Primer Proyecto de Programación Moogle

Olivia Ortiz Arboláez
Julio de 2023

Para hacer este proyecto, requerí un proceso de investigación sobre distintas utilidades además de una planificación y puesta en práctica de las mismas tal que se adaptasen al objetivo de hacer un código en C# para un buscador.

1 Inicio

1.1 Cargar los Documentos

Claro, para que el buscador funcione tiene que leer los archivos que estén dentro de una carpeta. Ese sería el primer paso. En la clase Program del proyecto se crea un objeto **DB** tipo Database, que es una clase que posee varias propiedades. Dichas propiedades servirán para trabajar con los archivos que compondrán la base de datos del buscador.

```
using System ;
using System.IO ;

public class Database
{
    public Document[] docs ;
}
```

Como se puede ver en este código, la clase Database posee un array tipo Document llamado docs.
T odos los objetos tipo Document tienen la propiedad ruta en dicha clase. Por tanto el array contendrá constancia de dicha propiedad en cada elemento que posea.

Ahora bien, es momento de cargar los documentos. Para ello fue empleada la siguiente función:

```
     private void CargarDocs(string folder)
    {
        IEnumerable <string> Verarchivos = Directory.EnumerateFiles (folder , "*.
            txt" , SearchOption.AllDirectories) ;
        this.docs = new Document[Verarchivos.Count()] ;

        Parallel.For(0 , docs.Length , i =>
        {
            this.docs[i] = new Document(Verarchivos.ElementAt(i)) ;
        });

    }
```

Esta función no hace más que leer los archivos .txt que se encuentran en la carpeta que se le es pasada por parámetro y, por cada archivo que encuentra la función EnumerateFiles crea un nuevo objeto tipo documento que es enviado hacia el array docs mediante un bucle.

Quedando como elementos de docs[] los archivos .txt que están en la carpeta[1]

1.2 Almacenar datos

Teniendo ya constancia de cada documento que será empleado para la futura búsqueda, estamos en disposición de guardar las palabras contenidas en cada uno de ellos. Para esto transité por un postergado proceso de análisis acerca de cuál sería la forma más conveniente para dicha tarea.[2]
En este caso:

```
public Dictionary <string , Dictionary <int , int> > \textcolor{yellow}{
     contenido} = new Dictionary <string , Dictionary <int , int> > () ;
```

Siendo este diccionario contenido una propiedad de Database y un diccionario de diccionarios, puesto que tendría por cada palabra que se le asigne, un diccionario de enteros asociado a esta. Este diccionario de enteros subordinado busca contener como Key documentos donde se encuentra la palabra y como Value la cantidad de veces que se halla en el documento actual.
Ejemplo: (Cactus, (1 , 4))

Pues, para que la búsqueda sea más eficiente, es necesario estandarizar las palabras que están en estos documentos, quitando problemáticas futuras como diferencias entre letras mayúsculas, minúsculas y acentos que pueden ser a veces simples errores humanos de escritura. Para esto empleé una función llamada Estandarizador en la clase Xtra. Esta clase Xtra posee métodos variados de utilidad que serán llamados conforme vayan siendo necesitados.

A continuación la función FillDict.

```
private void FillDict ()
 {
    for (int i = 0; i < docs.Length; i++)
    {
        string[] text = Xtra.Estandarizador( File.ReadAllText( docs[i].ruta)).
             Split() ;

        foreach (string word in text )
        {
            if ( ! this.\textcolor{yellow}{contenido}.ContainsKey(word) )
            {
                this.\textcolor{yellow}{contenido}.Add( word , new Dictionary <
                     int , int> () ) ;
            }
```

[1] La carpeta de ejemplo empleada tenia de nombre Content, pero para generalizar se utiliza folder.

[2] Varias ideas cruzaron por mi mente: la creación de un array para que fuesen almacenadas las palabras, una lista o lo empleado finalmente para este propósito,un diccionario.

```
            if (! \textcolor{yellow}{contenido}[word].ContainsKey(i) )
            {
                \textcolor{yellow}{contenido}[word].Add(i , 0);
            }

            \textcolor{yellow}{contenido}[word][i]++ ;
        }
    }
}
```

La misma lee las palabras que hay en los documentos según la ruta, separa estas palabras, las estandariza y las coloca en un array tipo string. Luego toma estas palabras y las añade al diccionario contenido si no están y el documento en que se encuentra.

Cumpliéndose así nuestro primer objetivo de cargar todos los archivos y almacenar las palabras para trabajar, avancemos.

2 Trabajo con vectores

2.1 Hallar TF-IDF a contenido.

El TF-IDF (***en inglés.*** *Term Frequency - Inverse Document Frequency*) sirve para evaluar la importancia de un término en un documento dentro de un conjunto de documentos. Lo esencial detrás del TF-IDF es que las palabras que ocurren con frecuencia en un documento, pero raramente en otros documentos, adquieren mayor importancia y son distintivas de ese documento en particular.

Para el procesamiento de los datos guardados en contenido nos hará falta tener una noción de cada palabra con su peso en la base de datos. Esto conllevó a la creación de una nueva clase, Vector. Que cuenta con otro diccionario que ha de tener las palabras y su TF*IDF calculado.

```
using System ;

public class Vector
{
// Propiedades
public Dictionary <string , double> peso = new Dictionary <string ,double> ();

}
```

Bien, para calcular el **TF** (*Frecuencia de Término*) si bien la fórmula típica es:

$$TF = \frac{\#\ de\ veces\ que\ el\ término\ aparece\ en\ el\ documento}{\#\ total\ de\ términos\ en\ el\ documento} \quad (1)$$

Debido a que al dividir estos valores obtenemos números muy pequeños con los que trabajar posteriormente si usamos esta ecuación, fue necesario utilizar

una modificación de dicha fórmula para el Proyecto. Siendo la fórmula en realidad empleada para obtener el TF la siguiente:

$$TF = \frac{\#\ de\ veces\ que\ el\ término\ aparece\ en\ el\ documento}{\#\ de\ veces\ está\ la\ palabra\ más\ repetida\ en\ el\ documento} \tag{2}$$

Para obtener la cantidad de veces que estaba la palabra más repetida en cada documento, se implementó la función MaxFrecuency, y una propiedad maxfrec en la clase Document para guardarla. Siendo MaxFrecuency:

```
private void MaxFrecuency(Dictionary <string , Dictionary <int , int> > \
    textcolor{yellow}{contenido})
{
    foreach (string key in \textcolor{yellow}{contenido}.Keys)
    {
        foreach (int indice in \textcolor{yellow}{contenido}[key].Keys)
        {
            if (docs[indice].maxfrec < \textcolor{yellow}{contenido}[key][
                indice])
            {
                docs[indice].maxfrec = \textcolor{yellow}{contenido}[key][
                    indice] ;
            }

        }
    }
}
```

Esta es solo una función que calcula las veces que se repite la palabra más común en el documento, no obstante con la misma podemos calcular el TF una vez accedamos a las valores del diccionario asignado como Value al diccionario contenido[3].

Para el **IDF** (*Frecuencia Inversa en el Documento*) si se utilizó su fórmula general:

$$IDF = \log \frac{cantidad\ total\ de\ documentos}{cantidad\ de\ documentos\ donde\ está\ la\ palabra} \tag{3}$$

Esta fórmula resulta sencilla de hallar al usar la propiedad .Length en el array docs y el método .Count() dentro del cada palabra en contenido. Una vez con todo esto precisado, es momento de calcular.

```
    public void TFIDF (Dictionary<string , Dictionary<int , int>> \textcolor{
        yellow}{contenido})
{
    MaxFrecuency(\textcolor{yellow}{contenido}) ;
    foreach( string word in \textcolor{yellow}{contenido}.Keys)
    {
        //formula IDF
        double idf = Math.Log10( (docs.Length +1.0) / (\textcolor{yellow}{
            contenido}[word].Count() + 1.0 ));
        foreach (int num in \textcolor{yellow}{contenido}[word].Keys)
```

[3]Un ejemplo: acceder a (Cactus , (1 , **4**) //donde 4 es el valor al que se busca acceder.)

```
        {
            //formula TF
            double tf = (double)\textcolor{yellow}{contenido}[word][num] / (
                docs[num].maxfrec+ 1.0) ;

            //Crea los vectores documentos con su TFIDF asociado.
            docs[num].Documentow[word] = tf * idf ;

        }
    }
}
```

De esta forma tenemos cada palabra que ya existía en el contenido guardada en el diccionario peso de la clase Vector con su TF-IDF calculado.

2.2 Hallar TF-IDF a la query

Es tiempo ahora de trabajar con la consulta del usuario, a la cual denominaremos query. Esta query ha de estar normalizada también para eliminar inconvenientes como los mencionados anteriormente al estandarizar los archivos en Database. Por lo que una vez más se emplea el método Estandarizador de la clase abstracta Xtra.

Ahora una vez estandarizadas las palabras, usaremos .Split() y las enviaremos a un array tipo String.

Aplicaremos el mismo procedimiento empleado con contenido para calcular el TF-IDF de la query, con el objetivo de crear un elemento quervector tipo Vector para asignar los valores del TF*IDF al diccionario peso correspondiente. Para calcular el **TF** de la query me fue necesario crear las siguientes funciones:

- Contar: Devuelve la cantidad de veces que está una palabra en el array:

```
public static int Contar (string [] arr , string word)
{
    int conteo = 0 ;

    for (int i = 0; i < arr.Length; i++)
    {
      if(arr[i] == word)
      {
        conteo ++ ;
      }
    }

    return conteo ;

}
```

- FrecMAX: Halla la mayor cantidad de repeticiones en el array :

```
public static int FrecMAX (string [] array)
{
```

```
    int Maximo = 0 ;
    int rept = 0 ;

     for (int i = 0; i < array.Length; i++)
     {
        for (int j = 0; j < array.Length; j++)
        {

         if(array[i] == array[j])
         {
            rept ++ ;
         }
        }
        Maximo = Math.Max(Maximo , rept) ;
        rept = 0 ;
     }

    return Maximo;

}
```

Consecutivamente se utiliza una función similar a cuando se trabajó con el diccionario contenido, pero esta vez utilizando las dos funciones implementadas en la clase Xtra para el TF. La siguiente función crea el Vector Query :

```
private void CrearVectorQuery (string [] query)
{
    foreach (string item in query)
    {

     if(\textcolor{yellow}{contenido}.ContainsKey(item))
     {
        double idf = Math.Log10((docs.Length) + 1.0) / (\textcolor{yellow}{
             contenido}[item].Count() + 1.0);
        double tf = Xtra.Contar(query , item) / (Xtra.FrecMAX(query) + 1.0) ;
        quervector[item] = tf * idf ;
     }


    }
}
```

2.3 Calcular Similitud del Coseno

Actualmente, contamos con el Vector Documentow y el Vector Query, que guardan en un diccionario la relación de cada palabra con su peso calculado. ¿Cómo podemos saber qué tan ***similares*** son estos vectores que tenemos? Aplicaremos para ello la fórmula de la Similitud del Coseno.

- La Similitud del Coseno devuelve un valor que se encuentra entre -1 y 1.
- Un resultado de -1 en este caso al calcularla, no es posible, puesto que trabajamos sólo con los casos en que esté o no esté la palabra.
- Mientras más cercano sea el resultado a 0, indica que son diferentes los vectores o no tienen relación alguna.

- Mientras más cercano sea el resultado a 1 de calcular la Similitud del Coseno, mayor será la similitud entre ambos.

Entonces, sea dicha fórmula la siguiente:

$$\frac{\sum_{\vec{\delta}*\vec{\nu}}}{|\vec{\delta}| * |\vec{\nu}|} \tag{4}$$

$$\vec{\delta} \quad vector \quad query \quad y \quad \vec{\nu} \quad vector \quad Documento \tag{5}$$

Lo presente en el numerador de esta fórmula es llamado producto escalar de ambos vectores y se obtiene multiplicando cada elemento del vector query por su correspondiente en el vector Documentow. [4]

$$\vec{\delta}_1 * \vec{\nu}_1 + \vec{\delta}_2 * \vec{\nu}_2 \tag{6}$$

Para calcular lo que se halla en el denominador de la fórmula, que es denominado ¨Normas de Vectores¨, en el Proyecto se utiliza un método llamado Norma de la clase Xtra. La fórmula general para hallar la norma es:

$$\sqrt{\vec{\nu}_1^2 + \vec{\nu}_2^2 + \vec{\nu}_3^2} \tag{7}$$

Teniendo todo esto en cuenta, se empleó el siguiente método SimilitudVect de la clase Vector :

```
 public static double SimilitudVect (Dictionary <string , double> Documentow
      , Dictionary <string , double> quervector)
{
    double Escalar = 0.0;
    foreach (string word in quervector.Keys)
    {
           if (Documentow.ContainsKey(word))
           {
               Escalar += quervector[word] * Documentow[word];
           }

    }
    return Escalar / (Vector.Norma(Documentow) * Vector.Norma(quervector)) ;
}
```

Este resultado se guarda en una propiedad Document llamada Score mediante un bucle.

3 Obtener Resultados

Dentro de la propiedad Score tenemos varios valores, conforme sean más cercanos a 1, el documento asociado al valor más alto será el que tenga mayores coincidencias respecto a la búsqueda, y así sucesivamente. Por lo cual hace falta

[4] Únicamente para recordar, sería multiplicar los valores en peso de ambos vectores entre ellos. Ejemplo: **Documentow[cactus] * quervector[cactus]**.

Figure 1: Barra de búsqueda

organizar estos valores de mayor a menor, y así poder llamarlos. Usando el algoritmo Selection Sort pero adaptado al docs [], que es un array tipo Document, se consiguió este propósito.

```
 public static void SelectionSort(Document [] array)
{
    int n = array.Length;
    for (int i = 0; i < n - 1 ; i++)
    {
     int Maxindice = i;
     for (int j = i + 1; j < n; j++)
     {
      if(array[j].Score > array[Maxindice].Score)
      {
        Maxindice = j;
      }
     }
     Document temporal = array[Maxindice] ;
     array[Maxindice] = array[i] ;
     array[i] = temporal ;
    }
}
```

Ahora con los resultados de mayor valor ordenados, se emplea un bucle para devolver los primeros. Una vez hecho esto el código está listo para ser ejecutado, Solamente ha de realizar una consulta en la barra de búsqueda.