Proyecto Final de Sistemas de Recuperación de Información

Rocio Ortiz Gancedo and Carlos Toledo Silva

Universidad de La Habana, Cuba

Abstract.

1 Introducción

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 en 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 la 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 una colección grande de términos 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 y las consultas y un diccionario que contiene a cada término con su respectivo $idfs$.

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. Además el tamaño del heap se fija según la cantidad de documentos que se deseen recuperar para cada consulta, para de esta forma evitar que el heap crezca indefinidamente.

3 Implementación del sistema

3.1 Preprocesamiento de los documentos

Las funciones para el preprocesamiento de los documentos de la colección Cranfield las podemos encontrar en "cran_preprocess.py". Primero tenemos la función `cran_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. Para saber el documento actual y la cantidad que de documentos que se ha visto simplemente se aumenta la variable `actual_document`.

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 asociad y en caso contrario se agregá 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 registrada 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 al 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 `cran_preprocessing` u otro que de una salida similar. Esté método devuelve un diccionario que tiene como llaves a cada uno de los términos, 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. Esté 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 $O(n \cdot m)$.

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 comprobar 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 que 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.

El último método "cran_recovered_documents" se utiliza para obtener, de la relación de relevancia entre consultas y documentos de la colección, la cantidad de documentos relevantes para cada una de las consultas.

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 y en otro la cantidad de documentos relevantes para cada consulta.

4.1 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 entero `m` que indica la cantidad de documentos que se quieren recuperar. Esta función devuelve una lista de los m 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 distinta de cero:
    - Si la cantidad de documentos en el heap alcanzó el valor deseado, o sea el valor de `m`, se empuja el documento en el heap y luego se extrae el de menor similitud.

    * De lo contrario, se empuja el documento en el heap.
- Crear la lista `result` de cardinalidad `m`
- 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.
- Reotornar 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 $O(\log m)$ 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 $O(n \cdot (a + b + \log m))$ siendo n el número de documentos. Pasar los documentos de la lista al heap tiene complejidad $O(m \log m)$. Obviamente $m \leq n$, pues no se pueden devolver más documentos de los que hay en la colección. Por tanto la complejidad temporal de esta función es $O(n \cdot (a + b + \log m))$.

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 enteros tal que la el elemento en la posición i indica la cantidad de documentos que se desean recuperar 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 $O(k \cdot n \cdot (a + b + \log m))$.

5 Análisis de los resultados del sistema

6 Conclusiones

References