PROYECTO DE PROGRAMACION I

Rocio Serrano Laguna C-113

MOOGLE!

CIENCIA DE LA COMPUTACION

Facultad de Matemática y Computación

El presente proyecto consiste en desarrollar un motor de búsqueda de texto en ficheros “txt” para la aplicación web Moogle. Esta aplicación totalmente original tiene el propósito de buscar inteligentemente un texto en un conjunto de documentos. Está desarrollada con tecnología .Net Core 6.0, específicamente usando Blazor como framework web para la interfaz gráfica, y en el lenguaje C#. Además, está dividida en dos componentes fundamentales:

- MoogleServer: Es un servidor web que renderiza la interfaz grafica y sirve los resultados.
- MoogleEngine: Es una biblioteca de clases donde está…ehem…casi implementada la lógica del algoritmo de búsqueda.

Solución aplicada

Para la integración del proyecto con Moogle se realizó la modificación del método “Moogle.Query” que esta en la clase “Moogle” del proyecto “MoogleEngine”. Este método devuelve un objeto de tipo “SearchResult”. Este objeto contiene los resultados de la búsqueda realizada por el usuario, que viene en un parámetro de tipo “string” llamado “query”. El método “Moogle.Query” quedó de la siguiente manera:

```cs
public static class Moogle
{
    public static SearchResult Query(string query) {
        TFIDF tfidf = new TFIDF("../Content", query);
        tfidf.computeTFIDF();
        Dictionary<string, double> result = tfidf.computeCOSSIM();
        string[][] results = tfidf.normalizeResult(result);

        SearchItem[] items = new SearchItem[results.Length];
        for(int i = 0; i < results.Length; i++) {
            items[i] = new SearchItem(results[i][0],
results[i][1], float.Parse(results[i][2]));
        }

        return new SearchResult(items, query);
    }
}
```

La implementación del proyecto consiste en una clase llamada TFIDF la cual contiene todos los métodos y algoritmos necesarios para realizar las operaciones de búsquedas.

En el método “Moogle.Query” lo primero que se hace es crear una instancia de la clase TFIDF invocando su constructor:

```cs
TFIDF tfidf = new TFIDF("../Content", query);
```

Donde:

- “string directory”: Ruta de la carpeta donde se almacenan los documentos “.txt” sobre los cuales se realizan las búsquedas.
- “string query”: Cadena de texto que representa la consulta introducida por el usuario.

Luego se invoca al método público “computeTFIDF()” de la clase “TFIDF” con el objetivo de realizar los cálculos necesarios:

```cs
tfidf.computeTFIDF();
```

Posteriormente, se procede a realizar el calculo de la similitud del coseno con el objetivo de encontrar los documentos más relevantes:

```cs
Dictionary<string, double> result = tfidf.computeCOSSIM();
```

Para finalizar, se realiza una normalización de los resultados para mostrarlos de forma ordenada:

```cs
string[][] results = tfidf.normalizeResult(result);
```

Una vez listos los resultados de la búsqueda solo queda poblar el array “SearchItem” con dichos resultados y devolver el objeto “SearchResult”:

```cs
SearchItem[] items = new SearchItem[results.Length];
for(int i = 0; i < results.Length; i++) {
    items[i] = new SearchItem(results[i][0], results[i][1],
float.Parse(results[i][2]));
}

return new SearchResult(items, query);
```

Método de búsqueda

Para realizar la búsqueda teniendo en cuenta los requerimientos planteados, se realizó un estudio de diferentes algoritmos de búsqueda. Como resultado se decidió utilizar un modelo algebraico conocido como modelo de espacio vectorial el cual representa documentos mediante el uso de vectores en un espacio lineal multidimensional. Dicho modelo se basa en el grado de similaridad de una consulta dada por el usuario con respecto a los documentos de la colección cuyos términos fueron ponderados, en este caso mediante “TF-IDF” (Frecuencia de Términos – Frecuencia Inversa de Documentos).

Para llevar a cabo el método de búsqueda seleccionado se implementó la clase TFIDF. Dicha clase está compuesta por 8 atributos y 9 métodos:

Parámetros:

1. Almacena la cadena de consulta. Es privado porque solo se accede dentro de la clase.

```cs
private string query;
```

2. Almacena el vector TF de la consulta. Es privado porque solo se accede dentro de la clase.

```cs
private Dictionary<string, double> vectorQuery = new
Dictionary<string, double>{};
```

3. Almacena el vector TF de todos los documentos. Es privado porque solo se accede dentro de la clase.

```cs
private Dictionary<string, Dictionary<string, int>> corpus = new
Dictionary<string, Dictionary<string, int>>{};
```

4. Almacena un vector con todos los términos.

```cs
private Dictionary<string, string[]> bagOfWords = new
Dictionary<string, string[]>{};
```

5. Almacena un vector con todas las oraciones o frases.

```cs
private Dictionary<string, string[]> bagOfSentences = new
Dictionary<string, string[]>{};
```

6. Almacena el vector TF.

```cs
public Dictionary<string, Dictionary<string, double>> tf = new
Dictionary<string, Dictionary<string, double>>{};
```

7. Almacena el vector IDF.

```cs
public Dictionary<string, double> idf = new Dictionary<string,
double>();
```

8. Almacena el vector TFIDF.

```cs
public Dictionary<string, Dictionary<string, double>> tfidf =
new Dictionary<string, Dictionary<string, double>>{};
```

Métodos:

1. Constructor de la clase. Se formatea el query y se cargan los ficheros fuentes de la búsqueda. También se inicializan las variables “query” y “corpus”.

```cs
public TFIDF(string directory, string query) {}
```

2. Formatea el criterio de búsqueda (query). Inicializa el valor del vectorQuery con el cálculo del TF del query.

```cs
private void formatQuery(string query) {}
```

3. Calcula el TF por cada documento. Se realiza una división entre la cantidad de apariciones de cada término y la cantidad de palabras del documento, con el objetivo de no priorizar a los documentos más grandes, teniendo en cuenta, por ejemplo, que:
   Un término con una ocurrencia de 10 en un documento de 800 términos es menos relevante que un término con una ocurrencia de 8 en un documento de 100 términos.

```cs
public Dictionary<string, Dictionary<string, double>>
computeTF() {}
```

4. Calcula el IDF. Se calcula a partir de log(número total de documentos/número de documentos en los que el término esté presente).

```cs
public Dictionary<string, double> computeIDF() {}
```

5. Calcula el TFIDF. Se calcula a partir de la multiplicación del vector TF y el vector IDF.

```cs
public Dictionary<string, Dictionary<string, double>>
computeTFIDF() {}
```

6. Calcula la similitud del coseno. Este método nos permite asignar un score a cada documento en relación al criterio de búsqueda.

```cs
public Dictionary<string, double> computeCOSSIM() {}
```

7. Crea un array con los resultados de la búsqueda. Utiliza el algoritmo de la distancia de Levenshtein para determinar el fragmento de texto (snippet) que se mostrará por cada documento resultante.

```cs
public string[][] normalizeResult(Dictionary<string, double>
simOfDocs) {}
```

8. Aplica expresiones regulares para eliminar caracteres especiales, números y cadenas no deseadas. Fragmenta la cadena obteniendo un array de palabras.

```cs
private static string[] Tokenize(string text){}
```

9. Calcula la distancia entre dos frases. Este método implementa el algoritmo de la distancia de Levenshtein. Además, devuelve el porcentaje de cambios.

```cs
public int LevenshteinDistance(string sentence, string query,
out double percent){}
```

Ejecutando el proyecto:

Como prerrequisito hay que tener instalado .Net Core 6.0. Luego, colocarse en la carpeta del proyecto y ejecutar:

- Para Linux:

```bash
make dev
```

- Para Windows

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