Proyecto Final de Sistemas de Recuperación de Información

Rocio Ortiz Gancedo and Carlos Toledo Silva

Universidad de La Habana, Cuba

Resumen Desde el surgimiento del almacenamiento de información, poder acceder a esta de forma cómoda y eficiente cuando se necesite, ha sido de gran importancia. Con la aparición de las computadoras esto se ha podido automatizar mediante programas, conocidos como Sistemas de Recuperación de Información. El proyecto que se presenta constituye un Sistema de Recuperación de Información para un conjunto de documentos utilizando el modelo vectorial. En este artículo se describe todo el proceso de diseño, implementación y análisis de los resultados del sistema.

1. Introducción

La búsqueda y recuperación de información es la ciencia que estudia la manera de hallar los datos en documentos electrónicos y cualquier tipo de colección documental digital.

Inicialmente el manejo a grandes volúmenes de información ocurría con las bibliotecas. Para acceder cómodamente a los libros y artículos se mantenían indexados usando catálogos. Desde 1891 se comienzan a utilizar equipos para automatizar la búsqueda en los mismos.

En los años 40 se emplea por primera vez el término Recuperación de Información y ya en los 50 comienzan a usarse computadoras para este fin.

El proceso de recuperación de información comienza cuando un usuario hace una consulta al sistema. Dentro de los modelos de Sistemas de Recuperación de Información encontramos el modelo vectorial que fue el que se desarrolló en este proyecto que a continuación se presenta.

El objetivo de este proyecto es brindar un Sistema de Recuperación de Información para una colección de datos dada y presentar un informe donde se explique bien el mismo.

El resto del documento está dividido en seis secciones principales. En la sección 2 se incluye el diseño del sistema, donde se explica brevemente este modelo, las razones para la selección del mismo y algunas ideas interesantes que se tuvieron en cuenta en la concepción del sistema. La sección 3 aborda la implementación del sistema, explicando cómo se procesaron y se representaron los documentos y consultas. Se incluye también la explicación de como se hace el análisis de similitud entre las consultas y los documentos. Además se explica cómo se realizó la retroalimentación y muestra el funcionamiento de la interfaz

gráfica. La sección 4 evalúa el sistema que se implementó, muestra las medidas usadas para esto y los resultados obtenidos. Estas evaluaciones se realizaron utilizando la colecciones **Cranfield** y **Medline**. La sección 5 realiza un análisis crítico de las ventajas y desventajas de la aplicación. Por último en la sección 6 y 7 se encuentran las conclusiones y recomendaciones respectivamente.

2. Diseño del sistema

2.1. Recordatorio de las características del modelo vectorial

En el modelo vectorial, el peso $w_{i,j}$ asociado al par (t_i, d_j) (siendo t_i el término i y d_j el documento j) es positivo y no binario. A su vez, los términos en la consulta están ponderados. Sea $w_{i,q}$ el peso asociado al par (t_i, q) (siendo q una consulta), donde $w_{i,q} \geq 0$. Entonces, el vector consulta q se define como $\overrightarrow{q} = (w_{1q}, w_{2q}, ..., w_{nq})$ donde n es la cantidad total de términos indexados en el sistema. El vector de un documento d_j se representa por $\overrightarrow{d_j} = (w_{1j}, w_{2j}, ..., w_{nj})$.

La correlación se calcula utilizando el coseno del ángulo comprendido entre los vectores documentos dj y la consulta q.

$$sim(d_j, q) = \frac{\overrightarrow{d_j} \cdot \overrightarrow{q}}{|\overrightarrow{d_j}| \cdot |\overrightarrow{q}|} \tag{1}$$

$$sim(d_j, q) = \frac{\sum_{i=1}^{n} w_{i,j} \cdot w_{i,q}}{\sqrt{\sum_{i=1}^{n} w_{i,j}^2} \cdot \sqrt{\sum_{i=1}^{n} w_{i,q}^2}} \tag{2}$$

Sea $freq_{i,j}$ la frecuencia del término t_i en el documento d_j. Entonces, la frecuencia normalizada $tf_{i,j}$ del término t_i en el documento d_j está dada por:

$$tf_{i,j} = \frac{freq_{i,j}}{max_l freq_{l,j}} \tag{3}$$

donde el máximo se calcula sobre todos los términos del documento d_j. Si el término t_i no aparece en el documento d_j entonces $tf_{i,j} = 0$.

Sea N la cantidad total de documentos en el sistema y n_i la cantidad de documentos en los que aparece el término t_i. La frecuencia de ocurrencia de un término t_i dentro de todos los documentos de la colección idf_i está dada por:

$$idf_i = \log \frac{N}{n_i} \tag{4}$$

El peso del término t_i en el documento d_j está dado por:

$$w_{i,j} = tf_{i,j} \cdot idf_i \tag{5}$$

El cálculo de los pesos en la consulta q se hace de la siguiente forma:

$$w_{i,q} = \begin{cases} 0, \ si \ freq_{i,q} = 0 \\ \left(a + (1-a)\frac{freq_{i,q}}{max_l freq_{l,q}}\right) \cdot \log \frac{N}{n_i}, \ en \ otro \ caso \end{cases} \tag{6}$$

donde $freq_{i,q}$ es la frecuencia del término t_i en el texto de la consulta q. El término a es de suavizado y permite amortiguar la contribución de la frecuencia del término, toma un valor entre 0 y 1. Los valores más usados son 0.4 y 0.5.

2.2. ¿Por qué seleccionamos el modelo vectorial?

Se seleccionó el modelo vectorial primeramente por la amplia cantidad de elementos impartidos durante el curso sobre este modelo. Además este presenta las siguientes ventajas:

- El esquema de ponderación $tf - idf$ para los documentos mejora el rendimiento de la recuperación.
- La estrategia de coincidencia parcial permite la recuperación de documentos que se aproximen a los requerimientos de la consulta.
- La fórmula del coseno ordena los documentos de acuerdo al grado de similitud.

Además de estas ventajas también cabe destacar la muy buena posibilidad de retroalimentación que admite este modelo.

2.3. Ideas interesantes

Usar diccionarios para representar a los vectores y las consultas Una idea interesante que utilizamos para representar lo que son los vectores en la teoría, como los vectores de los términos y de pesos tanto de los documentos como de las consultas, en lugar de como vectores, los implementamos como diccionarios, tal que la clave es el término i (preprocesado) y el valor correspondiente a dicha clave según cual sea el diccionario, sería la frecuencia o el peso del término en un documento o consulta en específico.

Esto lo hacemos debido a la gran cantidad de términos que pudieran haber en una colección grande de documentos y representar cada documento mediante un vector de longitud igual a la cantidad total de términos distintos sería altamente costoso en memoria y en tiempo de ejecución. Además de la gran probabilidad de que la matriz conformada por los vectores de frecuencia de los términos y en consecuencia la matriz formada por los vectores de pesos de los términos en los documentos sean muy esparcidas, pues lo más probable, si se tiene una colección grande de documentos, es que un término t_i, si es relevante, aparezca en una cantidad muy inferior de documentos con respecto al total de los mismos.

Además esto lo podemos hacer debido a que un término que no aparezca en un documento, dado que su frecuencia es cero y por como se calculan los pesos y la similitud entre un documento y una consulta, no afecta para nada el cálculo de estos parámetros. Veamos esto rápidamente:

Por (2) tenemos en el numerador una sumatoria donde el término i de la sumatoria es 0 si $w_{i,j} = 0 \vee w_{i,q} = 0$. Por (5) tenemos que $w_{i,j} = 0$ si $tf_{i,j} = 0 \vee idf_i = 0$. Luego por (3) tenemos que $tf_{i,j} = 0$ si $freq_{i,j} = 0$, o sea si el término t_i no aparece en el documento d_j

Para la consulta, por (6) tenemos que si la $freq_{i,q} = 0$ entonces $w_{i,q} = 0$.

Por tanto si el término t_i no aparece en el documento o no aparece en la consulta, entonces t_i no influye en la sumatoria del numerador.

Pasemos entonces a analizar el denominador. En este tenemos dos sumatorias: una que itera por los cuadrados de los pesos del vector del documento y otra que itera por los cuadrados de los pesos del vector de la consulta. Es evidente que si el peso de t_i en d_j es 0 entonces este no influye en la sumatoria. Lo mismo ocurre si un término no aparece en la consulta q.

Por tanto llegamos a la conclusión que para calcular la similitud entre una consulta y un documento solo necesitamos los términos que aparecen en el documento o en la consulta.

Guardar volmenes de información en .json, sobre todo aquellas que se utilizan mucho De esta forma se devidía la ejecución en partes, en vez de hacerlo todo de una vez lo cual se demoraría un tiempo considerable y además cada vez que se quisiera correr el sistema se estarían haciendo los mismos cálculos. Algunos objetos guardados en .json son la representación de los vectores de los documentos y las consultas y un diccionario que contiene a cada término con su respectivo idf_i. Los adjuntados en este trabajo son los referentes a los de la colección Medline.

Utilizar un heap para llevar el ranking de similitud de una consulta Utilizando esta estructura llevamos un orden parcial de las similitudes de una consulta con los documentos en el sistema.

3. Implementación del sistema

3.1. Preprocesamiento de los documentos

Las funciones para el preprocesamiento de los documentos de una colección las podemos encontrar en "collection_preprocess.py". Primero tenemos la función `collection_preprocessing` la cual, a partir de la colección de documentos, devuelve un diccionario tal que las llaves son los términos que aparecen en los documentos y el valor asociado a cada término es el conjunto de los documentos en los que aparece dicho término. Los términos a su vez son sometidos a un preprocesamiento y para esto utilizamos la librería `nltk`.

El proceso de tokenización se hace mediante la función `word_tokenize` la cual a partir de un texto(por defecto en inglés) devuelve los diferentes tokens de dicho texto. Luego los tokens son clasificados sintácticamente y etiquetados con dicha clasificación mediante el método `pos_tag`. Esto se hace pues para el proceso de "*lemmatizing*"(llevar las palabras a su raíz gramatical) que es el proceso que viene a continuación, tener los tokens clasificados mejora el rendimiento de este proceso. Después de realizado el proceso de *lemmatizing*, se chequea si los términos obtenidos son "*stopwords*"(palabras que no proveen información útil). Si un término i no es una *stopword*, si no está en el diccionario, se añade

como llave y se crea un conjunto con el documento actual. Si ya está el término en el diccionario entonces se añade al conjunto correspondiente al término el documento actual. Como es un conjunto si el documento ya está en el conjunto, este no se agregará.

Debido a que se analizan todos los documentos de una colección y de cada uno de estos se analizan todos sus términos, asumiendo que todas las operaciones que se realizan sobre un token se hacen en $O(1)$, llegamos a la conclusión que una llamada a esta función tiene una complejidad temporal $O(n \cdot m)$; siendo n la cantidad total de documentos y m la cantidad de tokens del documento que mayor cantidad de tokens tiene.

La otra función que aparece en este archivo es `terms_freq_doc`, la cual devuelve de forma perezosa un diccionario para cada documento de la colección y un valor entero. El diccionario tiene como llaves los términos que aparecen en dicho documento y el valor asociado a cada término es la frecuencia del término en el documento. El número entero que se devuelve junto con el diccionario es la frecuencia del término de mayor frecuencia en el documento. Cada vez que se detecte un documento nuevo, se crea un diccionario y un entero inicializado con 0. El procesamiento de los términos se hace de forma similar que en el método anterior. Por cada término i en el documento j se verifica si ya este fue agregado al diccionario. En caso afirmativo se incrementa en 1 su valor asociado y en caso contrario se agrega el término al diccionario, asociándole 1 como valor, pues es la primera vez que se detecta. Después de esto se comprueba si dado este aumento la frecuencia del término aumentó, de tal forma que se hizo mayor que la máxima frecuencia detectada hasta el momento para el documento. De ocurrir lo antes planteado se actualiza entonces la máxima frecuencia detectada (la variable `max_freq`). Cada vez que se termine de analizar un documento se devuelve el diccionario y el entero antes mencionado.

Como se explicó una ejecución completa de este método analiza cada uno de los documentos y sus tokens. Además realiza una gran cantidad de operaciones similares a las que se hacen en la función anterior y las que tiene diferente con respecto al otro tienen una complejidad temporal despreciable (son $O(1)$). Por tanto llegamos a la conclusión que una ejecución completa de este método es $O(n \cdot m)$, siendo n y m los mismos valores mencionados anteriormente.

3.2. Representación de los documentos

El código relacionado a este apartado lo podemos encontrar en "docs_representation.py". Primero veamos como hallar los valores idf_i para los diferentes términos. Estos valores se calculan utilizando la función `calculate_idfs` la cual recibe un diccionario que tiene como llaves a los diferentes términos y el objeto asociado a un término t_i es el conjunto de documentos en los que el término t_i aparece; además de un valor entero que indica la cantidad de documentos que hay en el sistema. Más concretamente este método recibe la salida del método `collection_preprocessing` u otro que de una salida similar. Esté método devuelve un diccionario que tiene como llaves a cada uno de los térmi-

nos, y el valor asociado a cada término t_i es su respectivo valor idf_i. Lo que se hace dentro del método es lo siguiente:

- Crear el diccionario `idfs`.
- Por cada término en el diccionario de entrada:
  - Calcular su idf_i correspondiente y guardar la pareja t_i y idf_i en el diccionario `idfs` como llave y valor respectivamente.
- Retornar el diccionario `idfs`.

Es sencillo notar que la complejidad temporal de esta función es $O(k)$, siendo k el número total de términos.

Veamos ahora como calcular los valores $tf_{i,j}$ para cada término en un documento. La función utilizada para esto es `calculate_tfijs` que recibe un objeto iterable, tal que cada elemento del objeto iterable es una tupla <diccionario, entero>. El diccionario tiene como llaves los términos que aparecen en un documento d_j y el entero indica la frecuencia del término de mayor frecuencia en el documento d_j. Este método retorna de forma perezosa un diccionario por cada documento d_j que contiene como llave los términos de dicho documento y para cada término su valor asociado es el valor $tf_{i,j}$ correspondiente. La forma de hacer esto es:

- Por cada elemento del objeto iterable:
  - Crear el diccionario `tfj`
  - Por cada término en el diccionario obtenido del objeto iterable:
    - Calcular el $tf_{i,j}$ correspondiente y guardar dicha pareja en el diccionario `tfj`
  - Retornar `tfj`

La idea de hacerlo de forma perezosa es no tener en memoria grandes volúmenes de información que no son necesarios en todo momento. Ya que se analizan los n documentos de la colección y en cada una la cantidad de términos es $O(m)$, una ejecución completa de este método es $O(n \cdot m)$.

Como ya vimos como calcular idf y tf a continuación se explicará como calcular entonces los "vectores" de pesos para cada uno de los documentos. Esto se hace mediante la función `calculate_weights`. Esta función recibe, como primer argumento, el mismo argumento que recibe la función `calculate_tfijs` descrita anteriormente, y como segundo argumento, un diccionario que contiene los términos y sus respectivos idf_i. Su funcionamiento es el siguiente:

- Crear una lista `vec_docs`, que será en la que se irán guardando los "vectores" de pesos
- Por cada elemento que devuelve el llamado a la fúncion `calculate_tfijs` (recordemos que cada elemento devuelto es un diccionario con los términos de un documento d_j y sus respectivos $tf_{i,j}$):
  - Crear el diccionario `vec_weights`
  - Por cada uno de los términos del diccionario devuelto:

    - Calcular su peso y guardar la pareja término y peso en el diccionario `vec_weights`
  - Añadir `vec_weights` a la lista `vec_docs`
- Retornar `vec_docs`.

En este método se hace un llamado a `calculate_tfijs` la cual sabemos que tiene complejidad temporal $O(n \cdot m)$ y se itera por los n elementos que esta devuelve y por cada uno se recorre una cantidad $O(m)$ de términos. Por tanto la complejidad temporal de esta función es $O(n \cdot m)$.

Ejecutando este archivo se calculan y se guardan en archivos .json los vectores de pesos de los documentos y los valores idf_i de los términos. Para especificar la colección deseada se debe indicar la dirección en la variable `collection` de la línea 36.

3.3. Preprocesamiento de las consultas

El preprocesamiento de las consultas se hace de una forma parecida al preprocesamiento de los documentos. El código referente a este apartado se encuentra en el archivo "queries_preprocess".

La primera función "query_preprocessing" se utiliza para preprocesar una sola consulta. Recibe el texto de una consulta y el diccionario `idfs` que contiene a los términos con sus respectivos valores de idf. Una consulta se procesa de la misma manera que un documento, lo que con un paso extra: luego de que un token haya sido completamente procesado se comprueba si pertenece al diccionario `idfs`, pues solo interesan las palabras que pertenezcan al menos a un documento de la colección. Este método devuelve un diccionario con los términos de la consulta y sus respectivas frecuencias, además de un entero que indica la frecuencia máxima de un término en la consulta. Su complejidad temporal es $O(m)$, siendo m la cantidad de palabras de la consulta.

El siguiente método "queries_preprocessing" se utiliza para el preprocesamiento de las consultas de la colección. Devuelve de forma perezosa el diccionario y el entero explicados anteriormente para cada una de las consultas en la colección. Su complejidad temporal es $O(n \cdot m)$ siendo n el número de consultas y m la cantidad de tokens de la consulta con mayor cantidad de tokens.

3.4. Representación de las consultas

El código para calcular los pesos de las consultas se encuentra en el archivo "queries_representation.py". En este se haya la función "calculate_weigths_queries" que recibe un valor para a, el diccionario `idfs` y un objeto iterable, donde cada elemento de este es una tupla <diccionario,entero>, tal que el diccionario contiene los términos de una consulta y su frecuencia en la misma y el entero indica la frecuancia máxima de un término en la consulta. Esta devuelve una lista con la representación en pesos de las consultas pasadas en el objeto iterable. La definición de la función es muy similar a la utilizada para calcular los pesos de los documentos; el único cambio es el de utilizar además la constante a de

suavizado para el cálculo de los pesos de una consulta. La complejidad temporal es $O(n \cdot m)$.

Ejecutando este archivo se guarda en un archivo .json los pesos de las consultas de la colección. Para especificar la colección deseada se debe indicar la dirección en la variable `collection` de la línea 25.

3.5. Similitud entre documentos y consultas

El código relacionado a este apartado se encuentra en el archivo "similarity.py". La primera función que se aprecia es `sim_doc_query`. Esta recibe los "vectores" de pesos de un documento y una consulta (que recordemos son diccionarios) y devuelve un número que representa la similitud entre el documento y la consulta. La forma de proceder es la siguiente:

- Si el documento o la consulta no tiene términos entonces la similitud es 0.
- Se calcula $\overrightarrow{d_j} \cdot \overrightarrow{q}$ utilizando solo los términos que tienen en común ambos
- Se calcula $|\overrightarrow{d_j}|$ con todos los términos de d_j
- Se calcula $|\overrightarrow{q}|$ con todos los términos de q
- Se retorna $\frac{\overrightarrow{d_j} \cdot \overrightarrow{q}}{|\overrightarrow{d_j}| \cdot |\overrightarrow{q}|}$

De forma evidente se aprecia que la complejidad temporal de esta función es $O(a + b)$, siendo a la cantidad de términos del documento y b la cantidad de términos de la consulta.

La próxima función es `sim_docs_query` la cual recibe una lista con los vectores de pesos de una colección de documentos, un vector de pesos de una consulta y un flotante `m` que indica la similitud mínima admisible entre documento y consulta. Esta función devuelve una lista de los documentos(o mejor dicho los id de dichos documentos) más similares a la consulta, ordenados de mayor a menor similitud. Su funcionamiento es el siguiente:

- Crear una lista `heap` que utilizaremos como heap, en el cual los documentos estarán ordenados parcialmente de menor a mayor similitud.
- Por cada uno de los documentos:
  - Calcular la similitud entre el documento y la consulta
  - Si la similitud es mayor o igual que `m` empujar el documento en el heap:
- Crear la lista `result` con cardinalidad igual a la cantidad de elementos en el heap
- Mientras el heap no este vacío:
  - Extraer el primer elemento del heap y colocar el documento en la posición más a la derecha de `result` a la que aún no se le ha asignado un documento.
- Retornar la lista `result`

Obsérvese que dada la implementación se garantiza que los documentos más similares a la consulta se guarden en el heap. Y que la lista `result` es un ranking de los documentos del heap ordenados de mayor a menor similitud con

la consulta. Las operaciones sobre el heap son $O(\log n)$, siendo n el número de documentos; y el cálculo de similitud ya vimos que es $O(a+b)$. Por tanto la complejidad temporal de guardar los documentos en el heap es $O(n \cdot (a+b+\log n))$. Pasar los documentos de la lista al heap tiene complejidad $O(n \log n)$. Por tanto la complejidad temporal de esta función es $O(n \cdot (a+b+\log n))$.

La última función `sim_docs_queries` recibe una lista de vectores de pesos de documentos, una lista de vectores de pesos de consultas y una lista de números fraccionarios tal que el valor en la posición i de esta última indica la similitud mínima admisible para la consulta i. Devuelve una lista donde cada elemento i de dicha lista es una lista con los documentos recuperados para la consulta i. Está función hace un llamado a la función `sim_docs_query` por cada una de las consultas, por tanto siendo k el número de consultas llegamos a la conclusión que el método tiene una complejidad temporal $O(k \cdot n \cdot (a+b+\log n))$.

3.6. Retroalimentacin

Para mejorar el sistema de forma que se pueda ir refinando la respuesta a determinada consulta se implementó la retroalimentación. A continuación se explicará cómo se realizó.

Algoritmo de Rocchio La idea que busca este algoritmo es que a partir de los documentos relevantes y no relevantes a una consulta se pueda obtener un nuevo vector que maximice la diferencia entre los centroides de ambos conjuntos de documentos. Para esto se aplica la siguiente fórmula:

$$\boldsymbol{q_m} = \alpha \boldsymbol{q_0} + \frac{\beta}{|D_r|} \sum_{\boldsymbol{d_j} \in \boldsymbol{D_r}} \boldsymbol{d_j} - \frac{\gamma}{|D_{nr}|} \sum_{\boldsymbol{d_j} \in \boldsymbol{D_{nr}}} \boldsymbol{d_j}$$

Donde D_r y D_{nr} son los conjuntos conocidos de documentos relevantes y no relevantes respectivamente.

q_0 es la consulta dada

α, β y γ son los pesos establecidos para cada término de consulta, comúnmente se usa 1, 0.75 y 0.15 respectivamente.

Para la implementación del mismo se realizó un sumador de dos vectores y un multiplicador entre vector y escalar. El primero recibe dos vectores(diccionarios) de pesos y recorre los términos del segundo vector. En caso de que el término i del vector 2 esté en el vector 1 se suman sus pesos correspondientes y se actualiza el peso en el vector 1; de lo contrario se agrega el término i del vector 2 con su peso correspondiente al vector 1. Con este procedimiento la suma de ambos vectores queda guardada en el vector 1. De esta forma se hicieron dos sumatorias: una con los vectores del conjunto de vectores de documentos relevantes y otra con los del conjunto de vectores de documentos no relevantes y los vectores resultantes se multiplicaron con $\frac{\beta}{|D_r|}$ y $-\frac{\gamma}{|D_{nr}|}$ respectivamente. Se multiplicó además el vector de la consulta inicial por α y se devolvió como resultado la suma de estos tres vectores. El código referente a esto se puede encontrar en el archivo "Rocchio.py".

3.7. Interfaz gráfica

Como la interfaz gráfica no es objetivo de la asignatura solo vamos a explicar el funcionamiento de la misma, solo decir que para su implementación se utilizó la librería `tkinter`. El código referente a la misma se encuentra en el archivo "user.py" y para acceder a esta se ejecuta dicho archivo. Cuando se ejecute es necesario comprobar que se cargue la última colección procesada. Esto se hace especificando la dirección en la línea 13 de este archivo.

Al iniciar la aplicación, esta luce como muestra la Fig. 1:

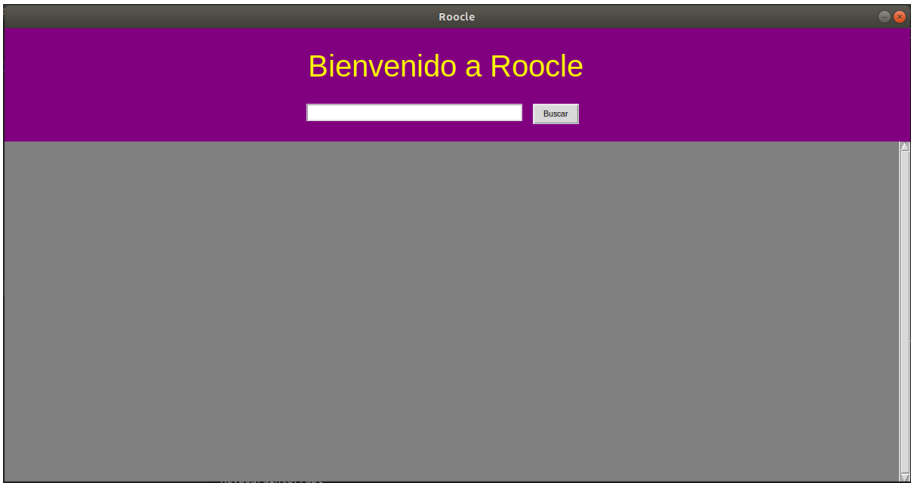

Figura 1. Inicio de la aplicación

Para realizar una consulta la escribimos en la barra que aparece debajo del cartel de bienvenida y presionamos el botón "Buscar". Al hacer esto la aplicación mostrará los 10 documentos más similares a la consulta realizada. Por ejemplo veamos, en la Fig 2, que se obtiene al realizar la primera consulta de la colección **Cranfield**.

En la Fig. 2 vemos el documento más similar a la consulta, para ver el resto de documentos utilizamos la *scrollbar* situada a la derecha.

Ahora obsérvese que a la derecha del documento aparece la pregunta de si ¿Ha sido útil el documento? Y a la derecha de esta dos botones: "Sí" y "No". El usuario al presionar uno de estos botones indica si el documento es relevante o no. Esta información luego se utiliza para el proceso de retroalimentación.

Por último se observa también el botón "Reconsultar" que este solo aparece una vez (a la derecha del documento más similar) y al presionar en él se indica que se rehaga la consulta con la información de relevancia e irrelevancia de los documentos. O sea, se vuelve a calcular la similitud utilizando como vector consulta el devuelto por el Algoritmo de Rocchio.

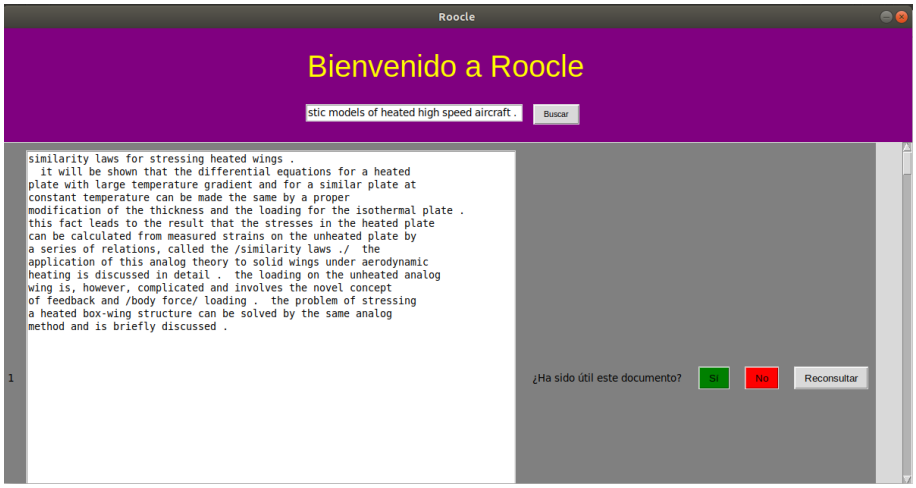

Figura 2. Ejemplo de consulta (Cranfield)

En la Fig. 3 se presenta un segundo ejemplo utilizando la colección **Medline**, realizando la primera consulta de esta colección.

4. Análisis de los resultados del sistema

4.1. Explicación de las Medidas

Para el análisis de los resultados del sistema se implementaron un conjunto de medidas que caracterizan el sistema de acuerdo a los documentos que recupera y los que no. En esto se destacan siete conjuntos fundamentales de documentos:

- Conjunto de documentos relevantes (REL)
- Conjunto de documentos irrelevantes (I)
- Conjunto de documentos recuperados (REC)
- Conjunto de documentos recuperados relevantes (RR)
- Conjunto de documentos recuperados no relevantes (RI)
- Conjunto de documentos no recuperados relevantes (NR)
- Conjunto de documentos no recuperados irrelevantes (NN)

Por la propia definicón de estos conocemos que se cumple la relación de la Fig 4.

A continuación se enuncian las medidas implementadas y se explica la idea utilizada.

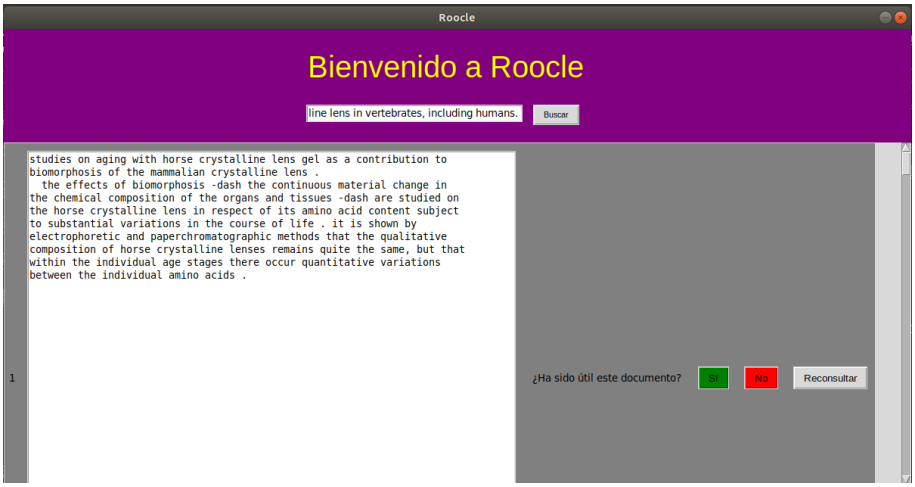

Figura 3. Ejemplo de consulta (Medline)

Precisión Esta medida caracteriza la respuesta a la consulta según los documentos relevantes que se recuperaron

$$P = \frac{|RR|}{|REC|}$$

Esta medida tiende a decrecer cuando aumentan los documentos recuperados. Para calcularla, una vez hallados los documentos relevantes para la consulta, utilizando la información de apoyo de la colección, se analizó cuáles se habían recuperado y de estos los que estaban entre los relevantes y se aplicó la fórmula. Esto se hizo para cada consulta de la colección y los resultados obtenidos se promediaron.

Recobrado Esta medida es fundamental en los procesos de recuperación de información.

$$R = \frac{|RR|}{|REL|}$$

Para calcularla, una vez obtenidos los documentos relevantes para la consulta se analizó de ellos cuántos se recuperaron, para aplicar la fórmula. De esta forma se procedió en cada consulta de la colección y se promediaron los resultados.

Medida F Esta medida permite enfatizar la precisión sobre el recobrado o viceversa:

$$F = \frac{(1+\beta^2)PR}{\beta^2 P + R}$$

Figura 4. Relación entre los conjuntos de documentos

$\beta < 1$ el recobrado tiene mayor peso
$\beta = 1$ la precisión y el recobrado tienen igual peso
$\beta > 1$ la precisión tiene mayor peso
Para hallar esta medida se utilizó el cálculo de las dos anteriores y se promedió para cada consulta de la colección.

Medida F1 Esta medida armoniza precisión y recobrado teniendo en cuenta ambos.

$$F1 = \frac{2PR}{P + R}$$

F1 tendrá un valor alto si la precisión y el recobrado son altos, luego puede interpretarse como un esfuerzo por hallar el mejor compromiso entre ambos. Para el cálculo de esta se tomó de apoyo los cálculos de precisión y recobrado anteriormente explicados y se promedió para cada consulta de la colección.

R-Precisión Esta medida es el ranking de documentos relevantes a una consulta para la cul existen R documentos relevantes.

$$P_R = \frac{|RR|_R}{R}$$

Para esta medida se seleccionaron los R primeros documentos recuperados y de estos se analizó cuáles eran relevantes para la consulta. Esto se hizo para cada consulta de la colección y se promediaron los resultados obtenidos.

Proporción de fallo Tiene en cuenta la cantidad de documentos irrelevantes

$$Fallout = \frac{|RI|}{|I|}$$

Para su cálculo, se tomaron los documentos relevantes para la consulta y los documentos recuperados. Con la diferencia entre el conjunto de todos los documentos y el conjunto de todos los documentos relevantes para determinada consulta se obtuvieron los documentos irrelevantes para la misma. Luego se compararon de estos cuáles se recuperaron y se aplicó la fórmula. Se procedió de la misma forma para cada consulta y se promediaron los resultados.

4.2. Análisis de los Resultados

Para cada consulta, utilizando la colección **Cranfield** (que cuenta con 1400 documentos y 225 consultas) y retornando los documentos cuya similitud con la consulta es mayor o igual a 0.1, se obtuvieron los siguientes resultados:

- Precisión promedio: 0.15463199286061183
- Recobrado promedio: 0.6024723852836802
- R-precisión5 promedio: 0.409777777777778
- Medida_F promedio $\beta = 0$: 0.15463199286061183
- Medida_F promedio $\beta = 2$: 0.33108870640070265
- Medida_F1 promedio: 0.22145562556179862
- Proporción de fallo promedio: 0.10862096867414041
- R-fallout5 promedio: 0.012918518518518506

Se observa una baja precisión promedio, sin embargo la R-precisión5 promedio es mucho más alta. El recobrado promedio lo podemos catalogar de aceptable.

Utilizando un umbral de similitud de 0.2 se obtienen los siguientes resultados:

- Precisión promedio: 0.43399651574757653
- Recobrado promedio: 0.34757621509916814
- R-precisión5 promedio: 0.3591111111111116
- Medida_F promedio $\beta = 0$: 0.43399651574757653
- Medida_F promedio $\beta = 2$: 0.32329056742389967
- Medida_F1 promedio: 0.32206114272154335
- Proporción de fallo promedio: 0.021765179813960357
- R-fallout5 promedio: 0.007901234567901236

Se aprecia como la precisión promedio aumentó en gran medida pero sin embargo el recobrado disminuyó bastante. Incluso la R-precisión5 disminuye debido a que para muchas consultas se devuelven menos de 5 documentos.

Por último si se fija el umbral de similitud en 0.15 se obtienen los siguientes resultados:

- Precisión promedio: 0.28814475479798424

- Recobrado promedio: 0.45801026483780116
- R-precisión5 promedio: 0.402666666666667
- Medida_F promedio $\beta = 0$: 0.28814475479798424
- Medida_F promedio $\beta = 2$: 0.3532712231591747
- Medida_F1 promedio: 0.29718448265513353
- Proporción de fallo promedio: 0.044796004410715566
- R-fallout5 promedio: 0.011377777777777777

Para este umbral de similitud se aprecia una compensación en cuanto a los valores de precisión y recobrado pero aún así los valores obtenidos no son los mejores.

Si nos fijamos en la Medida_F1 se puede apreciar que el mejor resultado se obtiene cuando el umbral de similitud es 0.2.

Pasemos ahora analizar el sistema con la colección **Medline** (que cuenta con 1033 documentos y 30 consultas) y con umbral de similitud 0.1:

- Precisión promedio: 0.6101614388092593
- Recobrado promedio: 0.4688505534796139
- R-precisión5 promedio: 0.6933333333333335
- Medida_F promedio $\beta = 0$: 0.6101614388092593
- Medida_F promedio $\beta = 2$: 0.4608114878487888
- Medida_F1 promedio: 0.47236203447374764
- Proporción de fallo promedio: 0.16790582403965298
- R-fallout5 promedio: 0.04444444444444446

Como se puede apreciar con este umbral de similitud la precisión promedio es aceptable y el recobrado se acerca al 50 %. La R-precisión5 promedio es bastante buena.

Utilizando ahora un umbral de similitud de 0.05 se obtienen los siguientes resultados:

- Precisión promedio: 0.3530530523152345
- Recobrado promedio: 0.7007911295917294
- R-precisión5 promedio: 0.7200000000000002
- Medida_F promedio $\beta = 0$: 0.3530530523152345
- Medida_F promedio $\beta = 2$: 0.5464053607927845
- Medida_F1 promedio: 0.4373855559506214
- Proporción de fallo promedio: 0.6087398373983741
- R-fallout5 promedio: 0.04666666666666668

Se aprecia como la precisión promedio disminuyó bastante, contrario al recobrado que aumentó en gran medida su valor. La R-precisión5 aumentá un poco su valor debido a que al bajar el umbral de similitud, consultas para las que se recuperaban menos de 5 documentos, ahora se devuelven 5 o más. Al bajar también el umbral de similitud también aumentó en gran medida la proporción de fallo promedio, aunque el R-fallout5 no varió mucho su valor.

Por último veamos ahora los resultados obtenidos utilizando 0.08 como umbral de similitud:

- Precisión promedio: 0.5377846857131453
- Recobrado promedio: 0.5580216992563317
- R-precisión5 promedio: 0.7200000000000002
- Medida_F promedio $\beta = 0$: 0.5377846857131453
- Medida_F promedio $\beta = 2$: 0.5205735784074199
- Medida_F1 promedio: 0.4979325593246483
- Proporcin de fallo promedio: 0.25763239875389343
- R-fallout5 promedio: 0.04555555555555557

Se aprecia una la existencia de una compensación entre la precisión y el recobrado. La R-precisión5 es bastante buena.

Comparando las Medidas_F1 con los diferentes valores de umbrales de similitud el mejor resultado se obtiene utilizando 0.08.

El código correspondiente a estos análisis se encuentra en el archivo "system_evaluation.py", al ejecutarlo es necesario especificar, en la línea 16, la dirección del archivo que contiene las relaciones de relevancia de la última colección procesada. Los valores de umbral de similitud se fijan en la línea 14.

5. Análisis de la aplicación

5.1. Ventajas

- Utiliza el modelo vectorial por lo que los documentos se ordenan de acuerdo al grado de similitud con la consulta, puediéndose recuperar documentos que se aproximen a los requerimientos de la consulta.
- Presenta una interfaz gráfica, de forma que sea más accesible y cómoda para los usuarios.
- Utiliza la retroalimentación para mejorar las respuestas a las consultas de acuerdo al criterio del usuario.
- Tiene una baja proporción de fallo.

5.2. Desventajas

- Utiliza un modelo vectorial por lo que asume que todos los términos indexados son mutuamente independientes.
- No se integró con algoritmos de Crawling.
- No se usa bases de conocimiento como tesauros u ontologías.
- Puede ser difícil lograr una buena compensación entre la precisión y el recobrado.

6. Conclusiones

En función de lo planteado en las secciones anteriores se arriban a las siguientes conclusiones:

- Se diseñó un Sistema de Recuperación de Información para una colección de documentos dada.
- Se realizó una interfaz gráfica para brindar comodidad al usuario.
- Se implementó la retroalimentación para el sistema de forma que mediante la ayuda del usuario se perfeccione su funcionamiento.
- Se analizaron los resultados obtenidos por el sistema con vistas a evaluar el funcionamiento del mismo.

7. Recomendaciones

En virtud del trabajo realizado y las posibilidades del mismo se recomienda:

- Ampliar el proyecto mediante la integración con algoritmos de Crawling y añadiendo bases de conocimiento como ontologías o tesauros.
- Utilizar técnicas más avanzadas de recuperación de información para obtener mejores resultados, sobre todo en la precisión y el recobrado.
- Utilizar técnicas más avanzdas para el preprocesamiento de de documentos y consultas lo cual conllevaría a la obtención de mejores resultados.

Referencias

1. Conferencias de Sistemas de Recuperación de Información. Curso 2021-2022.
2. Canal de yotube: sentdex
3. Documentación de python sobre heap