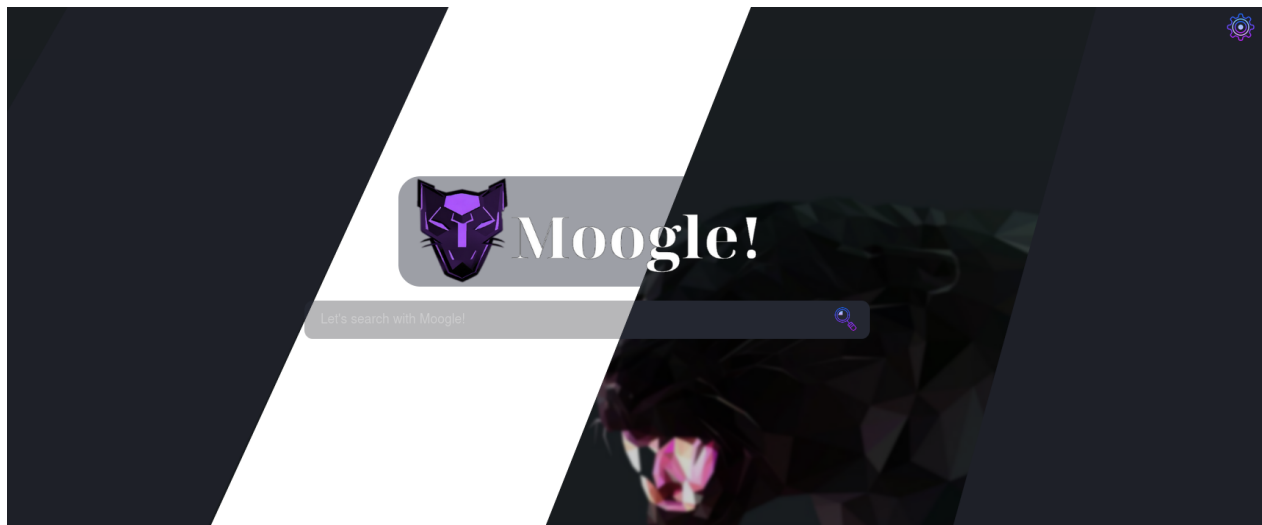


# Bienvenido a MOOGLE!

Rafael Sánchez Martinez

July 15, 2023



1er Proyecto de Programación - MatCom

Curso 2023-24

Grupo: C-122

Autor: Rafael A. Sánchez Martínez

## Features

- Soporta búsqueda de temas varios.
- Modo Oscuro y Modo Claro.
- Relativamente rápido, probado con 30 documentos(~40mb).
- Capacidad de uso de operadores de Inclusión ('^'), Exclusión('!') y Cercanía('~').
- Posibilidad de devolver sugerencias, una vez la consulta sea procesada y determinada incorrecta o inexistente en el Corpus.
- Muestras de pequeñas secciones de los documentos donde se haya encontrado lo solicitado.
- Muestra el Puntaje otorgado a cada documento dependiendo de lo consultado.

# 1 Funcionamiento

## 1. Como comenzar:

- Si usted se encuentra en Linux, ejecutar en la carpeta del proyecto desde una terminal:  
`make dev`
- Si usted se encuentra en Windows o MacOS, ejecutar en la carpeta del proyecto desde una terminal:  
`dotnet watch run -project Moogleserver`

## 2. Inicio:

- En la zona superior derecha elegir los modos(Oscuro, OscuroSólido, ClaroSólido).
- El programa inicia en ‘‘Program.cs’’
  - Linea 5 `Moogles.LetsGetStarted(@"../Content");`
  - Esta es la función invocada presente en "Moogles.cs": Linea-13  
`public static void LetsGetStarted(string path) corpus = new Corpus(path);`
- 2.1 Aquí se le da paso al motor de búsqueda, que tratará de crear el Diccionario "GeneralFiler" que contendrá todas las palabras de los documentos 'MASE Corpus -¿ Linea 4'
- 2.2 También se creará el diccionario casi más relevante del proyecto, "Docs", que almacenará cada documento con sus datos 'MASE Corpus -¿ Linea 5'

## 3. Corpus:

- 3.1 Se ejecuta el constructor de esta clase:
  - 3.1.1 - `GetInfo()`, esta función extraerá los archivos de la carpeta content y los agregará al GeneralFiler(VocabularioGeneral)
  - 3.1.2 - `IDF()`, esta función calculará el IDF de las de los documentos, llamando la función IDF de la línea 72 de esa misma clase.
  - 3.1.3 - `WW()` o Peso de la palabra, esta función la uso para guardar el peso de cada palabra en su documento
    - \* 3.1.3.1 Aquí se ejecuta la función `Peso()`, perteneciente a la clase 'Data'(Explicada más adelante), pero no hace más que calcular el peso de cada palabra y darle valor modular al documento procesado en cuestión .
- 3.2 En las anteriores funciones se utilizaba la función BuildGeneralFiler, la cual como su nombre indica, se encarga de construir en GeneralFiler(VocabularioGeneral), pero además procesar y desarrollar el diccionario "Docs".
  - 3.2.1 Aquí creo el objeto data `Data data = new Data();` y el conteo que indica cada palabra  
`int count = 0`
- Esta zona del código fue un descubrimiento excepcional, estoy orgulloso de ello, horas en la página de Microsoft(no es broma)  
[h]

```

1 private void BuildGeneralFiler(string nombre, int i)
2 {
3     Data data = new Data();
4     // Index count of each word
5     int count = 0;
6     // Splitting with signos de puntuacion
7
8     string txt = File.ReadAllText(Directory.GetFiles(Path, "*.txt")[i]).ToLower();
9     string[] palabras = txt.Split(new char[] { ' ', ',', '.', ';', '?', '!', '\t', '\n', ':', '"', '\r' }, StringSplitOptions.RemoveEmptyEntries);
10    foreach (string word in palabras)
11    {
12        if (word.Length == 1 && Char.IsPunctuation(word[0])) continue;
13        // aqui voy guardando cada una de las palabras en el Vocabulary del documento con sus indices
14        if (!data.Vocabulary.ContainsKey(word))
15        {
16            data.Vocabulary.Add(word, new List<int>());
17            data.pesos.Add(word, 0);
18            if (!GeneralFiler.ContainsKey(word))
19            {
20                GeneralFiler.Add(word, 0);
21            }
22        }
23        data.Vocabulary[word].Add(count);
24        if (data.Vocabulary[word].Count > data.MaxWordAppereance)
25        {
26            data.MaxWordAppereance = data.Vocabulary[word].Count;
27        }
28        count++;
29    }
30    Docs.Add(nombre, data);
31 }

```

- 3.3.2 Como se puede ver, aquí ocurre casi toda la "magia", aquí se usan métodos propios de los diccionarios como "ContainsKey", "Add", etc. Aquí se va a separar el texto de cada documento por espacios y diferentes caracteres (como se observa arriba), y se va a ir almacenando cada palabra resultante en el vocabulario con sus índices, y finalmente se va a formar el Diccionario docs, con los nombres de los documentos y sus datos
  - 2.3.2.1 Estos datos se van recopilando a lo largo del bucle foreach dentro de esta función BuildGeneralFiler
- 3.4 Data, la clase que contiene los datos de cada documento donde se calcula el peso de cada palabra en el documento que esté analizando en dicho momento, además le da valor a la variable "Module" del documento, que no es más que el módulo del vector de peso

```

1 public void Peso(Dictionary<string, double> GFiler)
2 { // Este es el metodo q calcula el peso de cada palabra en el documento y le da valor al Module del documento
3     foreach (var par in pesos)
4     {
5         pesos[par.Key] = (double)Vocabulary[par.Key].Count / (double)MaxWordAppereance * GFiler[par.Key];
6         Module += Math.Pow(pesos[par.Key], 2);
7     }
8     Module = Math.Sqrt(Module);
9 }

```

- 3.4.1 Otros valores de Data: int MaxWordAppereance = 0; -¿ Frecuencia de la palabra que más aparece Dictionary Vocabulary -¿ Este es el vocabulario del documento contra los índices de las palabras de ese vocabulario

#### 4. MASE: Consulta(Searcher) y Puntaje(Score)

- 4.1 El constructor de la clase, recibe la entrada del usuario(Query) desde el apartado grafico, y un Corpus(El ya creado anteriormente y creado al inicio del proyecto)
  - UsrInp es la consulta del usuario ya procesada, por el metodo ProcessQuery, que lo que hace no es más que separar en terminos la consulta
  - LetMeIn(Lista de Inclusion), LetMeOut(Lista de Exclusion), Closeness(Lista de cercanía), se encargaran de recibir los terminos de la búsqueda según los operadores colocados.
  - GetInfo, es la función casi que más caótica, aquí primeramente se separan los terminos segun sus operadores, posteriormente cada palabra de la consulta va para el diccionario Frqhzy con su cantidad de repeticiones:

```
1  if (!Frqhzy.ContainsKey(UsrInp[i]))
2      {
3          Frqhzy.Add(UsrInp[i], 0);
4      }
5      Frqhzy[UsrInp[i]] += count1 + 1;
6      if (MaxWordAppereance < Frqhzy[UsrInp[i]])
7      {
8          MaxWordAppereance = Frqhzy[UsrInp[i]];
9      }
```

- 4.2
  - Luego en la linea 43, el corpus declarado en esta clase searcher, pasa a ser el corpus enviado a consultar
  - GSSuggest, funcion que usando la distancia de Levensthein(Aun no optimizada), sustituye las palabras mal escritas o no encontradas de la consulta, por otras posiblemente más adecuadas. Además incorpora el método Suggestion()

```
1  private static string Suggestion(string word, Corpus corpus)
2  {
3      string suggestion = "";
4
5      if (!corpus.GeneralFiler.ContainsKey(word))
6      {
7          for (int i = 1; i < word.Length / 3 + 1; i++)
8          {
9              foreach (var pair in corpus.GeneralFiler)
10             {
11                 if (LevenstheinDistance(word, pair.Key) == 1) { suggestion = Compare(suggestion, pair.Key, word, corpus); }
12                 if (suggestion != "") return suggestion;
13             }
14         }
15     }
16     return suggestion;
17 }
```

- Usando la distancia de Levenstein(No optimizada aún), recorre palabra por palabra, para buscar por poca diferencia, la palabra más semejante de la consulta, para finalmente devolverla. [Mi idea es crear al inicio un diccionario extra, o varios, que abarquen todo el Vocabulario de mis Docs y ordene por tamaño todas las palabras, así a la hora de sugerir una palabra solo tendría que calcular con palabras 1 caracter más o menos grande, o de igual tamaño]
- Snippets = new string[corpus.Docs.Count] crea el string Snippets con longitud igual a la cantidad de documentos, este string pues almacenará eso, los snippets con score != 0.
- WVal = new double[UsrInp.Length] es un array de dobles, que tendrá los valores de los terminos de la consulta, con capacidad == cantidad de terminos del UsrInp(Consulta).
- 'Save W Value' usando la conocida formula de  $TF \cdot IDF$ , pues calcula los valores de peso de los terminos de la consulta. MASE LN -i 57
- Mod() Calculará el vector de pesos de la consulta

```

1 public void Mod()
2     {
3         for (int i = 0; i < WVal.Length; i++)
4         {
5             Module += Math.Pow(WVal[i], 2);
6         }
7         Module = Math.Sqrt(Module);
8     }

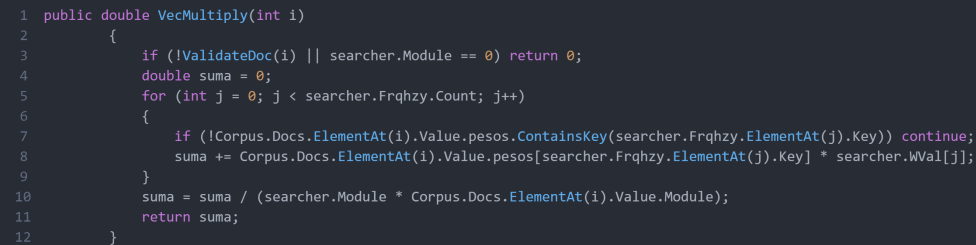
```

- FillSuggest() Metodo que modifica la sugerencia a devolver.

## 5. Score:

- 5.1 Clase que guardará los puntajes de cada documento, para finalmente ser mostrado en el partado gráfico
  - public Searcher searcher Se declara una consulta
  - public Corpus Corpus Se declara un corpus
  - public (string, double)[] tupla Se declara este array que será de igual tamaño que la cantidad de documentos, ademas almacena sus puntajes.

- 5.2 El constructor de esta clase
  - Comienza recibiendo y tomando una consulta y un corpus
  - Como había dicho, la tupla se crea, con igual longitud que la cantidad de documentos
  - FillScores() función que ordena las tuplas de mayor a menor, luego de haber ejecutado un producto vectorial con la función VecMultiply:



```

1 public double VecMultiply(int i)
2 {
3     if (!ValidateDoc(i) || searcher.Module == 0) return 0;
4     double suma = 0;
5     for (int j = 0; j < searcher.Frqhzy.Count; j++)
6     {
7         if (!Corpus.Docs.ElementAt(i).Value.pesos.ContainsKey(searcher.Frqhzy.ElementAt(j).Key)) continue;
8         suma += Corpus.Docs.ElementAt(i).Value.pesos[searcher.Frqhzy.ElementAt(j).Key] * searcher.WVal[j];
9     }
10    suma = suma / (searcher.Module * Corpus.Docs.ElementAt(i).Value.Module);
11    return suma;
12 }

```

- Esta función de FillScores(), también da inicio a la función FillSnippet(tupla) que va rellenando los snippet (Ver más adelante en la sección 5), además usa un método para modificar el score, llamada ModScore(), que depende de si alguna(s) palabra(s), pertenecen a la lista de Closeness y por tanto cambia el score en dependencia de la cercanía entre esos términos. Además usa un BubbleSort(), el método de sorteo más sencillo, recorre los términos a pares y los intercambia si están en el lugar equivocado.
- El método ModScore() usa además la función LowestDistance, que como su nombre indica devuelve la menor distancia entre dos términos en un documento.
- 5.3
  - Swap() Simple función que cambia dos elementos de lugar en un array.
  - TotalWeight() Método que suma y devuelve los pesos de una palabra en cada aparición de esta en cada documento. Se usa en la función Compare().
  - ValidateDoc() Función que otorgará puntaje igual a 0 a aquellos documentos que contengan una palabra excluida y también otorgará 0 a cada documento que no contenga a una palabra incluida.
  - Compare() método presente en la clase Searcher, Ln -j, 199, que se encargará de devolver entre dos palabras, la más importante usando el método TotalWeight()

## 6. Snippet

- 6.1 FillSnippet() es la función, que se encarga de llenar los snippets de aquellos documentos que pasaron el Score(!=0)
  - Usa el hilo "Relevant" sinónimo de MASIMPORTANTE, que contendrá la palabra con la cual se presentará el Snippet más adelante.
  - Al final de la función se invoca el método RetSnippet().
- 6.2 RetSnippet() Esta recibirá, esa palabra "Relevant" y el documento donde se encuentre y creará un snippet que contenga esa palabra.

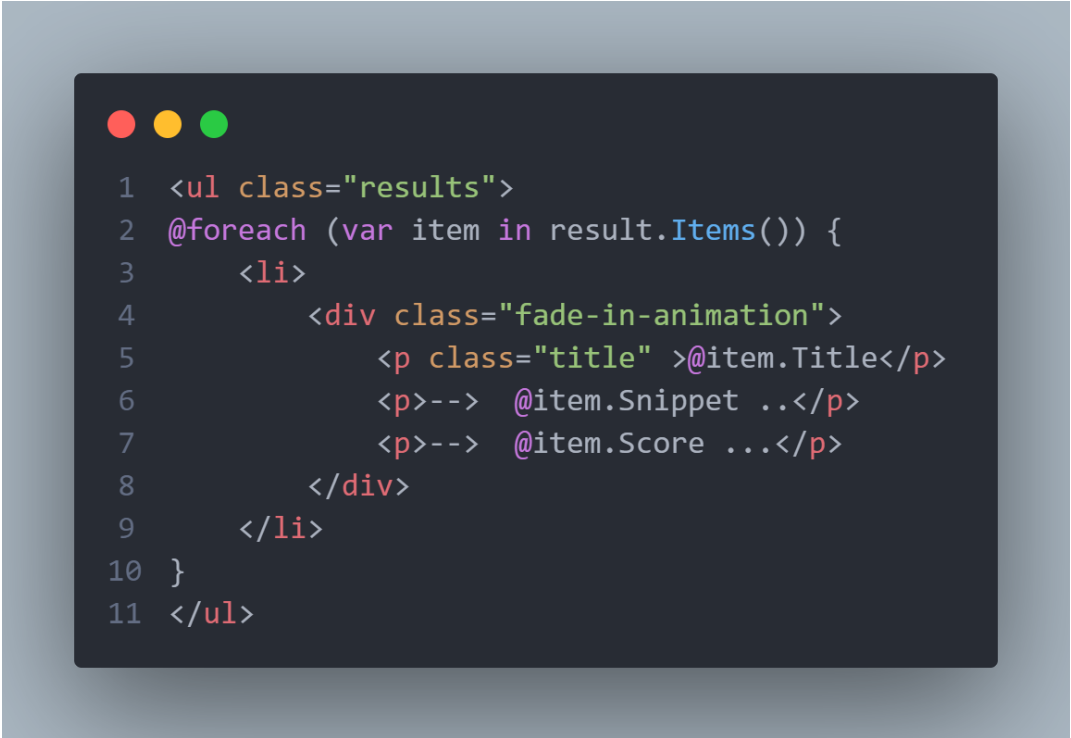
## 7. Cambios en SearchItem y SearchResult

- 7.1 SearchItem Score, lo cambié de float a double.
- 7.2 SearchResult
  - No declaro un objeto SearchItem[], sino una List<SearchItem> items, del mismo nombre.
  - El constructor de esta clase ahora recibe List<SearchItem> items, y su sobrecarga ahora hereda una Lista igualmente.
  - La variable "Count" devuelve this.items.Count en lugar de this.items.Length

```
1 public class SearchResult
2 {
3     public List<SearchItem> items;
4
5     public SearchResult(List<SearchItem> items, string suggestion="")
6     {
7         if (items == null) {
8             throw new ArgumentNullException("items");
9         }
10
11         this.items = items;
12         this.Suggestion = suggestion;
13     }
14
15     public SearchResult() : this(new List<SearchItem>()) {
16
17     }
18     public string Suggestion { get; private set; }
19
20     public IEnumerable<SearchItem> Items() {
21         return this.items;
22     }
23
24     public int Count { get { return this.items.Count; } }
25 }
```

## 8. Retornando al inicio:

- 8.1 La clase Moogole ahora:
  - Declara un Corpus, el mismo que da inicio al programa, y el cual se usará para la búsqueda.
  - Declara una Consulta(searcher), la cual hará todos los pasos y metodos anteriormente explicados.
  - Declara un Score, puntaje que una vez procesado, será mostrado en pantalla, una vez culmine la búsqueda
  - Un cronómetro(No Funcional... por ahora) que mostrará cuanto tardó la consulta (Google-like)
  - Finalmente método "Query" de tipo SearchResult
    - \* Dará orden de inicio a Searcher
    - \* Dará orden de inicio a Score
  - Esta función "Query" devolverá los "items" (Titulo, Snippet, Score) y la sugerencia final.
- 8.2 En Index.razor:
  - Existe una nueva variable double que tendrá el valor en segundos del tiempo tomado en procesar la consulta.
  - A esta región le agregué el "Score".



```
1 <ul class="results">
2   @foreach (var item in result.Items()) {
3     <li>
4       <div class="fade-in-animation">
5         <p class="title" >@item.Title</p>
6         <p>--> @item.Snippet ..</p>
7         <p>--> @item.Score ...</p>
8       </div>
9     </li>
10  }
11 </ul>
```

## 9. The End