Búsqueda y Minería de Información

P1: Implementación de un motor de búsqueda

Javier Alejandro Lougedo Lorente javier.lougedo@estudiante.uam.es
Alba Ramos Pedroviejo alba.ramosp@estudiante.uam.es

Búsqueda y Minería de Información Group 2461
Profesor: Alejandro Bellogín Kouki
Grado de Ingeniería Informática - Universidad Autónoma de Madrid
1 de marzo de 2021, Madrid, España

Resumen

En la presente práctica se lleva a cabo la implementación de un motor de búsqueda a partir de la librería Whoosh, con ayuda de las librerías de BeatifulSoup4, para el parseado de documentos HTML, y de la librería ZipFile, para la gestión de archivos y colecciones en formato zip. El resto de componentes pertenecen al Python Básico, a excepción del ejercicio tres que hace uso de la librería MatPlotLib para llevar a cabo el dibujado de las gráficas. Su instalación y uso se encuentran comentados más en detalle en el fichero *readme_lib.txt*.

Índice

En las secciones a continuación se van a comentar diversos aspectos relevantes de la implementación y resolución de cada ejercicio.

1. Ejercicio 1: Implementación basada en Whoosh

Se pide implementar las clases y módulos necesarios para que el programa main funcione. Realizamos esto basándonos en gran medida en el documento de WhoosyExample.py, así como en el output, cuando teníamos dudas de qué debía devolver cada función.

Definimos así Index, WhooshIndex (que implementa todas las funciones de Index, haciendo así Index las veces de clase abstracta, en cierta forma), y WhooshBuilder.

En nuestro caso, guardamos como se pide los dos campos de ruta y contenido en el esquema del documento. Valoramos inicialmente la posibilidad de añadir un nuevo campo título, al cual dar un mayor peso a la hora de buscar y diferenciar particularmente, pero esto daba conflictos con lo que se nos solicitaba en la práctica y haría cambiar las ponderaciones, con lo que finalmente decidimos prescindir de este campo.

Implementamos posteriormente la clase WhooshSearcher sin mucho problema en este apartado, logrando finalmente obtener el resultado que se puede observar en el fichero *output_ example.txt*. En él, obtenemos los resultados de la misma manera que en el ejemplo que se nos ofrece en clase, solo que con una ejecución más rápida (fruto, posiblemente, de ejecutarlo en un ordenador más rápido y disponible, y no de que seamos necesariamente más eficientes que el código propuesto).

Tras esto, nos lanzamos a la implementación del ejercicio dos con los dos modelos de ranking propios, siguiendo la línea propuesta (y que ya se ha podido ver en el *output_ example.txt* que coincide con la propuesta).

2. Ejercicio 2: Modelos vectoriales y ránking de búsqueda

Para implementar el modelo vectorial se han seguido las fórmulas vistas en clase que se muestran en la figura 1. El producto *tf-idf* se ha implementado solamente para aquellos términos que aparecen en la consulta, y realizando el sumatorio tendríamos el numerador de la fórmula del coseno, que se corresponde con el producto escalar solicitado en el apartado 2.1. Más concretamente, para calcular *idf* basta con mirar la frecuencia del término mediante la fórmula *index.doc_freq(t)*, y para calcular *tf* basta con mirar el vector de términos del documento mediante el método *index.term_freq(term, doc_id)*.

Para construir el ránking de producto escalar, simplemente se utiliza el *tf-idf* mencionado anteriormente y se va elaborando el sumatorio del numerador de la fórmula del coseno, evitando devolver en el ránking aquellos que resulten en 0.0 y ordenando de mayor a menor puntuación.

Para construir el ránking del coseno, es necesario calcular el mismo numerador que en el caso

$$idf(t) = \log \frac{|D| + 1}{|D_t| + 0{,}5}$$

$$tf(t,d) = \begin{cases} 1 + \log_2(frec(t,d)) & \text{si } frec(t,d) > 0 \\ 0 & \text{en cualquier otro caso} \end{cases}$$

$$cos(d,q) = \frac{\sum_t \in_q tf(t,d) idf(t)}{\sqrt{(\sum_t tf(t,d) idf(t))^2}}$$

Figura 1: Fórmulas empleadas.

del producto escalar. Habíamos pensado en guardar el numerador que vamos construyendo como atributo de la clase **VSMDotProductSearcher**, y así al realizar el ránking del coseno tenerlo heredado. Sin embargo, esto nos obligaría a tener que instanciar previamente un **VSMDotProductSearcher**, porque sino no tendríamos ningún numerador almacenado. Por este motivo, y para cumplir con el principio de responsabilidad única de los objetos, hemos decidido recalcular el numerador en esta clase también, de forma que sea posible conocer el ránking por coseno sin necesidad de instanciar uno de producto escalar.

Este numerador se debe dividir por el módulo del documento en cuestión, y calcular eficientemente este módulo es un paso crucial para que el buscador sea escalable. El módulo debe hacerse teniendo en cuenta todos los términos del vocabulario, por tanto calcularemos *idf* de cada uno, y luego para cada documento tendremos que calcular *tf* solamente si el término está contenido en el documento, en caso contrario no sumaremos nada y así también se ahorra en tiempo de ejecución.

Aunque en el enunciado se especifica que extendamos *WhooshBuilder* para calcular los módulos de cada documento, al realizar las pruebas se ha visto que no era lo más eficiente: en el momento de creación del índice no tenemos las funciones asociadas al mismo que se implementaron en el ejercicio 1. Por ello, hemos decidido realizar este procedimiento al momento de instanciar el buscador ya que, si bien se sacrifica algo de tiempo (bastante poco dada nuestra implementación) en la primera búsqueda, para las demás ya no es así y, además, el código resulta más limpio y breve que haciéndolo en la otra clase. Pensando en una implementación a gran escala, bastaría con realizar una primera búsqueda de prueba para generar estos módulos y ya después se podría poner el buscador en producción sin problemas.

Aclarado el punto de dónde implementar la función de cálculo de módulos, hemos decidido almacenar estos módulos todos en un mismo fichero, ordenados por id del documento al que pertenecen. De este modo, evitamos estar abriendo un fichero por cada documento y ahorramos tiempo de ejecución. Ahora bien, una vez almacenados es necesario obtener el módulo de cada documento, y no sería eficiente tener que mirar en el fichero cada vez. Por ello, al crear el buscador hemos ido guardando los módulos en una lista y así podemos acceder a la misma para conocerlos en tiempo de búsqueda de forma eficiente.

Para colocar un documento en la primera posición del ránking dada la búsqueda *"obama family tree"* es necesario que el documento sea el que más relacionado esté con la consulta. Para lograrlo, hemos creado un documento *obama-family-tree.html* que contiene 382 lineas con las palabras *"obama family tree"* cuatro veces repetidas en cada una. Con esto conseguimos que *idf* no varíe mucho, pero *tf* aumenta considerablemente para este documento. Podemos ver el resultado de ambos ránkings en la figura 2.

```
------------------------------
Checking search results
  WhooshSearcher for query 'obama family tree'
21.370715067364742       obama-family-tree.html
20.648804156609533       clueweb09-en0010-79-2218.html
19.460922650790266       clueweb09-en0010-57-32937.html
19.39656461223384        clueweb09-en0001-02-21241.html
18.889038145589705       clueweb09-en0008-45-29117.html

Done ( 0.1445608139038086 seconds )

  VSMDotProductSearcher for query 'obama family tree'
112.88082339113444       obama-family-tree.html
69.8169533495014         clueweb09-enwp01-59-16163.html
69.8169533495014         clueweb09-enwp02-06-15081.html
69.78292310613918        clueweb09-enwp03-00-6901.html
69.78292310613918        clueweb09-enwp03-07-2998.html

Done ( 0.8031532764434814 seconds )

  VSMCosineSearcher for query 'obama family tree'
1.5989234102484478       obama-family-tree.html
0.33956416356537616      clueweb09-en0010-79-2218.html
0.31248229714000164      clueweb09-en0009-30-2768.html
0.30938357202450995      clueweb09-en0009-30-2441.html
0.30886498176810195      clueweb09-en0009-30-2755.html

Done ( 0.8033409118652344 seconds )
```

Figura 2: Ránking tras incluir el nuevo documento HTML.

El documento, como hemos mencionado, se puede encontrar en el zip *docs1k-modified*.

3. Ejercicio 3: Gráficas y frecuencias de términos

En el main.py se ha añadido, comentada, la línea 52: esta ejecuta el método terms_stats para obtener las gráficas que se solicitan en el ejercicio, almacenándolas en una carpeta stats_output de manera automática y sin pop ups, en formato PDF. A continuación se muestran y comentan las gráficas generadas para cada colección.

En estas gráficas, concretamente en la de docs1k, se pueden observar las ideas de las leyes de Heap y de Zipf referentes a los términos y sus frecuencias en grandes colecciones de archivos y datos.

Relativo a la implementación de este ejercicio hay poco. Nos ayudamos de MatPlotLib, así como de las funciones previamente implementadas, con lo que hay poco que comentar. Sobre las gráficas, cabe mencionar que al solicitar al Plotter que ponga ambos ejes en escala logarítmica, elimina toda etiqueta previa que hubiese. De todas formas, comentamos aun así que el eje Y generalmente muestra cantidad (número de apariciones total o número de documentos en los que aparece) y el eje X es el referente a los términos, ordenados de mayor a menor, de forma que más a la izquierda se encuentran los que más veces salen (family en este caso) y a la derecha los que menos frecuentemente aparecen. Hemos tratado de mantener las etiquetas en ese sentido pero resultaba poco práctico ya que requería de más tiempo de ejecución, además de saturar la gráfica en gran medida.

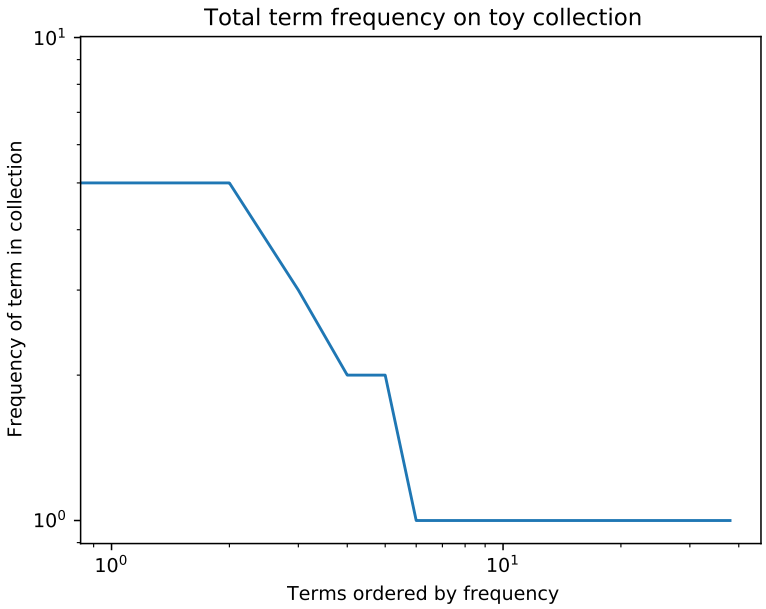

Figura 3: Frecuencia total de los términos colección *toy*

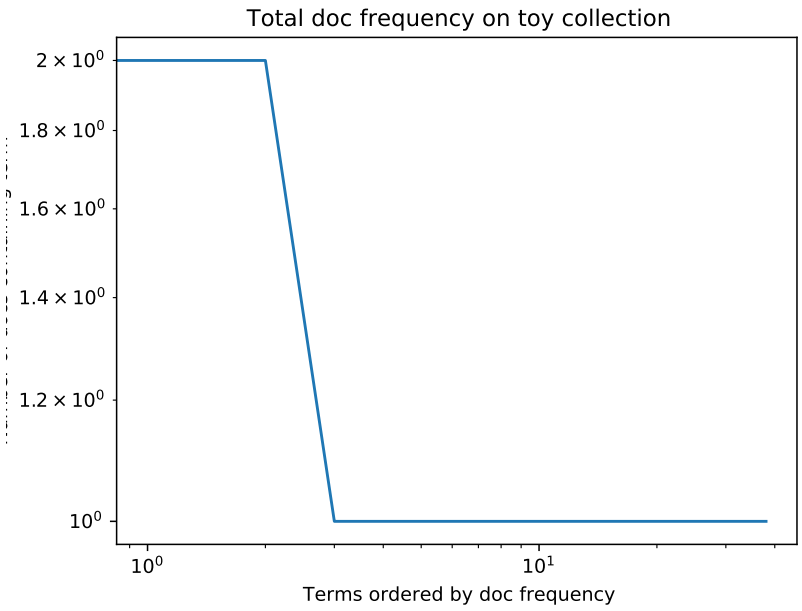

Figura 4: Términos por número de documentos colección *toy*

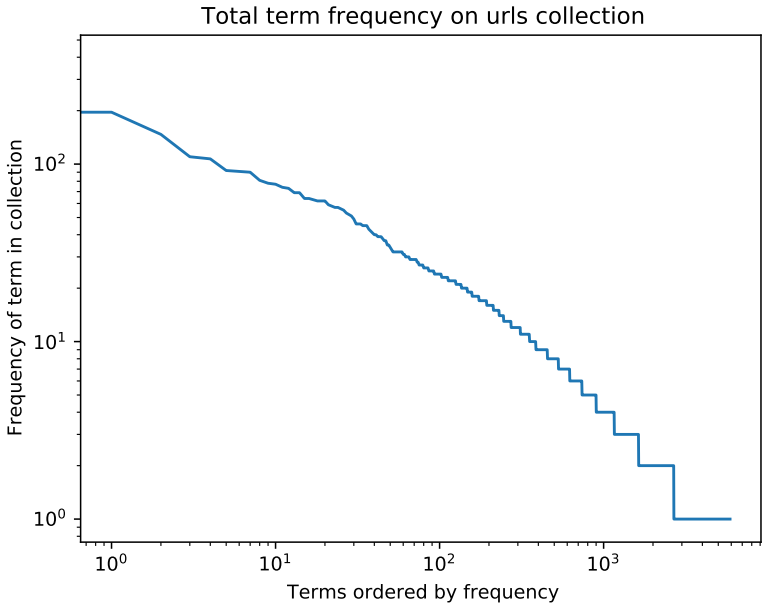

Figura 5: Frecuencia total de los términos colección *urls*

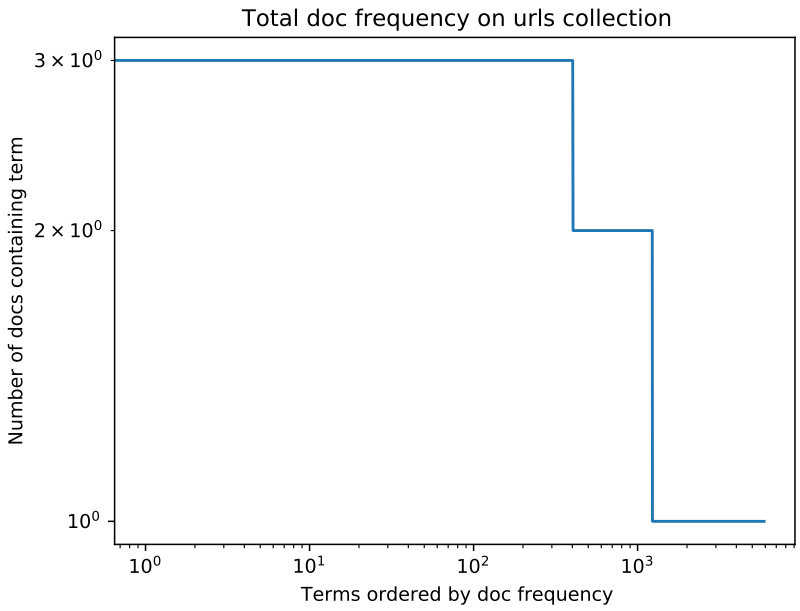

Figura 6: Términos por número de documentos colección *urls*

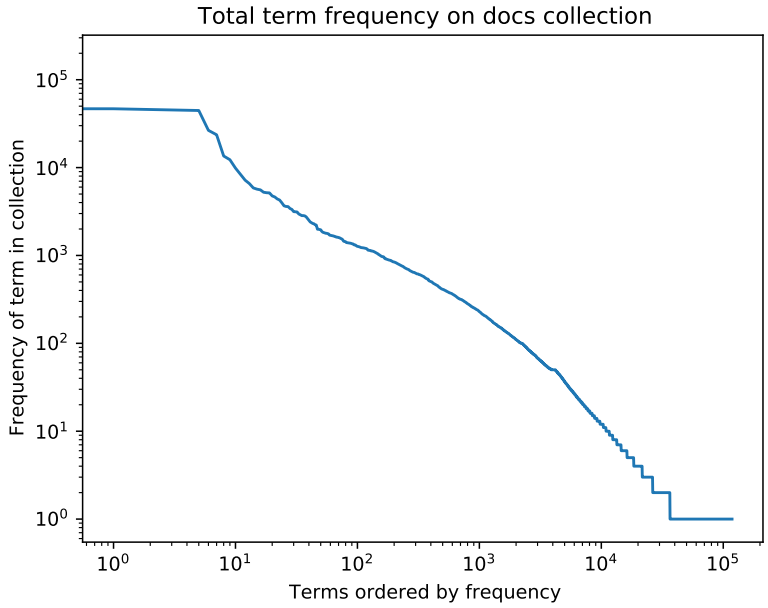

Figura 7: Frecuencia total de los términos colección *docs1k*

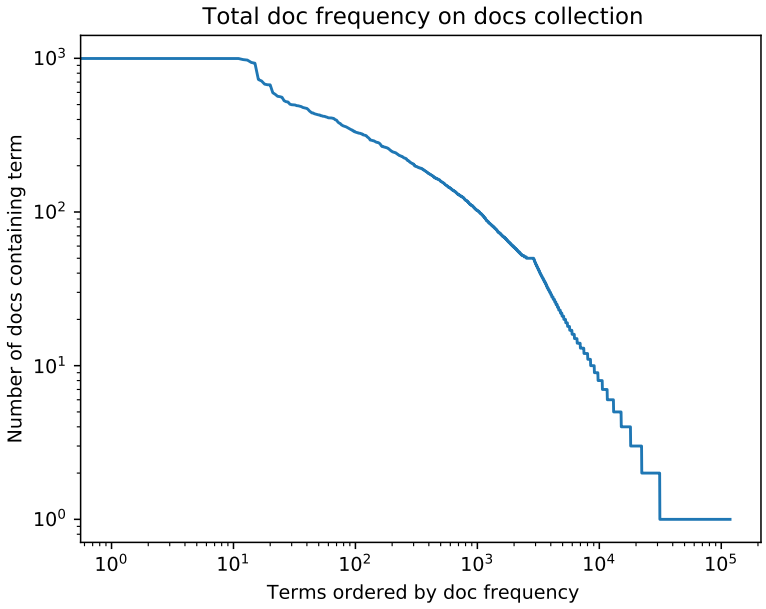

Figura 8: Términos por número de documentos colección *docs1k*

4. Conclusiones

Hemos aprendido como implementar un sencillo buscador con la ayuda de Whoosh y como gestionar los distintos índices y demás campos generados por este y por nosotros. Hemos entendido cual es la dinámica que han de seguir y las dificultades que podemos encontrar en la implementación, así como todos los problemas que implica y como resolverlos.

Hemos visto también la importancia de los índices y de ser ágil y trabajar de manera eficiente, ya que afrontar el problema desde una dirección incorrecta o poco eficiente/efectiva puede dar lugar a que nuestra solución sea válida, pero extremadamente ineficiente, como hemos podido comprobar en nuestras propias carnes durante esta práctica.