Proyecto Programación I

MOOGLE!

Edel Rey Díaz C-113

En el momento de aceptar la ardua tarea de implementar el motor de búsqueda Moogle!, me surgieron varias interrogantes sobre las cuales construí las principales ideas a desarrollar, para así dar solución a este problema:

- En primer lugar, la lectura y normalización de los datos contenidos en los documentos sobre los que se desea establecer la búsqueda.

- Utilizar estos datos en una especie de abstracción de forma tal que me permita clasificarla según algunos criterios de relevancia.

- Aplicar la normalización de la consulta, clasificar sus datos y realizar la búsqueda.

- Implementación de un grupo de operadores a aplicar sobre la consulta.

- La devolución de una sugerencia en caso de que no aparezcan algunas palabras de la consulta.

Con un poco de investigación sobre las funciones de C# y los SRI (Sistemas de Recuperación de Información) me dispuse a materializar cada una de estas ideas.

Con respecto a los SRI y la clasificación de datos

Dentro de los Sistemas de Recuperación de Información existe un modelo llamado *Modelo Vectorial*, y estos se basan en transformación de cierta información en vectores. La idea básica es que un conjunto de documentos se puede definir como un espacio vectorial, **E** donde **dim(E)** es igual al total de palabras distintas que aparecen en los documentos. De esta forma cada documento se puede representar como un vector de **E** de la siguiente forma:

"El perro es parte del hombre."

"Mi gato y mi perro se pelean mucho."

el	perro	es	parte	del	hombre	mi	se	gato	pelean	mucho
1	1	1	1	1	1	0	0	0	0	0
0	1	0	0	0	0	2	1	1	1	1

En la representación anterior se puede apreciar que en cada componente correspondiente a una palabra se le asocia su frecuencia en el documento. De esta manera el segundo texto se puede representar en un *array* de int de la forma:

{0, 1, 0, 0, 0, 0, 2, 1, 1, 1, 1}

Con esto en mente es posible hacer gran parte del trabajo.

Lectura y normalización de los documentos

En este apartado creé una clase llamada Documents que contiene varias variables globales con el objetivo de cargar los datos una sola vez a la hora de ejecutar el proyecto.

Estas son las variables:

- `string [] directories`

Contiene las direcciones de los archivos a analizar.

- `Dictionary <string, List<int>> [] index`

Contiene en cada posición un Dictionary dónde las llaves son las palabras que aparecen en el documento y su valor asociado es una List con sus posiciones.

- `string [] vocabulary`

Contiene el universo de las palabras que aparecen en los documentos.

- `string [] tittle`

Contiene los nombres de los documentos.

- `string [][] documents`

En cada posición contiene el documento correspondiente como un *array* de las palabras del mismo.

- `string [][] docLines`

En cada posición contiene el documento dividido por líneas de texto.

- `int [][] vectors`

En cada posición contiene un *array* de int con la frecuencia de cada palabra del documento y sus índices coinciden con las palabras de vocabulary.

- `float [][] pesos`

En cada posición contiene un *array* de float con el valor del peso de la palabra del documento y sus índices coinciden con las palabras de vocabulary.

También consta de varios métodos:

- `static string Directories()`

Devuelve la ruta de la carpeta Content, donde están almacenados los documentos.

- `static void ProcessDirectories()`

Este es un método void que inicializa la variable directories utilizando la clase DirectoryInfo de C# y su método GetFiles.

- `static int Counter()`

Llama al método ProcessDirectories y devuelve la cantidad de elementos de directories.

- `static string Reader(string s, int pos)`

Devuelve recibe la ruta de un archivo y el índice del mismo. Asigna a docLines el valor string correspondiente utilizando la función ReadAllLines de la clase File y devuelve un string con el texto íntegro.

- `static string [] Execute(string s)`

Recibe un documento en forma de string. Pone todas las letras en minúsculas con la función ToLower. Luego donde hay caracteres extraños como "!,@,#,$,%,..." separa

el string por la función Split. Devuelve un *array* de string con las palabras normalizadas.

- `static string Accent(string s)`

Recibe un string y remplaza las tildes por las letras correspondientes.

- `static void Tittle()`

Éste método inicializa la variable tittle tomando el nombre de cada archivo en la variable directories sin su extensión con la función GetFileNameWithoutExtension.

- `static void Make()`

Realiza la mayor parte de la inicialización de las variables de Documents. Primero ejecuta el método ProcessDirectories. Como variable auxiliar crea el Diccionary<string, int> vocabularioDic dónde la llave es cada palabra sin repetirse entre todos los documentos y el valor asociado es la cantidad de documentos en los que aparece dicha palabra. También crea un *array* de Diccionary<string, int> vecTemp donde la llave de cada diccionario se corresponden con la de vocabularioDic y su valor asociado es la cantidad de veces que aparece en el documento correspondiente al índice del *array*. Luego, en un ciclo for, por cada documento asigna a la variable documents su valor correspondiente y crea una Diccionary<string, int> temp, donde iremos guardando las palabras que encontremos en documents con el objetivo de contar en vocabularioDic cada palabra una vez por documento y a al mismo tiempo poder contar las veces que se repite una palabra por documento. Luego por cada palabra de documents en la posición actual, si esta no está contenida en vocabularioDic la agrega con frecuencia 0, y si esta ya aparece en vocabularioDic evalúa dos condiciones:

*Si esta no aparece en vecTemp, la agrega con frecuencia 0 y la agrega index.

*Si esta no aparece en temp aumenta su valor en vocabularioDic en 1 y la agrega a temp.

Y luego si esta palabra aparece en vecTemp aumenta su frecuencia en 1 y agrega su índice a index.

Después se le asignan a vocabulario las llaves de vocabularioDic y a repeticiones sus valores asociados. Luego por cada documento se le asigna a vectores los valores de vecTemp correspondientes con las posiciones de vocabulario a la vez que se asignan

los valores de pesos por medio del método Peso (del cual hablaré más adelante) de la clase Moogle. Por último ejecuta el método Tittle.

Esta clase contiene otros métodos de los cuales hablaré en su apartado correspondiente.

Clasificación de los datos

Una vez terminada la transformación en vectores de los datos analizados, es necesaria su clasificación. Para esto el *modelo vectorial* consta de los términos **tf** (Term Frecuency) e **idf** (Inverse Document Frecuency).

$$\boldsymbol{tf} = \frac{\boldsymbol{nFrec}}{\mathbf{nFrec} * \boldsymbol{max}}$$

Donde **nFrec** es la cantidad de repeticiones en el documento y **max** el total de palabras del documento.

$$\boldsymbol{idf} = \log \frac{\boldsymbol{nDocs}}{\boldsymbol{doc}}$$

Donde **nDocs** es la cantidad de documentos en los que aparece y **docs** el total de documentos.

El resultado de la puntuación de la palabra está definido por:

$$\boldsymbol{peso} = (\boldsymbol{a} + (\mathbf{1} - \boldsymbol{a}) * \boldsymbol{tf}) * \boldsymbol{idf}$$

Donde **a** es un valor entre 0 y 1 para "aligerar" la diferencia de peso de los términos más frecuentes.

Para esta tarea implementé varios métodos dentro de la clase Moogle. Estos son los que hacen la "magia" y casi todo el proceso algorítmico.

- `static float Ponderar(int nFrec, int max, int nDocs)`

Devuelve un float los valores de peso de las palabras atendiendo a su tf e idf y.

- `static float[] Peso(string [] list, int [] v, int [] rep)`

Recibe un string [] con las palabras del documento, un int [] con las frecuencias de las palabras en el documento y un int [] con cantidad de documentos en los que aparece cada palabra. Luego por cada palabra del documento calcula su peso mediante el método Ponderar mencionado anteriormente.

Normalización y clasificación de la consulta y ejecución de la búsqueda

Una vez clasificada la información es necesario utilizar estos datos para comparar que tan relevantes son para la búsqueda. De esta forma podemos definir el **score** de un documento de la siguiente forma.

$$score_j = \sum_{i=0}^{n} \frac{PQ_i * PW_{ij}}{\sqrt{|PQ_i * PW_{ij}|}}$$

Donde **n** = **dim(E)**, **PQ** son los pesos de los términos consulta y **PW** son los pesos de los términos de los documentos.

Como parte de la búsqueda existe implementado el método Query de la clase Moogle que recibe un string query y devuelve un SearchResult.

Para este apartado creé la clase Search cuyo método principal es Result que recibe un string query y devuelve un SearchItem []. Otros métodos de esta clase son:

- `static string ExecuteQuery(string s)`

Éste método es parecido en funcionalidad al método Execute de la clase Documents, pero no normaliza los caracteres correspondientes a los operadores de interés.

- `static int [] VectorQuery(string [] list)`

Recibe un string [] con las palabras de la query y devuelve un int [] con su frecuencia en la misma.

- `static SearchItem[] Ordenar(float [] sc, string [] tt, string [] snp)`

Recibe los score, tittle y snipett resultados de la búsqueda. Realiza la ordenación por burbuja apoyándose de la variable bool cambios, donde si se realiza un cambio de posición de los elementos, se vuelve a ejecutar sobre su resultado de manera recursiva.

- `static SearchItem[] Redimencionar(SearchItem [] respuesta)`

Elimina los resultados donde el score sea menor que un valor acotado (en este caso en 0.5f) hasta un máximo de 15 resultados de forma tal que se ignoren los resultados que no se relacionan con la consulta. Devuelve el nuevo resultado filtrado.

- `public static SearchItem[] Result(string query)`

Ejecuta el método ExecuteQuery asignando su valor a la variable string [] enter y normaliza la consulta.

Crea las tuplas:

- `(int, string, string) [] VecOp`
- `(string, int) [] rel`

Itera por los elementos de enter y realiza las siguientes acciones:

-Si encuentra el operador "*", asigna a rel en Item1 la palabra actual y en Item2 la cantidad de veces que aparece este operador en la palabra.

-Si encuentra los operadores "!", "^", asigna a vecOp en Item1 2 ó 3 respectivamente, en Item2 la palabra actual y en Item3 el string vacío.

-Si encuentra el operador "~", asigna a vecOp en Item1 1, en Item2 la palabra de la posición anterior y en Item3 la palabra de la posición siguiente.

-Si la palabra en cuestión no aparece en el universo de palabras, asigna a enter el resultado devuelto por el método Suggestion de la clase Moogle (de la cual hablaré más tarde).

Asigna a string sugerencia de la clase Moogle el valor de string.Join(" ", enter).

Luego calcula el peso de las palabras de la consulta mediante el método Peso de la clase Moogle y lo asigna a la variable global queryRel de la misma clase. Después calcula los valores de score de los documentos con uso del método Relevance de la clase Moogle. Con respecto a éste método:

```
static float Relevance(float [] consulta, float [] doc)
```

-Recibe los pesos de la consulta y los pesos de las palabras por documento.

-Devuelve un float score con el score asociado.

Luego de esto aplica los operadores sobre los scores mediante el método Containtment de la clase Operator.

Se ordenan los elementos de la query según su score mediante los métodos:

- `static void Order(string [] enter)`

Relaciona cada palabra de la query con su score y las ordena con Ordenation.

- `static void Ordenation(string [] enter, float [] enterF)`

Ordena los elementos de enter en relación con enterF.

Se asignan los snippets solo a los documentos que contengan un contenido relacionado con la búsqueda con un score mayor que el acotado (en este caso es 0.5f, el mismo que el del método Redimencionar). Para esto creé un método el método SnippetLine en la clase Documents.

```
static string SnippetLine(int i, string [] enter)
```

Este método recibe el índice correspondiente al documento las palabras de la consulta. Luego busca por orden de relevancia si la palabra aparece en el diccionario que contiene sus índices. Luego busca la primera línea del documento en la que aparezca la palabra de mayor relevancia almacenada en docLines y la devuelve.

Ya con los valores de los snippets ordena los resultados con el método Ordenar. Después elimina los resultados que no son de interés con el método Redimencionar. Luego de esto devuelve los resultados de la búsqueda.

Implementación de los operadores

Los clasifiqué en dos grupos:

*Los de contención:

-Operador de inclusión "^".

-Operador de exclusión "!".

-Operador de cercanía "~" (éste lo clasifiqué así porque para aplicarse es necesario que ambas palabras aparezcan en el documento).

El de relevancia "".

Los de contención:

```
static void Containment(float score, string s, int n, string t)
```

Para estos creé el método Containment de la clase Operator que recibe los scores de los documentos, la palabra sobre la que se aplica el operador, un int n entre 1 y 3 que identifica el operador y un string t que es una palabra sobre la que se utiliza el operador "~".

*Para n = 2 ("!"):

-Por cada documento, si la palabra s aparece su score es 0.

*Para n = 3 ("^"):

-Por cada documento, si la palabra s no aparece su score es 0.

*Para n = 1 (“~”):

Por cada documento, crea int min con valor inicial el máximo entero posible. Si index contiene la llave s y la llave t, haya la menor distancia entre sus índices y se la asigna a min (cabe resaltar que si min = 1 termina el ciclo). Luego el score resultante es el score actual más 1/min.

El de importancia:

```
static void Importance(string[]enter,float[]queryRel,(string,int)[] rel)
```

Modifica el score correspondiente a la palabra con el operador, multiplicando el valor de rel.Item2 asociado por 2 por su score original.

Implementación de la sugerencia

Para este apartado utilicé la distancia de Levenshtein la cual permite establecer una comparación entre dos palabras aplicando las siguientes operaciones:

- Sustitución: Sustituir una letra por otra.
- Inserción: Introducir una letra.
- Eliminación: Eliminar una letra.

De modo que si se quiere saber la diferencia entre “gatos” y “gate” sería de la siguiente forma:

		g	a	t	o	S
	0	1	2	3	4	5
g	1	0	1	2	3	4
a	2	1	0	1	2	3
t	3	2	1	0	1	2
e	4	3	2	1	1	2

Los elementos en naranja son los cambios realizados y el elemento final de la matriz es la distancia entre las palabras.

```
static int Comparer(string s, string t)
```

Éste método crea una matriz de dimensión [s.Length + 1, t.Length + 1]

Recorre la matriz por los elementos **ij**, compara las palabras por los índices, y si son distintos los caracteres, asigna 1 a los cambios. Luego halla el mínimo entre los elementos (i,j+1), (i+1,j) y el valor de la diagonal + el de los cambios. Devuelve el valor del final de la matriz correspondiente a la distancia.

```
static string Suggestion(string s)
```

Con ayuda de Comparer busca la palabra con menor distancia y la devuelve como sugerencia. Si la distancia mínima es igual a la longitud de la palabra original, devuelve la misma palabra.