Proyecto Moogle!

1er año de la carrera de Ciencias de la Computación
Universidad de La Habana

Vagner Válik Alfonso Rivero
C121

Contenido

1- Base de Datos.

Estructura y Disposición

La base de datos del proyecto “Moogle!” consiste en una serie de documentos de texto situados en la carpeta “Content”. Esta implementación llama y relaciona a cada documento con sus propiedades mediante el orden en que están dispuestos en la carpeta “Content”. Así, si hay n documentos, cada propiedad tiene n elementos, y se señala el n-ésimo documento a través del índice n-1. Para extraer y ser capaz de manipular esa masa de información, he implementado tres clases diferentes dentro de la librería de clases “MoogleEngine”. Estas son las clases: DataBase, Matrix y Vector.

La clase “DataBase” maneja la información textual de cada documento: nombre de archivo, contenido y vocabulario global. La clase “Matrix” toma un objeto DataBase, y construye un array de objetos “Vector”, donde cada uno de estos representa un documento. La clase “Matrix” es la clase principal del algoritmo de búsqueda, pues es donde ocurren los cálculos de score, donde se obtiene la sugerencia a partir del query, donde se genera el Snippet, y donde se ordena toda la información obtenida para presentarla al usuario. Por último, la clase Vector se encarga de representar cada documento de forma numérica.

Clase *DataBase*

La clase “DataBase” se ocupa de extraer el contenido de la base de datos, y llevarlo a un formato manipulable, de modo que se pueda acceder fácilmente a la información y realizar operaciones con la misma. Un objeto “DataBase” se inicializa pasándole un argumento en forma de string, que especifica la URL de la base de datos, en este caso “Content”. La clase contiene los siguientes campos y propiedades:

string address – *Contiene la URL de la base de datos.*

string[] AllText – *Contiene el contenido de cada documento en forma de string. Este campo se inicializa mediante el método privado “LoadDataBase” que utiliza los métodos “Directory.GetFiles” y “File.ReadAllText” para extraer el texto de cada archivo, y además elimina los saltos de línea con el método “String.Replace”, para facilitar la obtención del snippet más adelante.*

string[] FileNames – *Contiene los nombres de los documentos, para el parámetro “title” de la clase “SearchItem”. Estos se obtienen a través del método “Path.GetFileName”.*

string[][] Docs – *Contiene todas las palabras de cada documento en forma de un array de strings. El campo se inicializa mediante el método privado “GetWords”, el cual itera sobre el campo “AllText” y separa las palabras de cada documento utilizando el método “Regex.Split” con el patrón de expresión regular “[^a-zA-Z]” como delimitador.*

string[] AllWords – *Contiene el vocabulario global que se obtiene de la unión de todos los conjuntos de palabras de cada documento. Este es ordenado alfabéticamente a través del método "Array.Sort".*

Clase *Matrix*

La clase "Matrix" se ocupa de *"vectorizar"* el contenido de cada documento y calcular la relevancia de las palabras para llevar a cabo el proceso de búsqueda. Esta clase contiene el método "GetScores" el cual es responsable de realizar la búsqueda y devolver los documentos ordenados por relevancia, y demás elementos importantes. Profundizaré en la implementación de este método más adelante. Un objeto "Matrix" se inicializa pasándole un objeto "DataBase". La clase "Matrix" contiene los siguientes campos y propiedades:

DataBase Docs – *Objeto "DataBase que se pasa al constructor.*

float[] Idfs – *Contiene el Inverse Document Frequency (IDF) de cada palabra en el vocabulario. Este campo se inicializa a través del método "GetIdfs", el cual calcula la relevancia global de cada palabra.*

string[] voc – Obtiene el vocabulario del objeto "DataBase" pasado al constructor para hacer más fácil su acceso, ya que se convertirá en un recurso recurrente en esta clase.

Vector[] matrix – Contiene cada uno de los documentos en forma de objetos "Vector". Este campo se inicializa mediante el método "GetMatrix".

Clase *Vector*

La clase "Vector" se encarga de construir un array de floats (vector) a partir de un documento. Esta clase tiene dos constructores:

El primero se encarga de construir los vectores genéricos (aquellos que representan documentos). Este constructor recibe como argumentos un array de strings "words" (palabras del documento), otro array de strings "vocabulary" (vocabulario global) y, por último, un array de floats "Idfs", que contiene el IDF de cada palabra. Al construir un vector se obtiene un array de floats que contiene el Term Frequency – Inverse Document Frequency (TF-IDF) de cada palabra en el documento, en base al vocabulario ordenado. Esto se logra inicializando un array de floats del tamaño del vocabulario, luego iterar sobre el mismo calculando el TF de cada palabra en el documento, para luego multiplicar el resultado por el IDF. El vector se obtiene mediante el método "Vectorize".

El segundo constructor es una sobrecarga del primero, y se encarga de construir el vector "query", que representa la búsqueda hecha por el usuario. Este, a diferencia del primero, sólo recibe dos argumentos: un array de strings "words" (palabras del documento), y otro array de strings "vocabulary" (vocabulario global). El array "Idfs" no es importante en este vector, ya que vamos a construir un array de floats del tamaño del vocabulario, pero sólo vamos a poner el número 1 en las posiciones que contengan palabras insertadas por el usuario, y 0 en el resto de las posiciones. El vector se obtiene mediante el método (GetQueryVector).

La clase vector contiene los siguientes campos: string[] words (palabras del documento), string[] voc (vocabulario global), float[]? idfs (campo declarado como “nullable” para el vector “Query”, contiene los Idfs de todas las palabras) y float[] vector (ceros y unos para el query, TF-IDF’s para el vector de documento).

El método “Multiply” de la clase “Vector” toma como argumento otro objeto “Vector” (el cual será el query cuando invoquemos este método) y multiplica los valores de cada posición i de uno y otro vector para obtener el score del query en ese documento. Además, mantiene un registro de las palabras que coincidieron, así como de su relevancia, en forma de arrays. Luego ordena las coincidencias en base a su relevancia, de más relevante a menos relevante, para luego devolver un Tuple de tres elementos: el score (float score), las coincidencias (string[] matches), y la relevancia de esas coincidencias (float[] matchesRel).

2 – Búsqueda

Corrección

Para efectuar la corrección del query, he decidido que lo mejor era, en lugar correr el query y sugerir un query alternativo corrigiendo las palabras mal escritas, correr directamente la sugerencia. Esto se debe a que si existe una palabra mal escrita en el query, o que no se encuentre en el vocabulario de la base de datos, es irrelevante buscar tal palabra, pues esta simplemente no brindará resultados.

Método *GetSuggestion*

Este método perteneciente a la clase “Matrix” es el encargado de tomar el query y ofrecer una sugerencia a este, reemplazando las palabras mal escritas, o palabras que no se encuentran en ningún documento, por la palabra más parecida a ella en el vocabulario, calculando la *Distancia de Levanstheins* entre cada palabra del query, y cada palabra del vocabulario. Este método llama al método “GetCorrection” para cada una de las palabras del query, el cual, a su vez, recibe la palabra e itera sobre el vocabulario llamando al método “GetLevanstheins” en cada iteración, devolviendo un entero que representa la distancia entre una y otra palabra, para, finalmente, devolver la distancia más pequeña y el índice de la palabra más cercana del vocabulario. El método “GetSuggestion” se ejecuta directamente en el archivo “Index.razor”, dentro del método “RunQuery”, remplazando la llamada de la función “moogle.Query(query)” por “moogle.Query(moogle.M.GetSuggestion(query)”.

Método *Matrix.GetScores*

Este método toma por argumento un objeto “Vector”, (el cual se inicializa en el método “Moogle.Query”, a partir del query corregido de antemano), y devuelve un Tuple de cinco elementos, a saber: un array de strings “name” que contiene los nombres de los archivos, los cuales serán pasados al parámetro “title” de “SearchItem” en el método “moogle.Query”; un array de floats “score” el cual contiene el score de cada documento, que luego será pasado al parámetro “score” de “SearchItem” en el método “Moogle.Query”; un array de arrays de strings “matches”, que será pasado al parámetro “matches” de “SearchItem” (agregado por mí), el cual contiene las palabras coincidentes del query con cada documento, ordenadas por relevancia, ,

que luego será utilizado para resaltar las mismas en el snippet; un array de arrays de floats “matchesScores”, que contiene la relevancia de las coincidencias para cada documento (este en realidad no tiene un función fuera del método todavía, pero decidí mantenerlo para futuras modificaciones de la aplicación en las cuales pudiera hacer uso del mismo); y por último, un array de strings “snippet”, el cual será pasado al parámetro “snippet” de “SearchItem”.

El método “Matrix.GetScores”, básicamente llama al método “Vector.Multiply” desde el Vector “query”, pasándole cada uno de los vectores del objeto “Matrix” (documentos), e inserta la información resultante en los arrays (o listas que luego se convertirán en arrays) que conformarán el Tuple de *return* (name, score, matches, matchesRel, snippet). Cada uno de estos arrays se ordenarán en base al array “scores”, a través del método “Array.Sort” (por supuesto clonando el array “scores” tantas veces como arrays haya que ordenar), y luego invirtiendo cada array mediante el método “Array.Reverse”, para que queden ordenados de mayor a menor.

El constructor de la clase Moogle inicializa un objeto “DataBase” (Docs) y un objeto “Matrix” (M). En el método “Moogle.Query” se crea un objeto “Vector” a partir del parámetro “query” y se le pasa a la función “this.M.GetScores”, guardando el resultado en una variable genérica (var), luego se crea una lista de objetos “SearchItem” y a cada uno se le pasan los parámetros “name”, “snippet”, “score” y “matches”, y, ya que el método devuelve estos resultados ordenados de mayor a menor, se deja de iterar cuando el score es igual a 0, eliminando de la lista los documentos sin relevancia.

3- Snippet

Para obtener el snippet de cada documento, primero pensé simplemente en tomar la coincidencia más relevante de cada documento, y encontrar en el texto dónde se mencionaba la palabra. Luego me di cuenta de que esto no era suficiente, sino que debía encontrar la parte del texto donde se encontraran el mayor número de coincidencias posibles, para dar constancia al usuario de la relevancia de los resultados de su búsqueda. Luego de haber completado esta tarea, me encontré a mí mismo buscando en los snippets, después de haber realizado una búsqueda de prueba, las palabras que había insertado en el buscador, para ver si el snippet tenía sentido. Entonces me di cuenta, no sólo tenía que devolver un snippet válido y el más relevante de todos los posibles, sino que también era necesario resaltar las palabras coincidentes para que el usuario pudiera advertirlas rápidamente. Próximamente expongo cómo logré hacer todo lo anterior:

**Nota: El método “GetSnippet” está implementado para devolver una substring de aproximadamente 200 caracteres.*

Método *GetSnippet*

Este método de la clase “Matrix” se llama desde el método “GetScores” de la misma, y se le pasan los siguientes argumentos: un string “text” (el texto del documento, sacado del campo “AllText” de la clase “DataBase”), un array de strings “words” (las palabras coincidentes

en ese documento, y las que habrá que buscar en el texto para extraer el snippet), y un array de floats “relevance” (la relevancia de cada coincidencia, para darle prioridad a palabras más importantes). He mencionado que este método está implementado para devolver la parte del texto con el mayor número de coincidencias diferentes, dentro de una ventana de 200 caracteres aproximadamente. Así, si existen en el texto 4 coincidencias de la búsqueda, por ejemplo, y en una ventana de 200 caracteres la palabra más importante se repite 10 veces, pero otra ventana contiene las otras 3 palabras sólo una vez, el método devolverá la segunda. Por otra parte, si en dos ventanas se repiten palabras diferentes el mismo número de veces, el método dará prioridad a la palabra más relevante. Para implementar esta característica, usé un *approach* de “sliding window”, iterando sobre todas las posibles ventanas de 200 caracteres en el texto. En cada iteración, se itera sobre cada palabra, y se llama al método “Matrix.GetAllMatches”, el cual devuelve la cantidad de coincidencias de la palabra dentro de esa ventana. Si devuelve 0 (no hay coincidencias), el bucle continúa a la próxima palabra, si encuentra coincidencias, se guarda una puntuación que resulta en el cálculo del número de coincidencias, multiplicado por la relevancia de la palabra, multiplicado por un escalar que aumenta en 1000 (cantidad arbitraria) cada vez que se encuentre una palabra diferente en la misma ventana, garantizando que las ventanas con más coincidencias diferentes tengan prioridad. Luego esa puntuación se compara con la máxima puntuación actual (que empieza en 0, por supuesto), y si es mayor, se sobrescribe la máxima puntuación y se guarda el índice de inicio de esa ventana. Luego el bucle continúa a la próxima ventana y la puntuación se resetea a 0.

Sin embargo, este método tiene un problema, y es que, ya que el índice de inicio de la ventana sólo se sobrescribe si la puntuación de esa ventana es estrictamente mayor a la puntuación máxima actual, cuando el bucle encuentra la ventana ganadora, y sigue iterando, el índice de inicio seguirá siendo el mismo, pues la puntuación de las ventanas subsecuentes será igual, o menor. Esto ocasiona que el snippet extraído contenga las coincidencias siempre al final del string, lo cual es, más que nada, un problema estético. La solución a esto es un método que corrija el índice de inicio, y “centre” las coincidencias en el snippet. Esto, y la clase “highlight” que expondré más adelante cumple la sola función de mejorar la experiencia de usuario.

Layout en Razor

Para centrar las coincidencias dentro del snippet, he implementado el método “Matrix.Correct”, el cual acepta como argumentos un string (snippet), y un array de strings (coincidencias dentro del snippet), y devuelve un entero, que es sumado al índice de inicio del snippet. Si es negativo, el índice disminuye, y, si es positivo, aumenta. Este método utiliza un objeto “MatchCollection” para encontrar el índice de la primera y última coincidencia. Si los índices son iguales, significa que sólo existe una coincidencia, en cuyo caso, y dado que tenemos garantizado que ésta se encontrará al final del string, el método devolverá 100 (centrando la palabra en el snippet). Si los índices son iguales, se sumarán. El “rango seguro” para la suma, y que indica que las coincidencias ya están centradas, es de 180 a 200. Por ejemplo, y esto es algo que podría pasar, si la primera coincidencia se encuentra en el índice 0, y la segunda se encuentra en el 190, efectivamente se encuentran centradas. Si la suma cae

en este rango, el método devolverá 0. En cambio, si la suma es menor a 180, lo que indica que las coincidencias están inclinadas hacia el principio del snippet, el *return* será 180 substraído de la suma, lo que resultará en un número negativo, que corregirá el inicio del snippet hacia la izquierda, centrando las coincidencias. De igual manera, si la suma es mayor a 200, significa que las coincidencias están inclinadas hacia el final del snippet, en cuyo caso el método devolverá un entero positivo, resultante de substraer 200 de la suma. En cualquier caso, el snippet resultará exitosamente centrado.

La segunda característica estética (o de experiencia de usuario) que he implementado, es la de resaltar en el snippet las palabras coincidentes. Esto lo he logrado mediante el método "Matrix.GetMin". Este método toma como argumentos una string (snippet), un array de strings (coincidencias) y un entero (posición), y devuelve un Tuple de dos elementos que contiene un string "word" (la primera palabra coincidente en el texto, a partir de la posición dada), y un entero "pos" (la posición de esa palabra). Cuando luego de la posición dada, no es encontrada ninguna coincidencia, el método devuelve (null, -1). Así, para resaltar las palabras coincidentes, sin afectar el output del snippet, he implementado lo siguiente en el archivo "Index.razor":

```
<div class="item">
    <p class="title">@item.Title</p>
    <p>… @{
        string snip = item.Snippet;
        int pos = 0;
        while(true){
            var x  = moogle.GetMin(snip, item.Matches, pos);
            if (x.pos == -1){
                @(snip.Substring(pos))
                break;
            }
            @snip.Substring(pos, x.pos – pos))
            <span class="highlight">@x.word</span>
            pos = x.pos + x.word.Length;
        }}… </p></div>
```

Luego de esto solo queda cambiar el *background-color* de la clase "highlight" en el archivo "Index.razor.css" para que las palabras coincidentes sean resaltadas.

4- Upcoming features

En este capítulo, voy a listar algunas de las características que quiero implementar en el proyecto en el futuro. Independientemente de que el tiempo me alcance para llevarlas a la exposición, me gustaría desarrollarlas con el objetivo de obtener práctica y experiencia.

Operadores

Esta es una de las características a las que daré mayor prioridad, y trataré de implementar al menos 2 operadores antes de la exposición.

Accesibilidad de los documentos en la interfaz.

Hacer que los documentos resultantes de la búsqueda sean accesibles para el usuario, lo cual es sumamente importante para un buscador. Ya he intentado la implementación de esta característica, desafortunadamente sin éxito, ya que no encuentro la forma de acceder a los documentos locales desde el *localhost.* Confío en que lo podré solucionar, al fin y al cabo, es solo agregar un *anchor tag* al título con la URL correcta.

Lematización de palabras.

Esta es una de las múltiples formas de hacer la búsqueda más efectiva, y consiste en llevar las palabras a su forma primitiva. La implementación de esta característica requiere de una librería externa, o de la implementación de un patrón de lematización propio. Trataré de implementarlo por una de esas dos vías, preferiblemente la primera.

Mejoras de rendimiento

He estado probando este proyecto con una base de datos de 7 documentos, y, según mi percepción, 5 minutos de duración para el proceso de carga de la base de datos es demasiado. Hice unas pruebas y creo que el proyecto está cargando la base de datos dos veces, por alguna razón. Este es uno de los problemas que tendré que solucionar más adelante.