Moogle!

UH

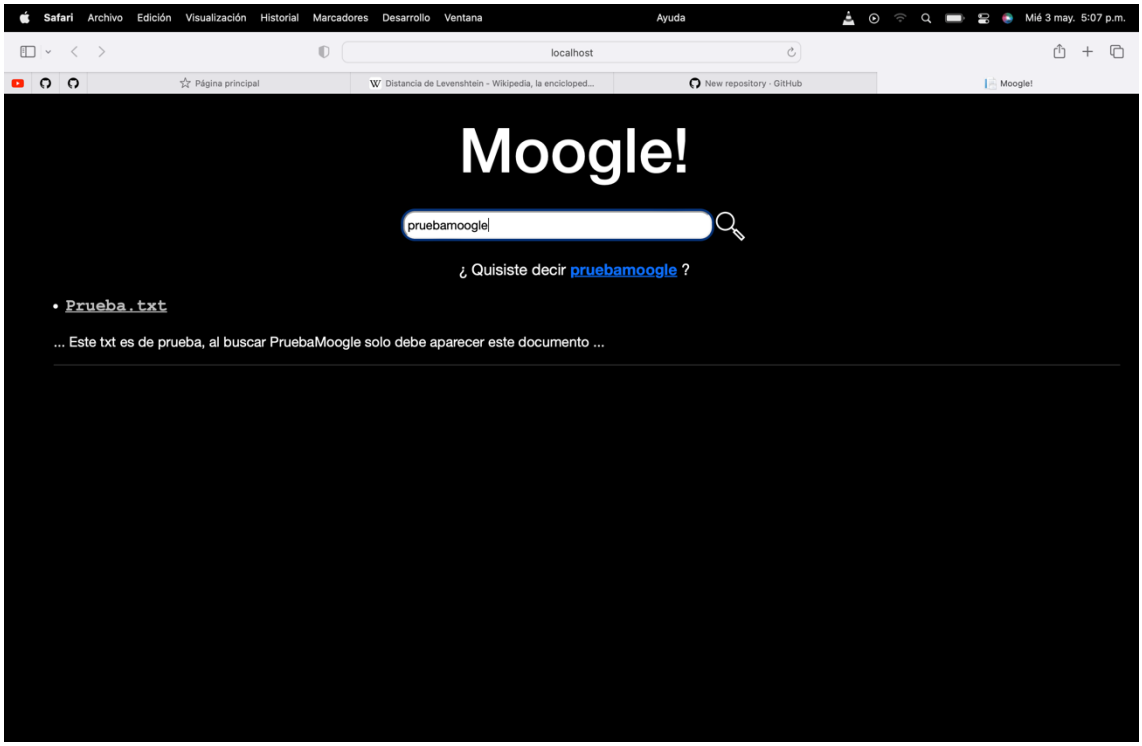
informe

Diego Manuel Viera Martínez
C-111

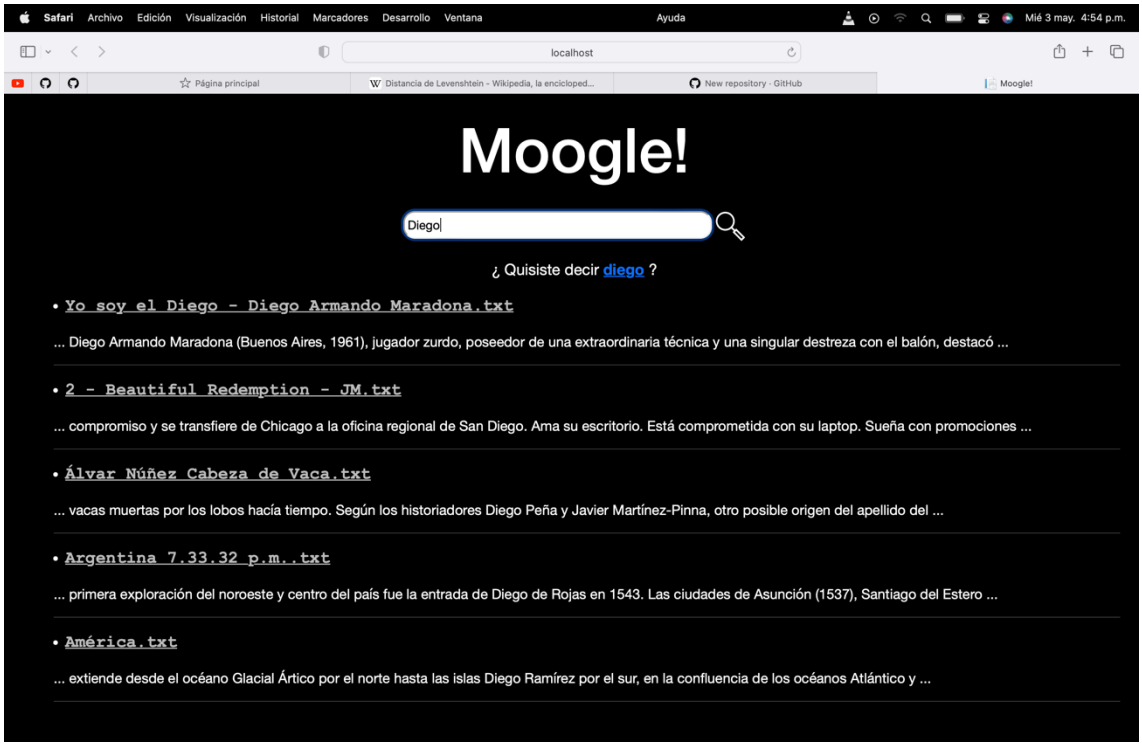
Funcionalidades:

1- Buscador:
-Busca hasta los 5 documentos más relevantes según su búsqueda y los ordena de mayor a menor relevancia, muestra el título y el snippet. Si no hay exactamente 5 o más de 5 documentos muestra solo los documentos encontrados, si ningún documento es relevante, informa que no hay resultados para la búsqueda.

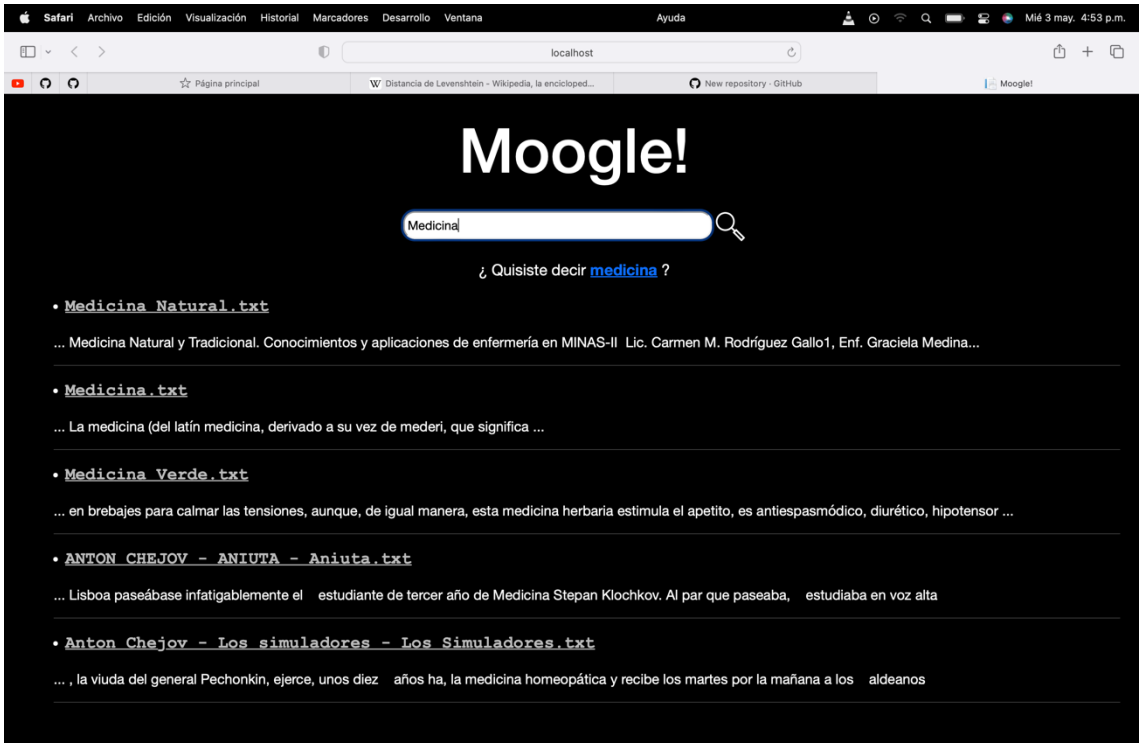
Ejemplos:
1- (Solo aparece un documento)



2-

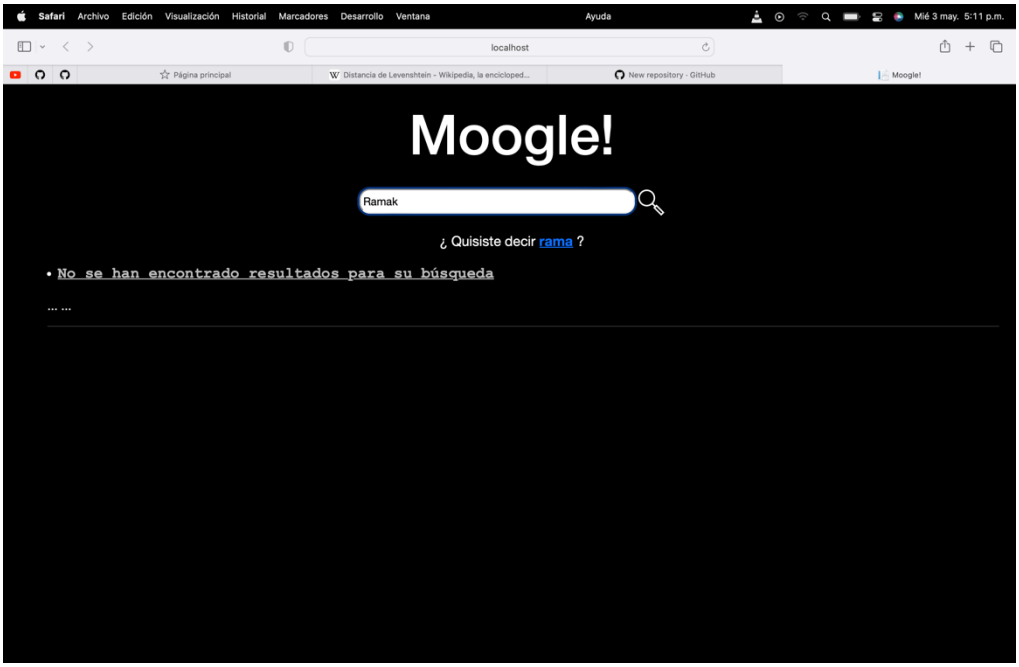


3-



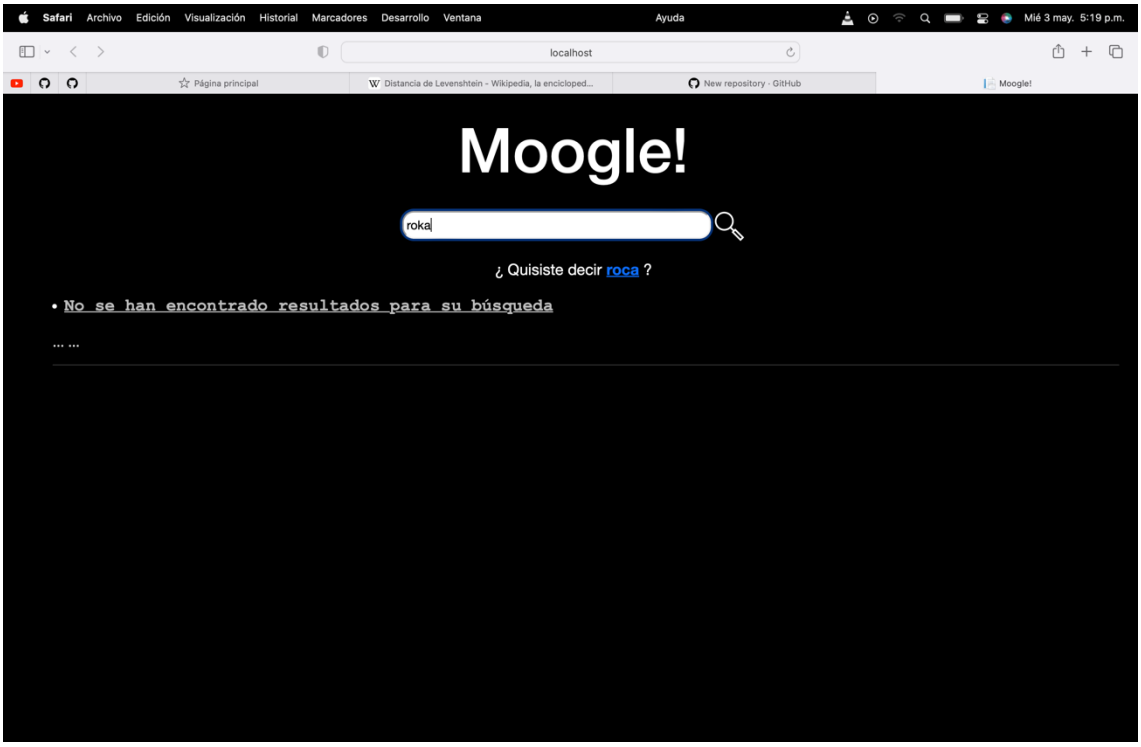
2- Snippet:
-El snippet representa una porción del texto en el cual se encuentra por primera vez la palabra de la búsqueda con mayor importancia para el documento. En los ejemplos del Buscador se muestran los snippet de cada documento.

3- Sugerencia:
-Si la búsqueda contiene una palabra que no aparece en ninguno de los documentos, hace una sugerencia de una palabra corregida que si existe en la base de datos.
Ejemplos:
1-



1.1 Escribe ramak (esta palabra no existe) y luego sugiere (rama), al tocar este hipervínculo hace una nueva búsqueda con la palabra sugerida.

2-



4- Abrir documentos:
-Al dar clic sobre el nombre de los txt , abre una nueva ventana mostrando el contenido del txt.

Explicación del código:

1) Class Tools:

1.1-La clase Tools, primeramente obtiene las direcciones de cada documento y las almacena (DireccionTextos), Almacena en (textos) los textos ya leídos como string, utilizo (FilesNames) para guardar solo los nombres de los textos, almacena en (Text_Words) las listas de palabras tokenizadas de cada documento(esto lo utilizo para obtener cuantas palabras tiene el documento).

1.2-Tools contiene el método de inicialización GetValues() :

-Este método lee los textos y los almacena en (Tools.textos) , le da los valores a: TFxIDF.TodasLasPalabras(Obtiene todas las palabras sin repetirse y almacena además en cuentos documentos aparece), TFxIDF.DiccionarioDeCadaDocumento (Crea un diccionario para cada documento que contiene las palabras de ese documento sin repetirse con el valor de cuantas veces aparece en el documento).
Obtiene los nombres de los textos y los almacena en (Tools.FilesNames).

-Finalmente actualiza los valores de TFxIDF.DiccionarioDeCadaDocumento calculando el TF * IDF de cada palabra de cada diccionario , utilizando los valores de TFxIDF.DiccionarioDeCadaDocumento (obtengo la cantidad de veces que se repite la palabra en el documento) y lo divido entre la cantidad de palabras del documento, obteniendo así el TF; con TFxIDF.TodasLasPalabras obtengo en cuantos documentos aparece la palabra , luego para

calcular el IDF [log(cantidad de documentos / la cantidad de documentos donde aparece la palabra)].

```
foreach(string s in TFxIDF.DiccionarioDeCadaDocumento[i].Keys)
                {
                    //i representa mi documento actual y s la palabra
del documento
                    TFxIDF.DiccionarioDeCadaDocumento[i][s] =
(TFxIDF.DiccionarioDeCadaDocumento[i][s] /
TotalDePalabrasDelDocumento)*(Math.Log10(Tools.textos.Count/TFxIDF.Todas
LasPalabras[s]));
                    //Calculo del TF*IDF
                }
```

1.3-Esta clase contiene además el método Separar_palabras(string texto), el cual retorna una lista de las palabras tokenizadas del texto (necesario para crear los diccionarios para cada documento).

2) Class TFxIDF

2.1-La clase TFxIDF tiene las siguientes propiedades:
- (Dictionary<string , double> TodasLasPalabras), contiene todas las palabras de todos los documentos sin repetirse, almacenando a su vez , en cuántos documentos se encuentra cada palabra (necesario para calcular IDF).

- (List<Dictionary<string , double>> DiccionarioDeCadaDocumento), contiene los diccionarios para cada documento por separado (Cada elemento de la lista reperesenta el diccionario para el documento correspondiente) los diccionarios almacenan las palabras del texto correspondiente sin repetirse , con el valor de la palabra calcualdo en el método de inicialización GetValues() de la clase Tools utilizando TF * IDF

2.2-TFxIDF contiene el método Calcular_Tf_Idf_query(string query) el cual devuelve un Dictionary<string , double> vectorquery , de las palabras del query sin repetirse con su valor (TF * IDF).

3) Class Similitud_Coseno

3.1-Contiene el método Calcular_Similitud_Coseno(Dictionary<string , double> vectorquery, Dictionary<string , double> Doc), en éste método se utiliza el calculo de similitud coseno[(Multiplicación escalar de los vectores) / (Multiplicación de los módulos de los vectores)], el cual expresa que tan parecidos son dos vectores (Expresamos nuestros textos como vectores). Nuestros vectores serían:
(Dictionary<string , double> **vectorquery**) y (Dictionary<string , double> **Doc**). Calculando de esta forma el score de cada texto respecto al query.

3.2-El método ScoreTop(string query) utiliza el método Calcular_Similitud_Coseno(Dictionary<string , double> vectorquery, Dictionary<string , double> Doc) para cada texto y devuelve una lista de los textos con score diferente de 0 , ordenados de mayor score a menor score

Calculando el score de cada documento:

```
List<(string , string , float , int)> ScoreTxT = new List<(string , string , float , int)>();
            Dictionary<string,double> queryTfIdf = TFxIDF.Calcular_Tf_Idf_query(query);
            for(int i = 0 ; i < Tools.textos.Count ; i++)
            {
                float score = (float)Calcular_Similitud_Coseno(queryTfIdf ,
TFxIDF.DiccionarioDeCadaDocumento[i]);
                if(score != 0){
                    ScoreTxT.Add((Tools.FilesNames[i] , " En Desarrollo (snipet) " , score ,
i));//Guardo i para representar mi doc a la hora de calcular el snippet
                }
            }
```

4) En el script Moogle.cs en el método Query llámo al método ScoreTop(string query) y obtengo mi lista de documentos más relevantes según el query, creo un array del tipo SearchItem (SearchItem[] item), añado hasta los 5 documentos más relevantes y finalmente retorno una instancia del tipo SearchResult(item , suggestion).

Opcionales:

1) Snippet

1.1-Para el snippet primeramente busco con el método BestWord(Dictionary<string , double> vectorquery , int index) la palabra más relevante de mi query según el documento. Luego con el método snippet(string BestWord , int index_of_text) , busco el índice de la primera aparición de mi BestWord en el documento y creo un rango de hasta 140 caractéres a partir de este índice, creando de esta forma una porción del texto que contiene la palabra del query más relevante para el documento.

2) Sugerencia

2.1-Para la sugerencia utilizo el algoritmo de Levenshtein, el cual expresa que tan parecidas son dos palabras, cree un método utilizando este algoritmo [LevenshteinDistance(string s , string t)], este retorna un int que representa la cantidad de cambios que deben realizarse para que las palabras sean iguales. Luego el método Suggestion(Dictionary<string , double> vectorquery), toma cada palabra de la búsqueda y verifica que existe en Tools.TodasLasPalabras, si esta existe mantiene la palabra tal como está y la añade a una lista resultante, si no existe la palabra utilizo el método de LevenshteinDistance(string s , string t) , para comparar mi palabra no existente con todas las palabras de Tools.TodasLasPalabras, y quedarme con la que menos cambios deben relaizarce (si varias palabras tienen la misma cantidad de cambios, me quedo simpre con la primera que encuentre con este mínimo de cambios), la añado la palabra con el mínimo de cambios a mi lista resultante y luego creo un string a partir de esta lista , iterando por todas las palabras de query y corrigíendolas.

Añadiendo las palabras de la lista resultante al string:

```
for(int i = 0 ; i < resultwords.Count ; i++)
            {
                if(i == resultwords.Count - 1)
                {
                    result += resultwords[i];//Mi última palabra no nececita el espacio final
                    break;
                }
                result += resultwords[i] + " ";
            }
```

3) Abrir los txt
En el script index.razor cree un método OpenTxt(string title) donde utlizo la clase Process de System.Diagnostics.

```
private static void OpenTxt(string title)//Método para abrir los txt
    {
        char separador = Path.DirectorySeparatorChar;//Obtener separador para diferentes
sistemas
        int index = Tools.DireccionTextos[1].IndexOf(separador + "Content" + separador);//indice
hasta Content
        string direction = Tools.DireccionTextos[1].Remove(index + 9);//Removiendo nombre de
texto para obtener mi ruta hasta content

        //Proceso que abre el txt
        try
        {
            using(System.Diagnostics.Process process = new System.Diagnostics.Process())
            {
                process.StartInfo.FileName = direction + title;
                process.StartInfo.UseShellExecute = true;
                process.StartInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Normal;
                process.Start();
            }
        }
        catch(System.Exception)
        {}
    }
```

Luego en donde se muestra el título añadí un hipervínculo con un evento @onclick que llama al método OpenTxt(item.Title)

```
<div class="item">
            <p class="title"><a class="Title_of_Doc" href="#" @onclick="() =>
OpenTxt(item.Title)">@item.Title</a></p>
            <p class="snippet">... @item.Snippet ...</p>
            <hr>
        </div>
```