_all* está activo, o hasta *self.SHOW_MAX*), presentando para cada uno su *artid*, título y URL extraídos de *self.articles*. Finalmente, retorna el conteo total de resultados.

La función *show_stats()* (implementada por Germán Soria) presenta **estadísticas resumidas** sobre el estado del motor de búsqueda y sus índices**.** Imprime en la consola métricas clave como:

- El número total de artículos indexados (*self.articles*).
- El total de archivos procesados (*self.docs*).
- El tamaño del vocabulario, es decir, términos únicos en *self.index*.
- La cantidad de *URLs* únicas (*self.urls*). Además, indica el estado (Activado/Desactivado) de los índices posicional (*self.positional*) y semántico (*self.semantic*).

4 Ampliación: búsqueda semántica

La **ampliación** propuesta para esta práctica consiste en añadir al indexador y recuperador la capacidad de resolver consultas de **búsqueda semántica**. Para poder implementarla, se ha proporcionado una nueva librería de *SAR_semantics.py*, que contiene métodos que interactúan con el **modelo semántico** que se seleccione al ejecutar programa, mediante la función *create_semantic_model()*. Estos nos permiten calcular las distancias semánticas entre frases y palabras, proceso necesario para recuperar los artículos con mayor similitud semántica con la consulta, independientemente de que contengan o no los términos de búsqueda.

Las modificaciones en las funciones *index_dir()*, *index_file()*, y *solve_query()* para llevar a cabo la ampliación semántica las ha realizado Jorge Rodríguez; y la implementación de las funciones *update_chuncks()*, *create_kdtree()*, *solve_semantic_query()* y *semantic_reranking()* la ha hecho William Atef.

Para hacer uso de **modelos semánticos**:

En la función *index_file()*, si la indexación semántica está activada, llamamos al método *update_chuncks()* pasándole el contenido del artículo y su *artid*. Esta función extrae las frases utilizando *sent_tokenize()* y **actualiza** los **atributos** *self.chuncks*, *self.chunck_index* y *serf.artid_to_emb*; para guardar la **relación entre un artículo y sus frases**, entre otras cosas.

Posteriormente, en el método *index_dir()* llamamos al método *create_kdtree()* si y solo si la búsqueda semántica está **activa** (se ha introducido el argumento -S) y tenemos *chuncks* almacenados.

KD-Tree (o árbol k-dimensional) es una estructura de datos utilizada para realizar **búsquedas rápidas de similitud** entre vectores en espacios de alta dimensión. En la búsqueda semántica, cada **frase** (*chunck*) de los artículos se representa como un **vector de embeddings** (usando modelos como SBERT, Beto, etc.).

La **creación** de esta estructura de datos se consigue llamando al método *create_kdtree()*, que llama al método *fit()* del **modelo semántico** y crea el *KD-Tree* a partir de los *chuncks* guardados en el indexador (como previamente mencionamos).

De esta forma obtenemos los datos necesarios a partir de los artículos para realizar las ampliaciones semánticas.

4.1 Recuperación por similitud semántica

Para implementar la **búsqueda semántica**, en la función *solve_query()* comprobamos si se ha introducido el argumento -S (*semantic*). Si ha sido así, pasamos a utilizar el método *solve_semantic_query()*, que resuelve la consulta utilizando el modelo semántico.

El procedimiento (al emplear dicha función) comienza realizando una **consulta inicial al** *self.model* para obtener un primer conjunto de *chuncks* (fragmentos de texto) relevantes. La cantidad de *chuncks* solicitados (*top_k*) parte de *self.MAX_EMBEDDINGS* y se ajusta al total disponible. Si los chuncks recuperados están dentro del umbral de similitud (*self.semantic_threshold*), la función **expande la búsqueda iterativamente**: incrementa *top_k* y vuelve a consultar al modelo. Este ciclo busca asegurar que no se omitan *chuncks* relevantes que superen el umbral establecido.

Una vez completada la fase de búsqueda (que puede haber sido expansiva o no), se aplica un **filtrado final**: todos los *chuncks* recuperados se evalúan, y solo aquellos cuya distancia semántica a la *query* es menor o igual al *self.semantic_threshold* son conservados.

Finalmente, los índices de estos *chuncks* relevantes y filtrados se **convierten en *artids* únicos**. Esto se logra utilizando la correspondencia almacenada en *self.chunck_index* (que vincula cada *chunck* con su artículo de origen), garantizando que la *posting_list* devuelta contenga cada *artid* una sola vez.

4.2 Ordenación por similitud semántica

El método *semantic_reranking()* reordena los artículos de una búsqueda booleana aplicando un criterio de similitud semántica de forma iterativa. Tras calcular el *embedding* de la consulta, la función inicia un bucle que ajusta dinámicamente un *top_k* de *chuncks* a considerar.

En cada iteración de este bucle, se consultan desde el *KD-Tree* los *top_k chuncks* más similares a la consulta. Con estos *chuncks*, se construye desde cero una lista provisional de artículos reordenados (*ranked_articles*), incorporando aquellos artículos de la lista de entrada que se encuentren en los *chuncks* de la iteración actual y **manteniendo el orden semántico**. Si esta lista provisional logra incluir todos los artículos de la entrada original, la función **retorna inmediatamente** esta *ranked_articles* **optimizando el proceso**.

Si no se cubren todos los artículos y *top_k* aún no ha alcanzado el total de *chuncks* disponibles en el corpus, se **incrementa *top_k*** y el ciclo se repite con una nueva consulta. El bucle finaliza si se da la condición de salida temprana o si *top_k* alcanza su valor máximo.

Finalmente, si el bucle concluyó porque *top_k* llegó al máximo (sin una cobertura total previa), los artículos de la lista de entrada que no fueron incluidos en *ranked_articles* se añaden al final. Es una medida que hemos incluido por **robustez del código**, ya que en un principio no debería ocurrir nunca.

También barajamos la posibilidad de **realizar una única consulta** al *KD-Tree* con *k = len(self.chuncks)*. Obtuvimos tiempos de ejecución de búsqueda similares o incluso menores, con un código mucho más simple, pero finalmente nos hemos limitado a la descripción que se nos proporcionaba de la función. Respecto a esto, debemos destacar una cuestión. Cuando iteramos buscando el primer valor de *top_k* que nos permita recuperar *chuncks* para todas las frases, en unas pocas iteraciones normalmente encuentra para todos los artículos de los test proporcionados. Observamos que hay un **cambio** en el orden de 2 *artids* de una de las pruebas; un cambio de orden que no ocurre si en vez de realizar varias iteraciones aumentando *top_k*, usamos una única consulta, como mencionamos previamente.

Por último, para implementar esta ampliación se ha modificado la función *solve_query()* para que, si *semantic_ranking* está activado, llame a la función *semantic_reranking()* para **reordenar el resultado** de la consulta antes de devolverlo.