Proyecto de Programación Moogle

Olivia Ortiz Arboláez Julio de 2023

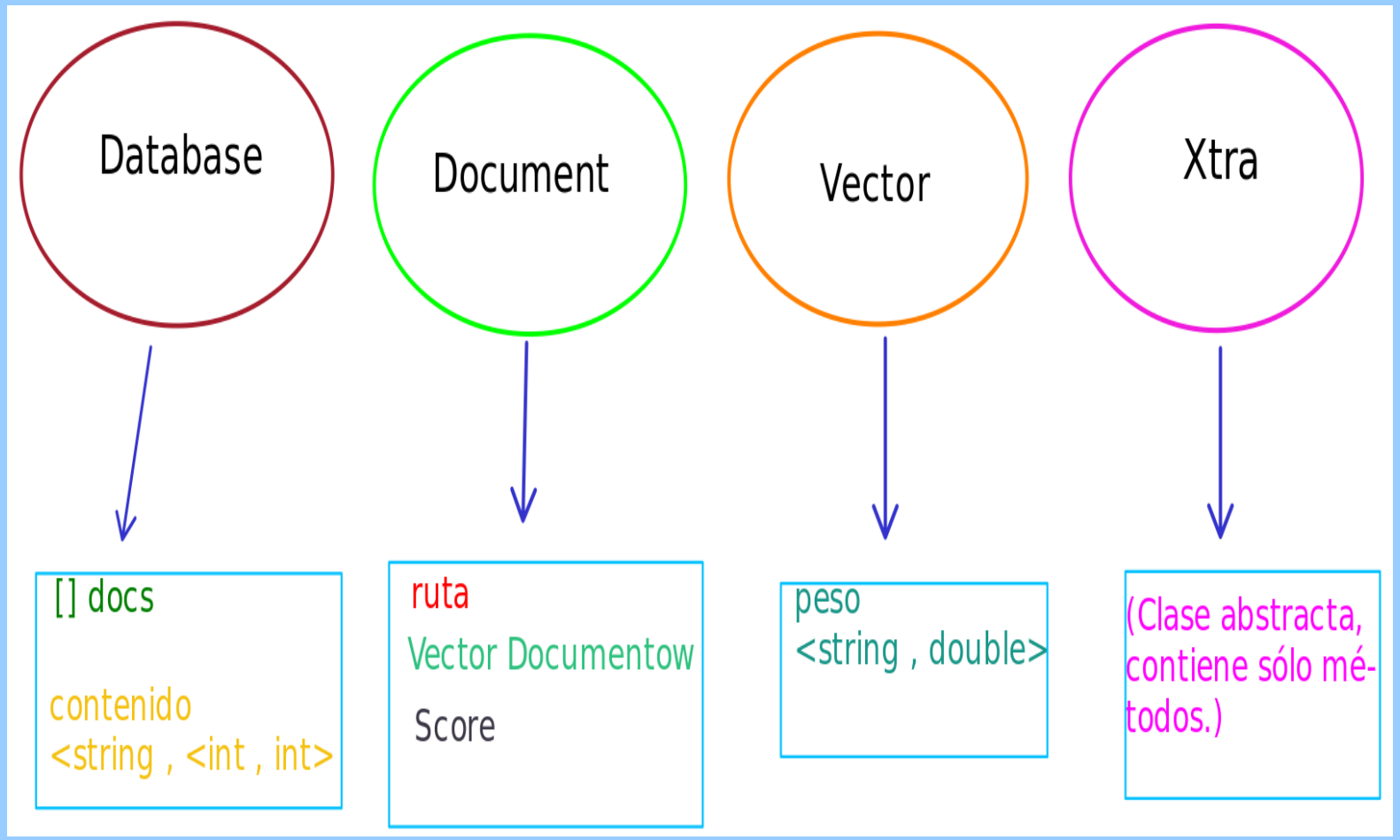
Moogle! Es un proyecto que busca implementar un motor de búsqueda de palabras de archivos .txt. Para esto fue requerido un proceso de investigación sobre ciertas utilidades, fórmulas y la correcta puesta en práctica de estas para conseguir hacer un código en C# que cumpla este objetivo.

Dicho esto, comencemos.

Inicio

En este proyecto fue necesaria la creación de varias clases para tener un código más organizado y entendible a la hora de idearlo. Estas clases se encuentran dentro de MoogleEngine y son:

- Database
- Document
- Vector
- Xtra


Database
Document
Vector
Xtra
[] docs
contenido
<string , <int , int>
ruta
Vector Documentow
Score
peso
<string , double>
(Clase abstracta, contiene sólo métodos.)

Nuestro programa comienza principalmente en Database, creando un nuevo array tipo Document. Todos 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:

small

```
        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 con su ruta almacenada.

Ahora procedemos a llenar este diccionario:

small

```
public Dictionary<string, Dictionary<
    int, int>> contenido = new Dictionary
    <string, Dictionary<int, int>>()
    ;
```

Emplearemos la siguiente función FillDict para leer las palabras en las rutas de los documentos, estandarizarlas y organizarlas. Para Estandarizar las palabras usamos una función que está ya implementada en la clase Xtra.

small

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

small

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

small

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

                        contenido[word][i]++ ;
                    }
                }
            }
```

Trabajo con Vectores

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 obtener estas propiedades trabajaremos con el diccionario peso de la clase Vector.

small

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

}
```

Las fórmulas que emplearemos para calcular el TF y el IDF son las siguientes:

$$TF = \frac{\#\ de\ veces\ que\ el\ término\ aparece\ en\ el\ documento}{\#\ de\ veces\ está\ la\ palabra\ más\ repetida\ en\ el\ documento} \tag{1}$$

$$IDF = \log \frac{cantidad\ total\ de\ documentos}{cantidad\ de\ documentos\ donde\ está\ la\ palabra} \tag{2}$$

Mediante la función siguiente tendremos ya cada palabra previamente existente en contenido almacenada en un diccionario peso:

small

```
        public void TFIDF (Dictionary<string ,
            Dictionary<int , int>> \textcolor{
            yellow}{contenido})
    {
        MaxFrecuency(\textcolor{yellow}{
            contenido}) ;
        foreach( string word in \textcolor{
            yellow}{contenido}.Keys)
        {
```

small

```
            //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)
            {
             //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  ;
```

A continuación de esto se hace un procedimiento similar pero con las palabras de la búsqueda del usuario. Se estandarizan estas palabras, se separan y se envían a un array que se le es enviado a una función que halla el TF*IDF de la query. Para crear este vector query se usan dos funciones más de la clase Xtra, Contar y FrecMAX.

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:

$$\frac{\sum_{\vec{\delta} * \vec{\nu}}}{|\vec{\delta}| * |\vec{\nu}|} \tag{3}$$

$$\vec{\delta} \quad \textit{vector} \quad \textit{query} \quad \textit{y} \quad \vec{\nu} \quad \textit{vector} \quad \textit{Documento} \tag{4}$$

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.

$$\vec{\delta}_1 * \vec{\nu}_1 + \vec{\delta}_2 * \vec{\nu}_2 \tag{5}$$

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:

$$\sqrt{\vec{\nu}_1^2 + \vec{\nu}_2^2 + \vec{\nu}_3^2} \tag{6}$$

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

small

```
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];
            }
```

small

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

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

Devolver 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 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.

small

```
    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++)
       {
```

small

```
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.

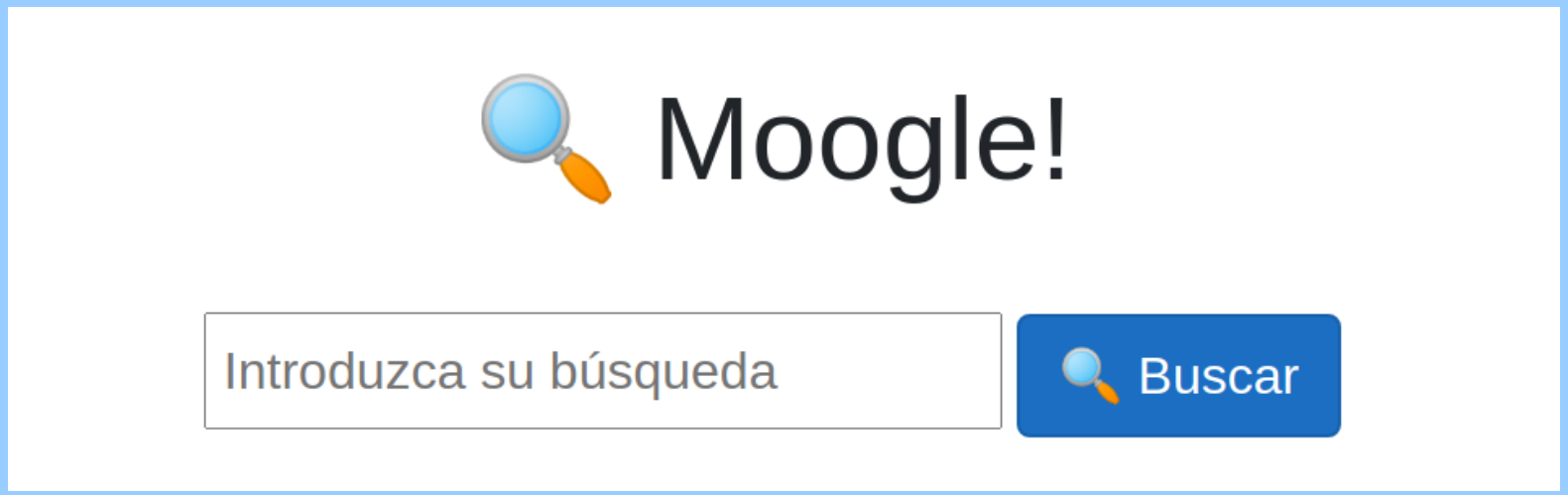


Figure: Barra de búsqueda