UNIVERSITAT POLITÈCNICA DE VALÈNCIA

SISTEMAS DE ALMACENAMIENTO Y RECUPERACIÓN DE INFORMACIÓN

Memoria Práctica 4

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingeniería Informática

Curso 2020-21

Autores:

- Lishuang Sun (María)
- Antonio José Romero Barberá
- Vicent González Gramage
- Fabián Scherle Carboneres

ÍNDICE

1. Funcionalidades extra implementadas

Una vez implementada y testeada la funcionalidad básica hemos optado por desarrollar todas las funcionalidades extras propuestas en el boletín de la práctica. Dichas funcionalidades extras incluyen:

- *Stemming* de las noticias y las consultas.
- Ampliación *multifield* en la que se añade un nivel a los índices existentes, y así tomar en cuenta los campos de las noticias, de esta forma las consultas pueden incluir como prefijo uno de los campos (p.e. *campo:término*).
- Búsqueda de varios términos consecutivos utilizando las dobles comillas.
- Uso de *posting lists* posicionales.
- Ranking, permite devolver las noticias ordenadas en función de su relevancia.
- Permuterm, permite la búsqueda utilizando * y ? como comodines. En esta ampliación sólo se permite un comodín por palabra.
- Paréntesis, para modificar la prelación de los operadores lógicos.

2. Decisiones de implementación

2.1. Indexación

2.1.1. “index_dir” e “index_file”

El proceso de indexación comienza con el método *“index_dir”*, en este se recorre el directorio root y al encontrar un fichero con extensión *“.json”* llamamos al método *“index_file”* donde se realiza la indexación básica para el contenido de un documento, además de la indexación *multifield* y/o *positional* si se requiere. Una vez acabada dicha indexación se crean los índices *permuterm* y *stemming* opcionalmente.

En el caso de *“index_file”*, primero leemos el contenido del nombre del fichero JSON pasado como parámetro, luego se crea la entrada en el diccionario *self.docs* como un par (clave,valor) siendo la clave el identificador del documento y se le asocia como valor el *path* del documento.

Para cada noticia que se encuentre en el documento:

- Creamos una entrada en *self.news* como un par (clave,valor) donde el identificador de la noticia es la clave y su valor es una lista cuyo primer elemento es el identificador del documento y segundo elemento la posición de la noticia en el documento.
- Existe un atributo *"fields"* que nos indica los campos a indexar. Si no se requiere la opción *multifield* solo se indexa el campo *'article'*.
- Se extraen los tokens contenidos en cada campo en caso de que este tenga que ser tokenizado, si no es así, se separa cada término por el carácter espacio.
- Indexamos cada término de modo que usando el método adicional *"indexTerms"* obtenemos una posting list de los ID de las noticias donde aparece dicho término. Sin embargo, si se requiere un índice posicional, usamos el método *"indexPositionalTerms"*, de modo que además se guardan las posiciones de los términos en la noticia.

Nota: Los identificadores asociados a las noticias y documentos y las posiciones tanto de las noticias como los términos comienzan desde 0.

2.1.2. "make_stemming"

En el método *"make_stemming"* recorremos todos los campos que hemos guardado en *self.index,* y para cada campo, se extraen los términos para clasificarlos por *stems*.

A cada término se le aplica *"stem"*, un método del objeto *self.stemmer* de tipo SnowballStemmer, que devuelve el *stem* del término.

Diferentes términos de un mismo campo se agrupan en función de su *stem*, es decir, todos los términos con un mismo *stem* son miembros de una misma lista.

Al final obtenemos el índice *self.sindex*, que es un diccionario cuyas claves son los campos y cuyos valores son diccionarios con pares de stem (clave) y listas de términos (valor).

2.1.3. "make_permuterm"

En este método obtenemos el índice Permuterm de cada término en *self.index[campo]* y los guardamos en el índice *self.ptindex*. Este último es

un diccionario cuyas claves son los campos y cuyos valores son diccionarios con pares de términos de dicho campo (clave) e índice Permuterm asociado a dicho término (valor).

Cada índice Permuterm está representado como una lista. El símbolo final de cada término es el dólar $, de modo que tras añadir $ al final de cada término, vamos rotando por la derecha dicha cadena caracter a caracter usando el método "*permutar*", construyendo así los índices Permuterm.

2.1.4. "show_stats"

En *"show_stats"* se imprime por pantalla las estadísticas obtenidas al realizar el proceso de indexación explicado en el apartado *"index_dir" e "index_file"*, para ello se sigue un formato idéntico al de los ficheros de estadísticas proporcionados en la práctica:

- El número de días es la cantidad de documentos en *self.docs.*
- El número de noticias es la cantidad de noticias en *self.news*.
- El número de tokens indexados es la cantidad de términos indexados para cada campo en *self.index*.
- El número de *Permuterm* es la cantidad de *términos permutados* para cada campo de *self.ptindex*.
- El número de *stemm*ers es la cantidad de *stems* indexados para cada campo en *self.sindex*.
- Si se permiten consultas posicionales.

2.2. Búsqueda

2.2.1. "solve_query"

Este método se puede ver como la unidad central de la búsqueda, ya que recibe una consulta, la resuelve y devuelve una *posting list*.

Para ello, se divide la consulta por palabras, y verificamos si la consulta tiene 2 o menos palabras y no es una búsqueda posicional (tiene comillas). Inferimos que la consulta tendrá un solo término (obtenemos y devolvemos su *posting list*) o tendrá un *NOT término*, de tal manera que llamaremos a este mismo método para obtener la *posting list* del término y luego llamamos al "*reverse_posting"*.

Si la consulta no cumple la condición previa, recorremos las palabras de derecha a izquierda.

A su vez, comprobamos si hay un paréntesis de cierre en algún término, si lo hay contamos tanto el número de paréntesis de cierre como de apertura que contiene este término y hallamos el paréntesis de apertura relacionado con el primer paréntesis de cierre. Al encontrarlo, eliminamos estos dos paréntesis y realizamos una llamada recursiva para obtener la *posting list* de los términos que se encontraban entre los paréntesis.

Además, comprobamos si algún término forma parte de una búsqueda posicional. En ese caso, si esta búsqueda es de una palabra, ignoramos las comillas, caso contrario, buscamos las comillas donde inicia esta búsqueda posicional, y llamamos a "*get_posting"* para obtener el resultado de dicha búsqueda.

Si no se cumplen ninguna de las condiciones previas, simplemente se llama a "*solve_query*".

También verificamos si hay un NOT delante de un paréntesis o una búsqueda posicional, en este caso llamamos a "*reverse_posting*".

Finalmente, si llegamos al principio de la consulta devolvemos el resultado obtenido. Caso contrario, realizamos una llamada recursiva de todo lo que hay desde el inicio de la consulta hasta el término anterior al puntero y a continuación, vemos si la palabra en la que está el puntero es un AND o un OR, y obtenemos el resultado final de la query llamando a "*and_posting"* o *"or_posting"* respectivamente.

2.2.2. "get_posting"

El propósito de este método es obtener la *posting list* de un término o una cadena de varios términos dada como parámetro de entrada, para ello se hacen las siguientes comprobaciones e invocaciones:

- En caso de tener una cadena de términos entre comillas, invocamos al método *"get_positionals"*.
- En caso de tener una cadena de términos sin comillas, utilizamos *"and_posting"* sobre los términos.

- Comprobamos si el término tiene los comodines * y *?*, en ese caso invocamos *"get_permuterm"*.
- Si se requiere la opción *stemming* para la consulta y el término no lleva comillas, aplicamos el método *"get_stemming"*.

Si no se cumple ninguna de las comprobaciones anteriores, devolvemos la *posting list* asociada a dicho término.

2.2.2.1. "get_positionals"

El propósito de este método es devolver la posting list asociada a una secuencia de términos.

Primero obtenemos la *posting list* intersección resultante de aplicar *"and_posting"* sobre dichos términos. Luego recorremos dicha *posting list* y comprobamos si en cada noticia los términos son consecutivos. Para ello nos fijamos en el primer término y, para todas sus posiciones en las noticias, verificamos si la posición del segundo término es la posición del primero más 1, y si la del tercer término es la posición del primero más 2 y así sucesivamente con el resto.

2.2.2.2. "get_stemming"

Este método es similar al método *"make_stemming"*. Aplicamos el "*stem"* sobre un término que se nos pasa como argumento y creamos la lista a devolver vacía. Si existe el *stem* como entrada en el diccionario *self.sindex[campo]* devolvemos la *posting list*, si no, devolvemos la lista vacía.

2.2.2.2. "get_permuterm"

Se trata de obtener la posting list (ID noticias, y posiciones si se solicita) de cada término que cumpla las condiciones de prefijo y sufijo del *término con comodín* de una determinada consulta.

En el método *"get_permuterm"* distinguimos si el comodín usado es ? (flag=True) o * (flag=False). Llamando al método *"final"*, obtenemos el

término con comodín permutado, de modo que se añade el símbolo $ al final y rotamos caracter a caracter hasta que el comodín se sitúe en la última posición de dicha cadena, y ya podemos usarlo para buscar en los índices Permuterm correctamente.

Quitando el comodín, el *término* permutado debe ser prefijo del término t permutado del índice Permuterm *self.ptindex[campo][t]* que se está recorriendo actualmente, para que dicho término t cumpla las condiciones del *término con comodín*.

En el caso de que el comodín sea ?, dicho *término con comodín* permutado debe tener la misma longitud que el término t permutado, ya que este comodín sustituye exactamente un caracter.

En el caso de que el comodín sea *, dicho *término con comodín* permutado debe tener una longitud menor o igual que el término t permutado, ya que este comodín puede sustituir de 0 a muchos caracteres.

En ambos casos se invoca al método *"self.or_posting"* para evitar obtener una lista con ID de noticias repetidos, ya que dos términos diferentes pueden aparecer en el mismo campo y la misma noticia.

2.2.3. "reverse_posting"

Debemos devolver una *posting list* con todas las noticias que no estén entre las que nos pasan como argumento.

Para este método, la forma de operar es recorriendo todos los ID de las noticias guardadas en *self.news* y la lista que se nos pasa como argumento.

Creamos una lista resultante vacía. Recorremos ambas listas con un bucle y cuando los ID de las noticias de ambas listas coinciden, se omite dicha noticia. En cualquier otro caso, se van añadiendo las noticias a la lista resultante.

Puede ocurrir que hayamos terminado de recorrer la *posting list* pasada como parámetro de entrada. En ese caso, las noticias restantes en *self.news* se deben añadir a la lista resultante.

2.2.4. “and_posting”

Dadas dos *posting lists*, nos vamos a quedar con todos los ID de las noticias que aparezcan en ambas listas.

Puede que una lista tenga menor longitud que la otra, por lo que cuando se acabe de recorrer una de ellas, será el momento de devolver la lista resultante.

Durante el recorrido, si el ID de la noticia en una lista coincide con el de la otra, se añade dicho ID a la lista a devolver, en cualquier otro caso, se sigue recorriendo de uno en uno las *posting lists*.

2.2.5. “or_posting”

Dadas dos *posting lists*, queremos quedarnos con los ID de las noticias que estén en alguna de las dos listas.

Puede que una lista tenga menor longitud que la otra, por lo que cuando se acabe de recorrer una de ellas, los ID restantes de la lista más larga también se añaden a la lista resultante.

2.2.6. “minus_posting”

Dadas dos *posting lists* de números (*newid*) en orden creciente, se van comparando los elementos de dichas listas con el objetivo de obtener otra lista que contenga los elementos de la lista p1 que no estén en la lista p2.

En el caso de que coincida un *newid* de la lista p1 con un *newid* de la lista p2, no se guarda este *newid* en la lista resultante.

Puede que la lista p1 tenga menor longitud que p2, por lo que cuando se acabe de recorrer p1, ya obtenemos la lista resultante. En cambio, si p1 tiene mayor longitud que p2, los ID restantes de la lista p1 también se añaden a la lista resultante.

2.2.7. “solve_and_show”

Este método resuelve una consulta y muestra la información recuperada. Para ello hacemos uso del método *“solve_query”* sobre la consulta pasada como parámetro y obtenemos la *posting list*.

Para imprimir la información tenemos en consideración el siguiente formato:

- Número de resultados, que equivale a la cantidad de elementos en la *posting list* recuperada.
- *Score*, haciendo uso del atributo *self.weight* y del método “*rank_result”* obtenemos su valor y ordenamos los resultados.
- El identificador de la noticia contenido en la *posting list*.
- La fecha de la noticia, título y palabras claves contenidos en el fichero JSON del documento al que pertenece la noticia.
- *Snippet*, para implementar esta opción hemos optado por crear un método adicional al que llamamos “*procesarQuery”*. A este método le pasamos como parámetro la consulta en forma de lista y este se encarga de eliminar aquellos elementos que no influyen en el proceso de búsqueda de los términos en las noticias que se han obtenido como resultado, para ello primero eliminamos “AND”, “NOT”, “OR”, los paréntesis y las comillas. A continuación, detectamos si hay algún término con comodín, en cuyo caso tomamos en cuenta los términos que se le asocian en el atributo *self.ptindex*. En caso de que se haga uso de la opción *stemming*, tomamos en cuenta el *stem*. Finalmente, consultamos los términos que se encuentran al principio y final de la noticia, obteniendo de ellos fragmentos de esta.

2.2.8. “rank_result”

Dados dos parámetros, una consulta y una *posting list* con el resultado de dicha consulta, este método puntúa los resultados dados y los devuelve ordenados según la puntuación otorgada.

Para ello, primero eliminamos todos los valores no alfanuméricos de la consulta. Consideramos el posible hecho de que un término de la consulta sea *multifield*, separando la palabra en campo y término, eliminando los

valores alfanuméricos de cada uno y luego volviéndose a juntar con el ‘:’ que indica que la palabra pertenece a un campo distinto de *‘article’*.

Después de obtener la consulta con solo los términos, aplicamos el pesado *tfxidf* para calcular las puntuaciones de cada noticia con cada término, sumando todos estos pesados al final y dando una puntuación total a la noticia.

Tras obtener la puntuación total de la noticia para esta consulta, guardamos el valor en *self.weight*, el cual es posteriormente tomado en otros métodos para mostrar la puntuación de la noticia en la consulta.

Finalmente, devolvemos la *posting list* ordenada.

3. Coordinación y contribución

Para llevar a cabo las distintas implementaciones nos hemos dividido los métodos de la siguiente manera:

- **Antonio:** *“make_stemming”*, *“get_stemming”*, *“reverse_posting”*, *“and_posting”.*
- **María:** *“show_stats”*, *“make_permuterm”*, *“get_permuterm”*, *“minus_posting”*
- **Vicent:** *“solve_query”*, *“or_posting”*, *“rank_result”.*
- **Fabián:** *“index_dir e index_file”*, *“get_positionals”*, *“solve_and_show”*, *“get_posting”.*

Una vez implementados, hemos optado por subir los ficheros en una carpeta compartida vía Google Drive, y el líder del equipo (Fabián) se encargaba de juntar los métodos en un solo fichero y subirlo a la misma carpeta compartida para que luego todos los miembros pudieran llevar a cabo las distintas pruebas para comprobar que todo estuviera en orden.

4. Consideraciones

Como consideraciones para la implementación de algunos métodos caben mencionar las siguientes:

- "*get_posting*", otorgamos prioridad a los términos que contienen comodines sobre la opción *stemming*.
- "*solve_query*", hemos considerado una forma de resolución recursiva, en la que la consulta se vaya reduciendo en tamaño con cada llamada hasta llegar a un término del cual se pueda obtener su *posting list*.
- "*solve_and_show*", hemos optado por solo considerar el campo 'article' de las noticias, al generarlo trabajamos a nivel de palabras obteniendo las posiciones del primer y último término de las noticias y mostrando los términos más cercanos a dichas posiciones.
- *"rank_result"*, hemos aplicado el pesado *tf*x*idf*, estudiado en clase. Este peso consiste en el cálculo del *idf* de un término (la frecuencia de documento inversa de t), consistente en que según el número de documentos de una colección aparezca un término, menor valor en los resultados finales tendrá, este valor se calcula con la fórmula $\log_{10}$(Número de documentos total/Número de documentos en los que aparece el término); y el cálculo del *tf* de un documento respecto a un término, que consiste en que a mayor número de apariciones de un término en un documento, mayor puntuación tendrá en el ranking (si el término es *multifield* y siendo este "date", al no tener que almacenarse cuando se crea el índice, se otorga un *tf* = 0). La fórmula de la cual se obtiene *tf* es 1+$\log_{10}$(Número de apariciones del término en el documento). Tras obtener tanto el *idf* como el *tf* de un término respecto a un documento, multiplicamos ambos valores y los añadimos a la puntuación del documento. Esto lo realizamos con todos los términos de la consulta, y obtenemos una puntuación final del documento.