Moogle!

> Proyecto de Programación I. Facultad de Matemática y Computación - Universidad de La Habana. Cursos 2021, 2022.

Moogle! es una aplicación *totalmente original* cuyo propósito es buscar inteligentemente un texto en un conjunto de documentos.

Es una aplicación web, desarrollada con tecnología .NET Core 6.0, específicamente usando Blazor como *framework* web para la interfaz gráfica, y en el lenguaje C#. La aplicación está dividida en dos componentes fundamentales:

- `MoogleServer` es un servidor web que renderiza la interfaz gráfica y sirve los resultados.
- `MoogleEngine` es una biblioteca de clases donde está implementada la lógica del algoritmo de búsqueda.

Sobre la búsqueda

La búsqueda pretende ser lo más inteligente posible, por ese motivo no nos limitamos a los documentos donde aparece exactamente la frase introducida por el usuario. - El usuario puede buscar no solo una palabra sino en general una frase cualquiera. - Si no aparecen todas las palabras de la frase en un documento, pero al menos aparecen algunas, este documento también es devuelto, pero con un `score` menor mientras menos palabras aparezcan. - El orden en que aparezcan en el documento los términos del `query` en general no se tiene en cuenta, ni siquiera que aparezcan en lugares totalmente diferentes del documento. - Si en diferentes documentos aparecen la misma cantidad de palabras de la consulta, (por ejemplo, 2 de las 3 palabras de la consulta `"algoritmos de ordenación"`), pero uno de ellos contiene una palabra más rara (por ejemplo, `"ordenación"` es más rara que `"algoritmos"` porque aparece en menos documentos), el documento con palabras más raras tiene un `score` más alto, porque es una respuesta más específica. - De la misma forma, si un documento tiene más términos de la consulta que otro, en general tiene un `score` más alto (a menos que sean términos menos relevantes). - Algunas palabras excesivamente comunes como las preposiciones, conjunciones, etc., son ignoradas por completo ya que aparecerán en la inmensa mayoría de los documentos (esto se hace de forma automática, o sea, que no hay una lista cableada de palabras a ignorar, sino que se computan de los documentos).

Evaluación del `score`

De manera general el valor de `score` corresponde a cuán relevante es el documento devuelto para la búsqueda realizada.

Si un documento no tiene ningún término de la consulta, y no es para nada relevante, entonces su `score` es `0.008` como mínimo (se ha determinado experimentalmente este valor) pero no debe haber ningún error o excepción en estos

casos. El score es un float entre `0` y `1` dado por el coseno entre el vector que representa la query y el que representa el documento dado.

Algoritmos de búsqueda

Nuestro algoritmo de búsqueda se basa en el `Sistema Vectorial de Recuperacion de Informacion` ,en principio es vectorizar los documentos y la consulta (normalizados) y , a partir de el coseno que forman la query y un documento (vectorizados) retornar un score, para esto usamos la clase `Vector` :

```
public class Vector
    {
        //Intstance Variables
        private double[] components;
        private int dimension;
        private double module;
        //Properties
        private double Module
        {
            get { return module; }
        }
        //Methods
        public Vector(double[] components) //Constructor
        {
            this.components = components;
            this.dimension = this.components.Length;
            this.module = GetModule();
        }
        private double GetModule()
        {
            double Sum = 0;
            foreach (var component in this.components)
            {
                Sum += (float)Math.Pow(component, 2);
            }
            return Math.Sqrt(Sum);
        }
        private static double DotProduct(Vector v1, Vector v2) // returns the dot product b
        {
            double res = 0;
            for (int i = 0; i < v1.dimension; i++)
            {
                res += v1.components[i] * v2.components[i];
            }
            return res;
        }
```

```
        public static double Cos(Vector v1, Vector v2) //Retrns the cosine between to vecto
        {
            if (v1.Module != 0 && v2.Module != 0)
            {
                return DotProduct(v1, v2) / (v1.Module * v2.Module);
            }
            else
            {
                return 0;
            }
        }
    }
```

En la clase `Moogle.cs` se vectorizan los textos y se devuelven los resultados de la consulta en orden descendente.

```
public static class Moogle
{
    private static Documents documents = new Documents(@"C:\Users\elsingon\Downloads\moogle-
    public static SearchResult Query(string query)
    {
        Query userInp = new Query(query, documents.WordSet, documents.IDF);
        double[] scores = GetScores(userInp);
        double[] positiveScores = scores.Where(x => x > 0).ToArray();
        int PosScoresLength = positiveScores.Length;
        string[] results = GetResults(scores);

        Array.Sort(scores, results);
        Array.Reverse(results);
        Array.Reverse(scores);
        SearchItem[] items = new SearchItem[PosScoresLength];
        for (int i = 0; i < PosScoresLength; i++)
        {
            items[i] = new SearchItem(Path.GetFileName(results[i]), ExtractText(results[i],c
        }
        return new SearchResult(items, query);

    }
    public static double[] GetScores(Query userInp)
    {
        double[] result = new double[documents.TF_IDF.GetLength(1)];
        Vector queryVector = new Vector(userInp.TF_IDF);
        for (int i = 0; i < documents.TF_IDF.GetLength(1); i++)
        {
            double[] doc = new double[documents.TF_IDF.GetLength(0)];
            for (int j = 0; j < documents.TF_IDF.GetLength(0); j++)
```

```
        {
            doc[j] = documents.TF_IDF[j, i];
        }
        Vector docVecor = new Vector(doc);
        result[i] = Vector.Cos(queryVector, docVecor);
    }
    return result;
}
public static string[] GetResults(double[] scores)
{
    List<string> results = new List<string>();
    for (int i = 0; i < documents.TF_IDF.GetLength(1); i++)
    {
        results.Add(documents.DocTF.ElementAt(i).Key);
    }
    return results.ToArray();
}
public static string ExtractText(string filePath, string query)
{
    string text = File.ReadAllText(filePath);
    string[] words = query.Split(' ');
    int start = -1;
    int end = -1;
    foreach (string word in words)
    {
        string pattern = $"(?i){Regex.Escape(word)}";
        Match match = Regex.Match(text, pattern);
        if (match.Success)
        {
            int index = match.Index;
            if (start == -1 || index < start)
            {
                start = Math.Max(0, index - 50);
            }
            if (end == -1 || index + match.Length > end)
            {
                end = Math.Min(text.Length - 1, index + match.Length + 50);
            }
        }
    }
    if (start == -1 || end == -1)
    {
        return "No se encontró ninguna coincidencia.";
    }
    return text.Substring(start, end - start);
}
```

```
}
```

Sobre el contenido a buscar

La idea original del proyecto es buscar en un conjunto de archivos de texto (con extensión `.txt`) que estén en la carpeta `Content`.

Ejecutando el proyecto

Lo primero que tendrás que hacer para poder trabajar en este proyecto es instalar .NET Core 6.0 . Luego, solo te debes parar en la carpeta del proyecto y ejecutar en la terminal de Linux:

```
make dev
```

Si estás en Windows, debes poder hacer lo mismo desde la terminal del WSL (Windows Subsystem for Linux). Si no tienes WSL ni posibilidad de instalarlo, deberías considerar seriamente instalar Linux, pero si de todas formas te empeñas en desarrollar el proyecto en Windows, el comando *ultimate* para ejecutar la aplicación es (desde la carpeta raíz del proyecto):

```
dotnet watch run --project MoogleServer
```

Cristhian Delgado Garcia C111