Búsqueda y Minería de Información. Práctica 1.

Daniel Santo-Tomás y Juan Velasco

Febrero 2021

1 Introduction

En esta primera práctica implementaremos un motor de búsqueda usando la librería *Whoosh* además de una serie de funciones de rankings sencillas para el motor de búsqueda.

2 Modelo Vectorial. Implementaciones

Se han implementado dos versiones del modelo vectorial. La primera versión se basa únicamente en el producto escalar mientras que la segunda se basa en el ángulo entre dos vectores. Vamos a ver cada una en detalle.

2.1 Producto Escalar

Lo primero que hacemos es crear los vectores que representan tanto a cada documento como a la query de búsqueda. Para ello utilizaremos el modelo de ponderación *tf-idf* donde *tf* mide la importancia de los términos en los documentos mientras que *idf* mide su rareza en los documentos.

Por tanto lo primero que tenemos que hacer es decidir que versión de ambas funciones usaremos en nuestra práctica. Nos hemos decidido por usar las siguientes versiones vistas en teoría:

$$tf(term, doc) = \begin{cases} 1 + ln(freq(term, doc)) \ si \ freq(term, doc) \neq 0 \\ 0 \ si \ freq(term, doc) = 0 \end{cases},$$

$$idf(term) = ln\Big(\frac{|D| + 1}{|D_{term}| + 0.5}\Big),$$

donde $term$ es un término del diccionario, doc es un documento de la colección, $|D|$ es la cantidad de documentos en la colección y D_{term} es la cantidad de documentos de la colección que contienen el término $term$.

Una vez vistas la funciones vamos a mostrar cómo representamos tanto los documentos como la query. Respecto a los documentos su vector consiste en realizar para cada término del diccionario $tf \cdot idf$ e incluirlo en su posición dentro del vector. Sin embargo para la query hemos usado un modelo biario de $tf - idf$

en el cual la entrada del vector correspondiente a su término vale 1 si el término está en la query y 0 si no. Por tanto el score final del documento es el siguiente:

$$score(doc) = \vec{doc} \cdot \vec{q} = \Sigma_{term \in query} tf(term, doc) \cdot idf(term).$$

Esto simplifica mucho la implementación, ya que simplemente iteramos por cada término en la query y calculamos el valor de $tf(term, doc) \cdot idf(term)$ para cada documento y lo sumamos con lo que ya teníamos guardándolo en un diccionario. Finalmente ordenamos los valores obtenidos de mayor a menor siendo los primeros resultados los que tienen más score.

La decisión de usar estas funciones $tf - idf$ ha sido para obtener los valores más próximos al fichero *output.txt* dado, ya que agilizaba la corrección de la implementación mientras lo estábamos implementando, así como evitar $ln(0)$ en idf.

2.2 Ángulo entre vectores

Este segundo modelo es un refinamiento del planteamiento anterior. Consiste en calcular $cos(doc, query)$ mediante la siguiente fórmula:

$$cos(doc, query) = \frac{doc \cdot query}{|doc| \cdot |query|}.$$

Este refinamiento consiste en tener en cuenta el tamaño de los documentos y penalizarlos si son muy largos, ya que el modelo anterior le daba más prioridad a los documentos que son muy largos.

En nuestra implementación hemos usado una variante del coseno para este segundo modelo. Hemos eliminado $|query|$ ya que es siempre igual para todo documento por lo que la diferencia de score entre documentos se mantiene en la misma proporción, haciendo que no se modifique el ranking final.

A nivel código hemos usado para calcular el denominador la función de ranking anterior. Para ello hemos llamado al módulo padre con el valor de $cutoff = -1$ que devuelve el diccionario con todos los documentos y sus scores sin ordenar y después aplicamos su módulo a cada uno de los documentos. Este módulo ha sido calculado en la creación de el índice, más concretamente a la hora de hacer *commit*.

3 Modelo Vectorial. Cálculos para el documento

Una vez programada la búsqueda por producto escalar, hemos procedido a estudiar una serie de datos sobre la query "obama family tree", con el objetivo de generar un documento con máximo score de entre los dados en el zip. Para ello, primeramente hemos obtenido el *idf* aproximado de cada uno de los 3 términos (teniendo en cuenta en el cálculo la existencia de un documento más en el zip), siendo estos:

$$idf_{obama} = 2.46,$$

$$idf_{family} = 0.84,$$
$$idf_{tree} = 1.4.$$

Con esto se concluye que "obama" es el término cuya frecuencia más aporta al score (al ser más raro, cuanto más aparezca, mejor), mientras que "family" es el que menos aporta. De esta manera, definimos "frec" como el número de veces que aparece la palabra "family" en nuestro documento, y la frecuencia de los otros dos términos es una escala lineal de *freq*, quedando los *tf* definidos como sigue:

$$tf_{obama} = 1 + ln(3 \cdot frec)$$
$$tf_{family} = 1 + ln(frec)$$
$$tf_{tree} = 1 + ln(2 \cdot frec)$$

Por lo tanto, haciendo una sencilla cuenta, obtenemos cuanto valdría el score de nuestro documento con respecto a *freq*:

$$score = tf_{obama} \cdot idf_{obama} + tf_{family} \cdot idf_{family} + tf_{tree} \cdot idf_{tree} =$$
$$= (1 + ln(3 \cdot frec)) \cdot 2.46 + (1 + ln(frec)) \cdot 0.84 + (1 + ln(2 \cdot frec)) \cdot 1.4 =$$
$$= 4.7 + ln(3 \cdot frec) \cdot 2.46 + ln(frec) \cdot 0.84 + ln(2 \cdot frec) \cdot 1.4 =$$
$$= 4.7 + ln((3 \cdot frec)^{2.46} \cdot frec^{0.84} \cdot (2 \cdot frec)^{1.4}) =$$
$$= 4.7 + ln(39.37 \cdot frec^{4.7})$$

Esto debe ser mayor que el score del mejor documento dado por el searcher. Dicho score es (aproximadamente) 25.27, por lo tanto:

$$4.7 + ln(39.37 \cdot frec^{4.7}) > 25.27$$
$$ln(39.37 \cdot frec^{4.7}) > 20.57$$
$$frec > \sqrt[4.7]{\frac{e^{20.57}}{39.37}}$$
$$frec > 36.4$$

Aproximando para arriba por posibles errores al perder décimas por el camino, cogemos $frec = 38$, es decir, "obama" aparecerá $3 \cdot 38 = 114$ veces, "family" 38 veces y "tree" $2 \cdot 38 = 76$ veces. Efectivamente, al ejecutar el documento aparece el primero del ranking. Además, este mismo documento sirve como primero del ranking para el algoritmo de búsqueda por coseno debido a que nuestro documento es mucho más corto que el primero del ranking por coseno, haciendo que se vea mucho menos penalizado y por tanto manteniendo la primera posición.

4 Estadísticas de frecuencia

En este apartado vamos a visualizar las gráficas de frecuencia total de términos y cantidad de documentos que contiene cada término y comprobaremos si se cumplen las Leyes de Zipf y Heap.

4.1 Frecuencia total de términos

Vamos a visualizar primero las tres gráficas.

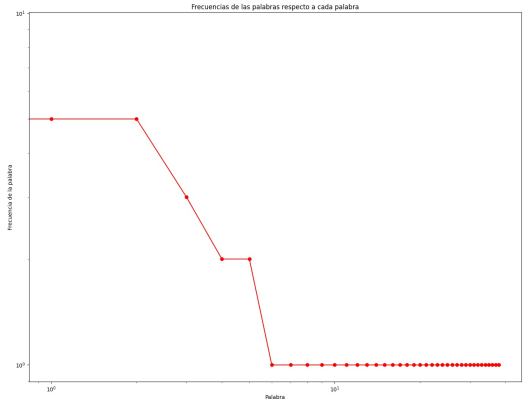

Figure 1: Frecuencia de términos. Colección "Toys".

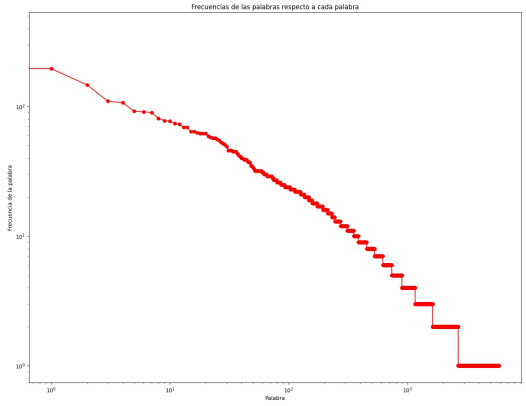

Figure 2: Frecuencia de términos. Colección "Urls".

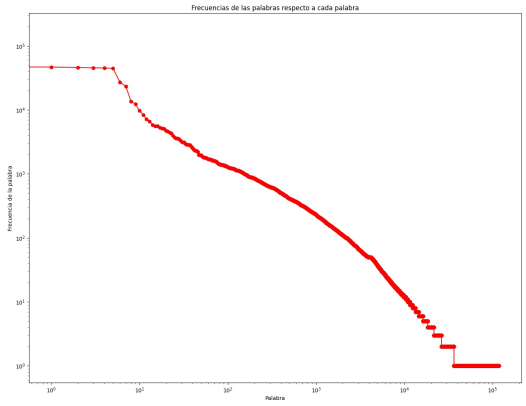

Figure 3: Frecuencia de términos. Colección "Docs1k".

Si nos fijamos en la colección de "Toys" no parece que se cumpla la ley de Zipf. ya que no parece que sea lineal en escala logarítmica. Sin embargo esto es debido a que la cantidad de vocabulario es muy baja y por tanto no hay suficientes puntos para realizar una buena interpolación. Esto lo vemos en que tanto para la colección "Urls" como la colección "Docs1k", al tener más vocabulario podemos ver una tendencia más lineal.

4.2 Número de documentos por término

Vamos a visualizar primero las tres gráficas.

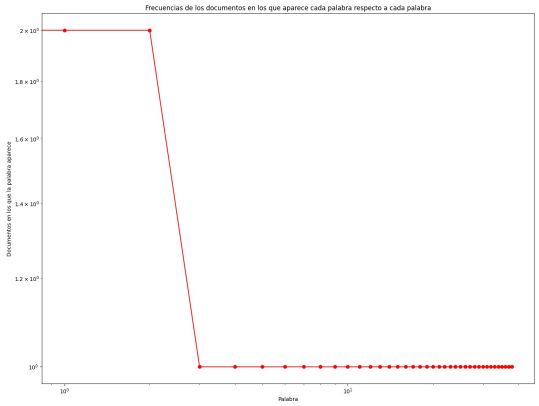

Figure 4: Número de documentos por término. Colección "Toys".

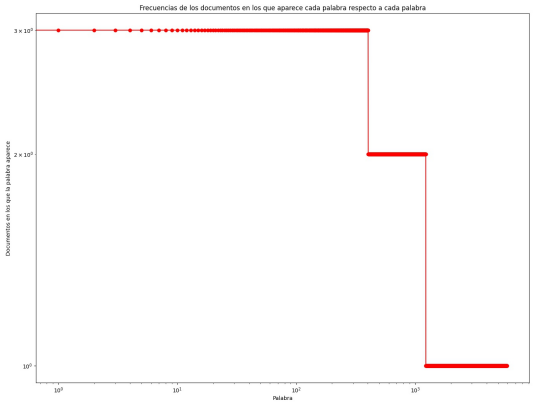

Figure 5: Número de documentos por término. Colección "Urls".

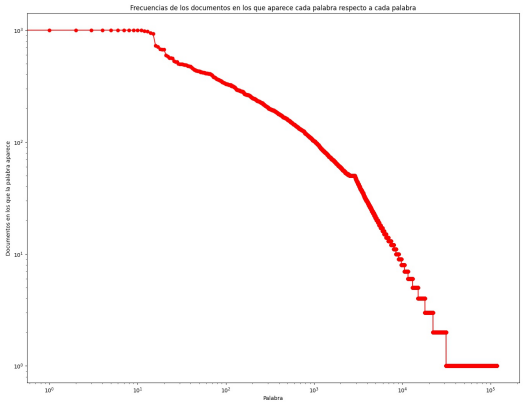

Figure 6: Número de documentos por término. Colección "Docs1k".

En este caso el mayor problema que hay es que el número de documentos de "Toys" y de "Urls", ya que es tan bajo que no se puede apreciar bien la ley de Heap. Esta ley la podemos ver de manera muy clara en la colección "Docs1k" donde podemos ver una gráfica logarítmica decreciente.

Si se desean ver las imágenes con mayor tamaño, se han incluido en la carpeta de la memoria.

5 Otros comentarios

1. En las funciones de ranking por producto vectorial, el parseo de la query se ha realizado dentro de la propia función, por lo que el constructor pasa un *None* a la clase padre como *parser*.

2. La creación del archivo de los módulos se realiza en la función *commit* dentro del módulo *Builder*.

3. La apertura de las distintas colecciones se realiza de la siguiente manera: primero se prueba si es una carpeta mediante un *try except*; si sale una excepción pasamos al caso de zip y realizamos la misma prueba y si sale una excepción finalmente suponemos que es un archivo txt. Por tanto si quisieramos probar con otro tipo de colección habrá que modificar la función, ya que nos hemos ceñido a lo que se nos da para esta práctica.

4. Hemos incluido en *src* el fichero *statistics.py* que es el encargado tanto de conseguir las frecuencias totales de los términos como el número de documentos por término. Para ejecutarlo (tanto este fichero como el *main.py*) hay que hacerlo desde la carpeta que contiene dentro la carpeta de las colecciones ya que si no el parser no funcionará bien.