Sistema Mejorado de Recuperación de Información

Prácticas de Sistemas de Recuperación de Información

Alejandro Arroyo Loaisa

Prácticas de SRI

Universidad de Jaén - 2024

ÍNDICE DE CONTENIDOS

ÍNDICE DE CÓDIGOS

ÍNDICE DE DIAGRAMAS

Sistema Mejorado

Una vez se ha terminado el Sistema Básico y se ha comprobado su correcto funcionamiento, es hora de introducir mejoras en el sistema que no solo agilicen el procesamiento de los datos, sino que además hagan que consiga devolver mejores documentos. Esto es, mejorar la **eficiencia** y mejorar la **eficacia**, aumentando la efectividad del sistema.

En este documento se presentarán las mejoras implementadas. Dicho Sistema Mejorado es el que se presentará a la competición de la asignatura.

Mejoras de Eficiencia

La eficiencia es la capacidad de lograr los resultados deseados con el mínimo posible de recursos, en este caso tiempo y espacio. Durante la construcción del Sistema Básico, ya se eligieron las estructuras más rápidas, ligeras y adecuadas para nuestro objetivo, por lo que será cuestión de estudiar cómo reducir aún más el tiempo de ejecución y el tamaño del volumen de datos, si es posible.

Mejora 1: Unificación de algunos bucles

En muchas partes del código, la información a veces se recorría más veces de las necesarias con el uso de bucles. Se utilizaban dos o más bucles que comprendían las mismas condiciones de parada y recorrían exactamente los mismos datos, en vez de unificarlos y que se tenga el mismo funcionamiento pero con menos iteraciones. Explicar esto es más sencillo con un ejemplo:

→Quiero dividir cada frecuencia de palabra entre su frecuencia máxima de fichero y multiplicar por el idf:

Código 1 - Pseudocódigo unificación de bucles

```
ANTES:
      Bucle recorriendo todas las palabras
            Divido cada frecuencia de palabra entre su máxima
      Fin de Bucle
      Bucle recorriendo todas las palabras
            Multiplico por el idf
      Fin de Bucle

AHORA:
      Bucle recorriendo todas las palabras
            Divido cada frecuencia de palabra entre su máxima
            Multiplico por el idf
      Fin de Bucle
```

Pasando de un $O(n+n+...) = O(a*n)$ siendo a el número de bucles usados, hasta $O(n)$. Esto en el código, y ya no tan simplificado, se refleja de la siguiente manera:

- **ANTES:** Cuatro bucles que recorren todas las palabras del índice → $O(4*n)$

```python
for palabra in indice:
    for nombre_fichero in indice[palabra]:

        if nombre_fichero not in frec_max:
            frec_max[nombre_fichero] = indice[palabra][nombre_fichero]["reps"]

        else:
            if frec_max[nombre_fichero] < indice[palabra][nombre_fichero]["reps"]:
                frec_max[nombre_fichero] = indice[palabra][nombre_fichero]["reps"]

for palabra in indice:
    for nombre_fichero in indice[palabra]:

        indice[palabra][nombre_fichero]["reps"] / frec_max[nombre_fichero]


for palabra in indice:

    num_ficheros_aparicion = len(indice[palabra])
    idf = math.log2(num_ficheros_coleccion / num_ficheros_aparicion)

    for nombre_fichero in indice[palabra]:

        indice[palabra][nombre_fichero]["reps"] = frec_max[nombre_fichero] * idf

    indice[palabra] = { "idf": idf, "ficheros": indice[palabra]}

for palabra in indice:

        vector = [indice[palabra]["ficheros"][nombre_fichero]["reps"] for nombre_fichero in indice[palabra]["ficheros"]]
        modulo = math.sqrt(sum(value**2 for value in vector))

        for nombre_fichero in indice[palabra]["ficheros"]:
            indice[palabra]["ficheros"][nombre_fichero]["reps"] = indice[palabra]["ficheros"][nombre_fichero]["reps"] / modulo
```

Código 2 - Ejemplo SIN unificación de bucles

- **AHORA:** Un bucle que recorre todas las palabras del índice → $O(n)$

```python
for palabra in indice:
    for nombre_fichero in indice[palabra]:

        if nombre_fichero not in frec_max:
            frec_max[nombre_fichero] = indice[palabra][nombre_fichero]["reps"]

        else:
            if frec_max[nombre_fichero] < indice[palabra][nombre_fichero]["reps"]:
                frec_max[nombre_fichero] = indice[palabra][nombre_fichero]["reps"]


    num_ficheros_aparicion = len(indice[palabra])
    idf = math.log2(num_ficheros_coleccion / num_ficheros_aparicion)

    for nombre_fichero in indice[palabra]:

        indice[palabra][nombre_fichero]["reps"] = (indice[palabra][nombre_fichero]["reps"] / frec_max[nombre_fichero] )* idf

    indice[palabra] = { "idf": idf, "ficheros": indice[palabra]}

    vector = [indice[palabra]["ficheros"][nombre_fichero]["reps"] for nombre_fichero in indice[palabra]["ficheros"]]
    modulo = math.sqrt(sum(value**2 for value in vector))

    for nombre_fichero in indice[palabra]["ficheros"]:
        indice[palabra]["ficheros"][nombre_fichero]["reps"] = indice[palabra]["ficheros"][nombre_fichero]["reps"] / modulo
```

Código 3 - Ejemplo CON unificación de bucles

Mejora 2: Reducción en el número de cálculos

Se observó que algunas operaciones relativamente costosas que eran invariables a lo largo del tiempo estaban dentro de bucles, recalculándose para obtener exactamente el mismo resultado en cada iteración. Esto es muy ineficiente y va en detrimento del tiempo de ejecución. Por ejemplo:

- La lectura del listado de stopwords se hacía para cada consulta procesada, se extrajo esa operación del bucle para que solo se realizase una vez.

- La lectura del diccionario se hacía para cada expansión de consulta por sinónimos, siendo el diccionario invariable a lo largo del tiempo. Se extrajo esa operación del bucle para que solo se realizase una vez.

Entre otros. Todos los datos que sean invariables y se deban leer o calcular, se hace dicha operación exactamente una vez.

Mejora 3: Eliminación de la modularización

Como la Práctica 1 fue incremental, se planteó un sistema formado por módulos, y cada módulo añadía una funcionalidad más al sistema, hasta su completitud. Entre cada módulo, por tanto, se realizaban lecturas y escrituras de ficheros de todos los datos procesados hasta el momento, siendo esto impracticable para volúmenes de datos como el de la competición. Debido a que:

- Leer y escribir 1000 pequeños ficheros por cada módulo es ineficiente, pero sigue siendo un tiempo abarcable (*Práctica 1*).
- Leer y escribir 200.000 pequeños ficheros por cada módulo simplemente es impracticable (*Competición*).

El sistema de esta manera, tarda más tiempo en leer y escribir todos los datos que en procesarlos.

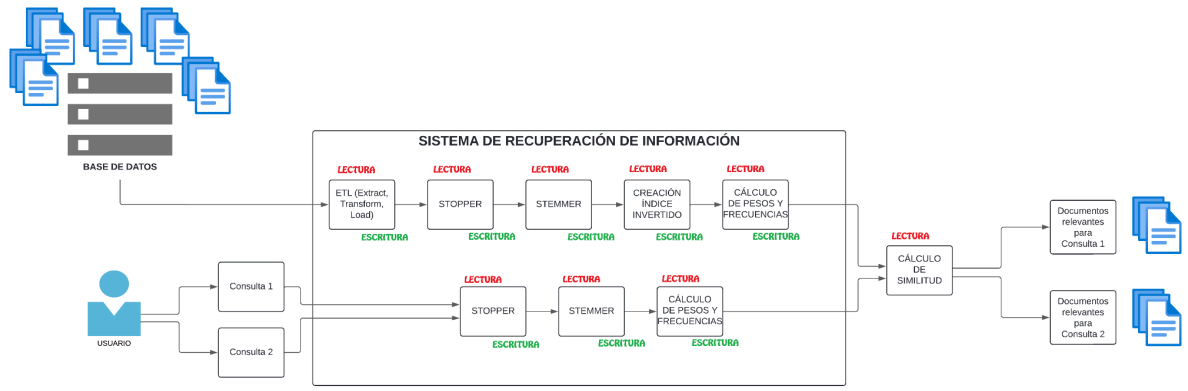

Diagrama 1 - Sistema CON modularización

Por lo tanto, el sistema fue modificado para **eliminar los módulos y las lecturas y escrituras innecesarias**, realizando solo una lectura de los documentos y solo una lectura de las consultas, y arrastrando todos los datos durante el procesamiento y hasta el final, desde un único script, devolviendo los resultados hasta por pantalla, sin escrituras (aunque para la competición exclusivamente esto se ajustó para poder generar el ficheros de resultados y participar), reduciendo el tiempo de ejecución significativamente.

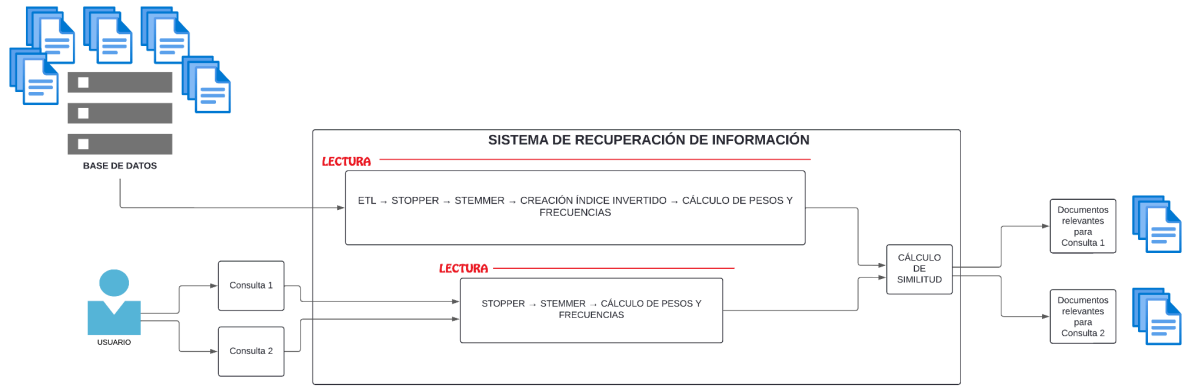

Diagrama 2 - Sistema SIN modularización

Mejoras de Eficacia

Una vez se ha desmodularizado el código y aumentado su eficiencia, vamos a tratar de mejorar los resultados obtenidos, esto es, **mejorar la eficacia**. Desde la asignatura se proporcionó un documento con un listado de posibles mejoras para el sistema, y se han implementado las siguientes:

Mejora 1: Aplicar pseudo-realimentación por relevancia (PRF)

Esta técnica trata de mejorar la precisión de la consulta al **agregar términos relevantes relacionados**. Se puede realizar de muchas maneras, pero la elegida se describe a continuación:

Se lanza una consulta, se obtienen los 5 documentos más similares y se añaden a la consulta original cinco términos de cada uno de ellos. En total 25. Volviendo a lanzar la consulta, esta vez con esta adición de estas palabras.

La idea es que si se obtuvieron esos documentos, es porque las palabras de la consulta aparecen en ellos y, por lo tanto, el resto de palabras de estos documentos estarán relacionadas. Si se añaden a la consulta puede añadir contexto a esta. Pero se debe tener cuidado en la selección porque puede perderse la intención original de la consulta.

Se ha tratado de aplicar de dos maneras:

- **Cogiendo las 5 palabras con mayor Frecuencia de Término (TF):**

Mide la frecuencia con la que un término aparece en un documento. Por lo tanto, se puede asumir que la repetición de una palabra le da importancia y peso en el contexto de la consulta.

- **Cogiendo las 5 palabras con mejor TF-IDF:**

Mide la importancia de un término en el corpus de documentos, mediante la siguiente idea: si un término aparece con frecuencia en un documento pero es poco común en el corpus en general.

Tras varias pruebas, los mejores resultados se obtuvieron con la segunda manera. Ya que se añaden términos importantes pero poco comunes, especificando la consulta.

Mejora 2: Expansión de consulta con sinónimos

Las consultas no suelen contener muchas palabras, y estamos limitados a la aparición expresa de los términos de la consulta en los documentos. Expandir la consulta con palabras parecidas puede ayudar a mejorar la cobertura. Esto se puede hacer, por ejemplo, con la **adición de sinónimos** a la consulta, que es la que se ha aplicado..

En este caso, se expande cada consulta entre 0 y 5 sinónimos por palabra, habiendo quitado ya las stopwords. El número de sinónimos que se añadirán finalmente vendrá determinado por la cantidad de sinónimos albergados en el diccionario.

Se ha tratado de conectar con APIs que dada una palabra te devuelva sus sinónimos, pero o son absurdamente lentas (entre 3-10s para devolver un conjunto de sinónimos), o no funcionan. Recordemos que nuestro sistema está en castellano y que el uso de este idioma disminuye el material en línea respecto del que apoyarse. Descartada la opción de traducir las palabras al inglés, encontrar sinónimo y traducirla de vuelta, no devuelve palabras adecuadas en el 90% de los casos y es muy lento.

Así que se ha descargado este diccionario de sinónimos en *.json* del siguiente enlace y tiene el siguiente formato:

https://github.com/edublancas/sinonimos/

```
[
  ["Biblia", "Sagrada Escritura", "Antiguo Testamento", "Nuevo
Testamento"],
  ["CD-ROM", "CD", "disco compacto"],
  ["Creador, constructor, demiurgo, artista", "Hacedor", "Edificador,
artífice"],
  ["Dios", "Todopoderoso", "Jehová", "Altísimo", "Alá"],
  ["Jesucristo", "Dios", "Jesús", "Cristo", "Nazareno", "Redentor",
"Mesías", "Salvador", "Señor"],
  ["Pelota", "Balón", "esférico", "bola", "canica", "globo"],

  ...

  ["Representar", "simbolizar, figurar, personificar, aparentar",
"encarnar, personalizar"]
]
```

Código 4 - Formato del diccionario descargado

Es una lista de listas de sinónimos. Cada lista es un conjunto de palabras que son sinónimos entre sí. Este formato de fichero, lista de listas, es impracticable para la búsqueda sinónimos, ya que en el peor de los casos te obliga a recorrer la información entera para lograr encontrar un elemento, $O(n*m)$. La solución a este problema sería crear un programa que recorra el documento entero, y almacene CADA palabra con su conjunto de sinónimos en un objeto diccionario, usando la propia palabra como clave.

De esta manera, con una sola lectura del documento al inicio del procesado de consultas, conseguiríamos crear un diccionario de sinónimos en el que la búsqueda de elementos es de eficiencia $O(1)$, incluso en el peor de los casos (el acceso a memoria en los diccionarios es inmediato, debido a la naturaleza hash de los mismos).

Con esto, la estructura resultaría de la siguiente manera:

```
{
   “Biblia”:  [“Sagrada  Escritura”,  “Antiguo  Testamento”,  “Nuevo
Testamento”],
   “Sagrada  Escritura”:  [“Biblia”,  “Antiguo  Testamento”,  “Nuevo
Testamento”],
   “Antiguo  Testamento”:  [“Biblia”,  “Sagrada  Escritura”,  “Nuevo
Testamento”],
      “Nuevo  Testamento”:  [“Biblia”,  “Sagrada  Escritura”,  “Antiguo
Testamento”],
. . .
}
```

Código 5 - Formato del diccionario generado

Obsérvese que para cada palabra hay una entrada en el diccionario, pese a la repetición de datos, respecto al formato anterior. Esto hace que el diccionario ocupe algo más en memoria, pero el acceso a la información sea inmediato, lo que nos conviene. Además se les aplica el mismo proceso de STOPPER y STEMMER que al corpus y consultas.

Una vez generado. Lo almacenamos en el directorio del sistema, y al arranque de la ejecución del mismo, hacemos que se lean los datos. Posteriormente en el procesado de consultas, hacemos que después de quitar las stopwords se mire palabra a palabra en la consulta, se busque en el diccionario y se añadan tantos sinónimos como sea posible, hasta un máximo de 5 por palabra, según existan.

El diccionario y código para generarlo ha sido entregado con esta práctica.

Mejora 3: Distancia BM25

La métrica *TF.IDF* no es la más utilizada actualmente, otras han demostrado funcionar mejor, como la **BM25**, ya que ajusta dinámicamente el *IDF* en función de la longitud de los documentos.

La fórmula del BM25 para calcular la similitud es la siguiente:

$$score(D,\ Q)\ =\ \sum_{i=1}^{n} IDF(q_i)\ \cdot\ \frac{f(q_i,\ D)\cdot(k_1+1)}{f(q_i,\ D)\ +\ k_1\cdot(1-b+b\cdot\frac{|D|}{avgdl})}$$

$$IDF(q_i) = log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}$$

Para llevar a cabo esta tarea, se ha utilizado la librería rank_bm25. Contiene distintas variantes de BM25, pero la que mejor ha resultado para nuestra colección ha sido **BM25Okapi**. Se importa en el proyecto con:

```
from rank_bm25 import BM25Okapi
```

Esta librería es muy sencilla y cómoda de utilizar. Para un corpus dado, en este caso nuestro índice tras todo el procesamiento, y una consulta dada, también tras aplicar el mismo procesado que al corpus, puedo obtener la similitud entre consulta y documentos con dos líneas:

```
objetoBM25 = BM25Okapi(corpus_procesado)
docs_similitud = bm25.get_scores(consulta_procesada)
```

Esto creará un objeto BM25Okapi con la información de nuestros documentos y, para una consulta que le indiquemos, devolverá un array con las similitudes para todos los documentos (sin ordenar).

Otra opción es la siguiente, que te devuelve ya el conjunto de *n* documentos que más similitud tienen con la consulta:

```
objetoBM25 = BM25Okapi(corpus_procesado)
docs_similitud = bm25.get_stop_n(consulta_procesada,corpus_procesado,n=1)
```

Conclusiones

Tras la realización de la mejora del sistema para la Práctica 2, en conjunto con la participación en la competición y mi ambición de no quedar fuera del podio, he llegado a las siguientes conclusiones tratando de mejorar mi sistema:

- Los tiempos de ejecución y la carga de procesamiento aumentan considerablemente cuando también lo hace el tamaño del corpus de documentos.

- Desmodularizar el sistema y eliminar las lecturas y escrituras innecesarias aligera los tiempos de ejecución notablemente.

- Las mejoras que expanden la consulta no son intrínsecamente buenas debido a que, al no analizar la semántica e intención de las consultas, se pierde la intención de la búsqueda y se añaden palabras que la dificultan. Se podría conseguir aplicando PLN.

- La medida BM25 para calcular la similitud de los documentos mejoró bastante al sistema, respecto de TF.IDF. Tener en cuenta la longitud de los documentos en el procesamiento aumentó la precisión.

- Trabajar con el idioma castellano es realmente complicado, respecto del inglés. No hay tanto material de apoyo en Internet (mención del diccionario).