🔍 Moogle!

Introduzca su búsqueda | 🔍 Buscar

Este MOOGLE! para realizar la búsqueda de un texto en ciertos documentos realiza lo siguiente:

*Aclarar que cree dos objetos aparte de los ya existentes SearchResult y SearchItem:

Document: Contiene su path, título, todo el texto que contiene, las palabras de este, una lista para su TF, otra para su TFIDF, un recorte de este y su score.

Snippet: Contiene todo su texto, las palabras del mismo, una lista para el TFIDF y su score.

*Antes de abrir el navegador se cargan todos los documentos de la carpeta Content de la siguiente manera:

-Obtiene la dirección de la carpeta y crea una lista con cada uno de los paths de los documentos.

```
public static List<Documents> Docs = new List<Documents>();
readonly static string Folder = Path.Combine(Directory.GetCurrentDirectory().Substring(0, Directory.GetCurrentDirectory().Length - 13), "Content")
static public List<string> paths = Directory.GetFiles(Folder, "*.txt").ToList();
static public int CantidadDoc = paths.Count;
```

- Por cada path añade los documentos a una lista, almacenando las palabras de todos estos en una lista.

```
foreach (string path in paths)
    {
        string aux = File.ReadAllText(path);
        string[] words = Regex.Split(aux.ToLower(), @"\W+");
        Docs.Add(new Documents(path, Path.GetFileNameWithoutExtension(path), aux , words , new List<double>(), new List<double>(),"", 0));
        Allwords.AddRange(words);
    }
 Allwords = Allwords.Distinct().ToList();
 TotalWords = Allwords.Count;
}
```

*Una vez abierto el programa, cuando el usuario introduce la búsqueda se realiza lo siguiente:

- Se llama a un método para que las variables necesarios vuelvan a su estado original permitiendo realizar varias búsquedas sin reiniciar el programa.

```
public static void Clean()
{
    foreach (Documents doc in Docs)
    {
        doc.TF.Clear();
        doc.TFIDF.Clear();
    }
    Search.QueryTF.Clear();
    Search.Dictionary.Clear();
    Search.Docxword.Clear();
    Search.QueryOP.Clear();
    Search.Suggest.Clear();
    Suggestion.SuggestList.Clear();
}
```

-En caso de que la búsqueda sea vacía devuelve un SearchResult con lo siguiente:

```
if (query == "")
{
    SearchItem[] items = new SearchItem[1];
    items[0] = new SearchItem("Ingrese un valor para buscar", "" , 0);
    return new SearchResult(items, query);
}
```

-Si no, almacena el query en minúsculas y llama a un constructor de las clases Search y Suggestion.

*La clase Search es la que realiza la búsqueda de los documentos más importantes, a través de lo siguiente:

-Guarda en una lista la búsqueda separada por espacios y en otra separada por todo tipo de signos con la clase Regex, añadiendo el singular de esas palabras haciendo Stemming gracias al Nuget Porter2Stemmer,.

```
List<string> StemmerQuery = new List<string>();
if (query != null)
{
    Query = new List<string>(Regex.Split(query, @"\W+"));
    Query.Remove("");
    QueryOP = Regex.Split(query, " ").ToList();
    QC = Query.Count;
    foreach (string word in Query)
    {
        StemmerQuery.Add(Stemming(word));
    }
    Query.AddRange(StemmerQuery);
```

```
public static string Stemming(string input)
{
    EnglishPorter2Stemmer stemming = new EnglishPorter2Stemmer();
    string result = stemming.Step0RemoveSPluralSuffix(input);
    result = stemming.Step1ARemoveOtherSPluralSuffixes(input);
    return result;
}
```

-Por cada palabra en la lista de las separada por espacios analiza con que empieza para si tiene algún operador darle menor, mayor importancia, o directamente no debe aparecer el documento, y por cada palabra en la otra dependiendo de si está o no en la lista de todas las palabras de los documentos añade a una lista Suggest 0 o 1 para saber si después debe buscar una palabra parecida, así como la añade a otra lista Dictionary que serían las palabras que buscará en los documentos.

```
List<double> OperatorImportance = new List<double>(
List<double> OperatorExist = new List<double>();
foreach (string item in QueryOP)
{
    if (item.StartsWith("*"))
    {
        OperatorImportance.Add(2);
        OperatorExist.Add(1);
    }
    else if (item.StartsWith("-"))
    {
        OperatorImportance.Add(0.5);
        OperatorExist.Add(1);
    }else if (item.StartsWith("^"))
    {
        OperatorImportance.Add(1);
        OperatorExist.Add(2);
    }
    else if(item.StartsWith("!")){
        OperatorImportance.Add(1);
        OperatorExist.Add(0);
    }
    else
    {
        OperatorImportance.Add(1);
        OperatorExist.Add(1);
    }
}
for (int i = 0; i < Query.Count - QC; i++)
{
    OperatorImportance.Add(OperatorImportance[i]);
    OperatorExist.Add(OperatorExist[i]);
}
```

```
foreach (string word in Query)
{
    bool aux = Library.Allwords.Contains(word);
    if (aux == true)
    {
        QueryTF.Add(OperatorImportance[index]);
        Suggest.Add(1);
        Dictionary.Add(word);
    }
    else
    {
        QueryTF.Add(0);
        Suggest.Add(0);
        Dictionary.Add(word);
    }
    index++;
}
index = 0;
Docxword = Enumerable.Repeat(0, QueryTF.Count).ToList();
```

-Luego calcula el TF y da TF 0 a los documentos que indiquen los operadores de existencia.

```
foreach (double binary in QueryTF)
{
    foreach (Documents doc in Library.Docs)
    {
        double cantidad = doc.Words.Count(s => s == Dictionary[index]);
        if (cantidad != 0)
        {
            Docxword[index] += 1;
            doc.TF.Add((1 + Math.Log10(cantidad)) * binary);
        }
        else
        {
            doc.TF.Add(0);
        }
    }
    index++;
}
```

```
foreach (Documents doc in Library.Docs)
{
    int indexoperator = 0;
    foreach (int aux in OperatorExist)
    {
        if (aux == 0 && doc.TF[indexoperator] != 0)
        {
            doc.TF = Enumerable.Repeat(0.0, QueryTF.Count).ToList();
        }
        if (aux == 2 && doc.TF[indexoperator] == 0)
        {
            doc.TF = Enumerable.Repeat(0.0, QueryTF.Count).ToList();
        }
        indexoperator++;
    }
```

-Luego calcula el TFIDF y la palabra de la búsqueda de mayor importancia en el documento.

```
int index = 0;
float score = 0;
foreach (double tf in doc.TF)
{
    doc.TFIDF.Add(doc.TF[index] * Math.Log10((double)Library.CantidadDoc / (1 + Docxword[index])));
    score += (float)doc.TFIDF[index];
    index++;
}
doc.Score = score;
if (score > 0)
{
    double max = doc.TFIDF.Max();
    if (max != 0)
    {
        string HighestTFIDF = Dictionary[(doc.TFIDF.IndexOf(max))];
        doc.Snippet = HighestTFIDF;
    }
}
```

-Luego añade a otra lista los documentos ordenados descendientemente por su score, y le halla el snippet hasta los 5 primeros mediante el método WordsAround.

```
public static string WordsAround(string input, string word, int numW)
{

    foreach (Match match in Regex.Matches(input, @$"\W+{word}\W+", RegexOptions.None))
    {
        int index = match.Index;
        int start = 0;
        if (index > numW / 2)
        {
            start = index - numW / 2;
        }
        int aux = input.Length;
        int end = numW;
        if (start + numW > aux)
        {
            end = aux - start;
        }
        string result = input.Substring(start, end);
        SearchSnippet.snippets.Add(new Snippets(result, Regex.Split(result.ToLower(), @"\W+"), new List<double>(), 0));
    }

    SearchSnippet searchSnippet = new SearchSnippet();
    return SearchSnippet.Result;
}
```

-El método WordsAround realiza varios recortes alrededor de la palabra más importante de la búsqueda en el documento añadiéndolos a una lista y llamando a un constructor de la clase SearchSnippet que realiza el mismo cálculo del score mediante el TFIDF realizado a los documentos.

```
public static List<Snippets> snippets = new List<Snippets>();
public static int indexr = 0;
public static string Result = "";
1 reference
public SearchSnippet()
{
    foreach (double binary in Search.QueryTF)
    {
        foreach (Snippets doc in snippets)
        {
            double cantidad = doc.Words.Count(s => s == Search.Dictionary[indexr]);
            if (cantidad > 0)
            {
                doc.TFIDF.Add(Search.QueryTF[indexr] * (1 + Math.Log10(cantidad)) * Math.Log10((double)Library.CantidadDoc / (1 + Search.Docxword[indexr])));
                doc.Score += (float)doc.TFIDF[indexr];
            }
            else
            {
                doc.TFIDF.Add(0);
            }
        }
        indexr++;
    }
```

*La clase Suggestion es la que dependiendo de si las palabras de la búsqueda estaban o no en los documentos, sugiere las más parecidas a estas a través de la distancia de Levenshtein.

```
public static int index = 0;
public static List<string> SuggestList= new List<string>();
public static string Suggest = "";
1 reference
public Suggestion(){

    for (int i = 0; i < Search.QC; i++)
    {
        if(Search.Suggest[index] == 0)
        {
            Search.Query[index] = SuggestWord(Search.Query[index]);
        }
        SuggestList.Add(Search.Query[index]);
        index++;
    }
    index = 0;
    Suggest = string.Join(" ", SuggestList.ToArray());
}
```

```
public string SuggestWord(string word)
{
    int distance = 10000000;
    string suggest = "";
    foreach(string word2 in Library.Allwords)
    {
        int aux = Levenshtein(word, word2);
        if(aux < distance) {
            distance = aux;
            suggest = word2;
        }
    }
    return suggest;
}
```

```
public int Levenshtein(string word, string word2)
{
    char[] charword = word.ToCharArray();
    char[] charword2 = word2.ToCharArray();
    int x = word.Length + 1;
    int y = word2.Length + 1;
    int[,] Levenshtein = new int[y,x];
    for(int i = 0; i < x; i++)
    {
        Levenshtein[0,i] = i;
    }
    for(int i = 0; i < y; i++)
    {
        Levenshtein[i, 0] = i;
    }
    for (int i = 1; i < y; i++)
    {
        for(int j = 1; j < x; j++)
        {
            if(charword2[i-1] == charword[j - 1])
            {
                Levenshtein[i,j] = Math.Min(Math.Min(Levenshtein[i - 1, j - 1], Levenshtein[i, j - 1]), Levenshtein[i - 1, j]);
            }
            else
            {
                Levenshtein[i,j] = Math.Min(Math.Min(Levenshtein[i-1,j-1] + 1, Levenshtein[i,j-1] + 1), Levenshtein[i-1,j] + 1);
            }
        }
    }
    return Levenshtein[y-1,x-1];
}
```

*Luego en caso de que no encuentre ningún documento relacionado con la búsqueda devuelve las palabras sugeridas y un SearchResult como el siguiente:

```
if (Search.OrdDocs.Count == 0)
{
    SearchItem[] items = new SearchItem[1];
    items[0] = new SearchItem("La busqueda no coincide con ningun documento", "", 0);
    return new SearchResult(items, Suggestion.Suggest);
}
```

-En caso de que sí existan documentos relacionados devuelve hasta 5 de ellos y las palabras sugeridas.

```
else {
    int r = Math.Min(Search.OrdDocs.Count, 5);
    SearchItem[] items = new SearchItem[r];
    for (int i = 0; i < r; i++)
    {
        items[i] = new SearchItem(Search.OrdDocs[i].Title, Search.OrdDocs[i].Snippet, Search.OrdDocs[i].Score);
    }
    return new SearchResult(items, Suggestion.Suggest);
}
```

Espero que con la lectura de este informe pudiera entender el funcionamiento del MOOGLE!