Moogle (Proyecto de Pogramación I)

Laura Alonso Rivero C-122

En este proyecto se propuso la tarea de crear una aplicación web capaz de encontrar un texto en un conjunto de documentos, para lograrlo se debía trabajar a partir de la biblioteca de clases ***'MoogleEngine'*** modificando el algoritmo de búsqueda.

El proyecto está estructurado de la siguiente manera:

- Para la carga de los datos existe una clase ***'LoadData'*** que contiene un diccionario ***('wordsDictionary')*** cuya llave está formada por las palabras de los documentos y como valor un diccionario ***('docsDictionary')*** que tiene como llave el id de cada documento en los que aparece la palabra y como valor los detalles del documento dados en una clase ***('WordDetails')*** que presenta el número de ocurrencias de la palabra en el documento y su **TF-IDF**.
- Existen clases auxiliares donde se guardan los detalles de los documentos (clases ***'Doc'*** y ***'directoryNames'*** almacenadas en una lista).
- Para la búsqueda de la consulta de palabras se utiliza el método ***'Search'*** que a partir de las palabras solicitadas por el usuario lista los detalles de los documentos encontrados (entre ellos: el camino completo, el nombre del documento el valor de **TF-IDF**).

Diccionario de palabras (palabra,
Diccionario de documentos (docId,
DetallesDelDocumento-palabra))

DetallesDelDocumento-palabra [número de ocurrencias de la palabra en el documento,
TF-IDF de la palabra en el documento]

Doc [id de la ruta, nombre del documento, numero de palabras del documento]

DirectoryName [id de la ruta, ruta completa del documento]

Diccionario de Documentos (docId, Doc)

La carga de los datos

Al iniciar el servicio web se realiza la carga de los datos. Los datos de entrada se encuentran en la carpeta ***'Content'***, en este caso se creó la clase ***'LoadData'*** para obtener el contenido de todos los ficheros txt que se encuentra en dicha carpeta.

La clase ***'LoadData'*** contiene como método principal a ***'Initialize'*** que se encargará de inicializar la base de datos dada partiendo de la obtención de las rutas de acceso a los documentos. Se escogerá una a una la ruta de acceso y siempre que sea diferente se llenará la lista de ***'directoryNames'*** que va a ser de tipo ***'directoryName'*** (clase que contiene la ruta de acceso y otorgara un id específico para cada una). Luego se leerá el contenido del documento utilizando el método ***'NormalizeDoc'*** de la clase ***'Normalize'*** para eliminar caracteres distintos a letras y números, además separará las palabras por espacios guardándolas en la lista de string ***'wordsFromFile'***. Posteriormente se obtendrán los valores de la clase ***'Doc'*** (contiene el nombre del documento, el id de la ruta y la cantidad de palabras que contiene) que se añade a la lista ***'documents'***.

Para cada fichero que se lea se procede a llenar el diccionario ***' wordsDictionary '*** (se insertaran solo las palabras que tengan un tamaño mayor estricto que 2 (así se eliminarán las letras y monosílabos que tendrán poca relevancia a la hora de procesar la consulta), los valores para esta llave serán otro diccionario llamado ***'docsDictionary'*** (su llave serán los documentos y los valores la clase ***'WordDetails'*** (presenta el **TF-IDF** de la palabra en el documento y su número de ocurrencia))).

El **TF-IDF (Term Frequency - Inverse Document Frecuency)** mide que tan relevante es una palabra en un conjunto de documentos gracias a la multiplicación del **tf** y el **idf**. El **tf** es la cantidad de veces que aparece una palabra en un documento entre la cantidad de palabras que posee el documento. Mientras el **idf** es el logaritmo de la razón entre el total de documentos que se procesan y los documentos que contienen a la palabra. En resumen, una palabra es más relevante cuando un documento la contiene mucho, pero en cambio esta aparece en pocos de los documentos de la muestra. En este caso para evitar que palabras como preposiciones, conjunciones y otras igual de comunes sean incorrectamente consideradas relevantes se estableció que si aparecía en más del 80% de los documentos fuesen descartadas

$$TF - IDF = \frac{\text{cant de apariciones}}{\text{cant de palabras}} \times \log \frac{\text{total de docs}}{\text{docs con la palabra}} \tag{1}$$

Búsqueda de palabras

Una vez inicializada la data se procede a pedir al usuario que introduzca su búsqueda. El usuario puede elegir una palabra o una frase e incluir en cada palabra operadores tales como:

- ‘¡‘ (exige que la palabra no se encuentre en los documentos devueltos)
- ‘^‘ (exige que la palabra se encuentre en los documentos devueltos)
- ‘*‘ (da mayor importancia a los documentos que contengan esta palabra)

Si estos operadores son usados erróneamente se detendrá la búsqueda automáticamente y se enviará un mensaje a la página advirtiendo su correcto uso. Casos de usos erróneos:

- Si la palabra contiene ‘¡‘ como primer carácter y después le siguen otros operadores
- Si la palabra contiene ‘^‘ como primer carácter y después le siguen otros operadores
- Si la palabra contiene ‘*‘ y después le sigue ‘¡‘

El usuario debe tener en cuenta es que si se utiliza ‘*‘ y después le sigue ‘^‘ el programa solo presentará como respuesta de la búsqueda únicamente los documentos que contengan esta palabra.

El método de búsqueda ***‘Query‘*** se encuentra en la clase ***‘Moogle‘*** y devolverá una variable de tipo ***‘SearchResult‘*** (contiene un arreglo de ***‘SearchItem‘*** y un string). Este empezara normalizando la query por el método ***‘NormalizeQuery‘*** de la clase ***‘Normalize‘*** para eliminar caracteres distintos a letras, números y operadores, además separara las palabras por espacios guardándolas en la lista de string ***‘queryListed‘***.

Una vez normalizada se procede a verificar por cada palabra si contiene algún operador al inicio. En caso positivo y que este bien empleado se devolverá la palabra sin operador para poder comenzar la búsqueda y se guardará la información del operador en una variable de tipo ***‘Operators‘*** para ser utilizada más adelante. En caso negativo se devolverá la palabra tal cual fue introducida.

Con esta palabra se procederá a llenar la lista ***‘results‘*** de tipo ***‘Searcher‘*** (contiene el nombre del documento donde está presente, el id de la ruta, el número de ocurrenciasde la palabra y el valor del *TF-IDF*) con el método ***‘Search‘*** de la clase ***‘LoadData‘*** siempre que la palabra se encuentre en alguno de los documentos o que no se haya considerada una palabra poco relevante.

En caso de que no se encuentre la palabra se llenará la lista ***‘suggestions‘*** que buscará por el método de ***‘FindSimilarities‘*** de la clase ***‘LevenshteinMethod‘*** las cuatro palabras presentes en el diccionario más similares a esta. Este método utiliza el algoritmo de Levenshtein (utiliza matrices siendo las filas las letras de la palabra de la query y las columnas las letras de la palabra del diccionario) para dar este criterio de similaridad. Para hacer la búsqueda de palabras más efectiva se consideró que si una palabra del diccionario tenía un tamaño menor/mayor o igual que el de la palabra a la que se le buscaba la similaridad aumentado/disminuido en 3 debía ser descartada pues los cambios necesarios para convertir una en la otra (eliminación, inserción o sustitución de caracteres) serían superiores al rango deseado (un valor de distancia menor o igual que 30). Luego se

añadirán a la lista las 4 palabras de menor valor (más similares).

Si la palabra es encontrada en los documentos entonces se procede a llenar por cada documento los datos de la lista ***'finalResults'*** de tipo ***'SearchItemPLus'*** (contiene el títulodel documento, el snippet, el score, la cantidad de palabras de la query que presenta el documento y el id de la ruta del documento). Si dos documentos presentan el mismo nombre, pero están en carpetas distintas igual se escogerán debido a que se le da un índice por su ruta y nombre. Si el índice no se encuentra en la lista añadirá el *TF-IDF* y lo sumará al peso del operador '*' si existe convirtiendo esto en el score y añadirá la palabra a la lista de palabras de la query que presenta el documento. De lo contrario suma el *TF-IDF* de existente con el de la nueva palabra y sumará al peso del operador '*' si existe al score existente y añadirá la palabra a la lista de palabras de la query presentes en el documento.

El snippet del documento se calculará a partir de la primera palabra de la consulta que aparezca en él. Para obtenerlo se llamará al método ***'Snippet'*** de la clase ***'Doc'***. Se estableció que este debía devolver un substring que contuviese a la palabra y 50 caracteres hacia la derecha partiendo del índice de la primera letra de la palabra (siempre que fuese posible) y 50 hacia la izquierda (siempre que fuese posible), teniendo en cuenta el tamaño de la palabra.

Una vez que se buscan todas las palabras de la consulta si el resultado de la misma es nulo se enviará al usuario una lista de palabras como sugerencia para que esta sea efectiva.

En caso de tener resultados se procederá a recorrer la lista de palabras con operadores. Si se encuentra que alguna palabra contiene '¡' se eliminan de la lista de búsqueda todos los documentos que la contengan. En cambio, si la palabra posee '^' se eliminan todos los documentos que no la contengan. Posteriormente se procede a ordenarlos partiendo de los siguientes criterios:

- Si el documento que se está verificando posee menor cantidad de palabras de la consulta que el anterior se encontrará en una posición inferior a este.
- Si el documento que se está verificando posee mayor cantidad de palabras de la consulta que el anterior se encontrará en una posición superior a este.
- Si la cantidad de palabras de la consulta de ambos documentos es igual, si el score de este documento es menor que el del anterior se encontrará en una posición inferior.
- Si el score del documento que se está verificando es mayor que el anterior se encontrara en una posición superior.
- Si los scores también son iguales se mantendrán en el orden que se encuentran.

Si no se encuentran operadores procederá a ordenarlos con los mismos criterios anteriores.

Luego se llenará una lista de tipo ***'SearchItem'*** (contiene el título del documento, el snippet y el score) con los elementos correspondientes de la lista ***'finalResults'***. Para

terminar, se devolverá una nueva variable de tipo ‘SearchResult‘ a la que se le pasará como primer valor esta lista llevada a arreglo.

De esta forma el usuario podrá saber en qué documentos esta la query buscada y se le da solución a la tarea principal.

Propuestas para el mejoramiento de la web en el futuro

- Se puede utilizar el algoritmo de Porter a la hora de hacer el diccionario de palabras pues con este se va a eliminar todos los sufijos tales como género, numero, persona, tiempos verbales, diminutivos... y se quedara solamente con las raíces de las palabras lo que reducirá el tamaño del diccionario y hará que la carga de la basa de datos y posterior búsqueda de la query sea más rápida y eficiente.
- Se puede guardar la data en un fichero estructurado para ahorrar tiempo a la hora de inicializar y recalcularlo solo cada vez que se carguen otros ficheros en la carpeta ***‘Content‘***.
- Se puede realizar una búsqueda exacta de frase de más de dos palabras que estén entre comillas en el criterio de búsqueda.