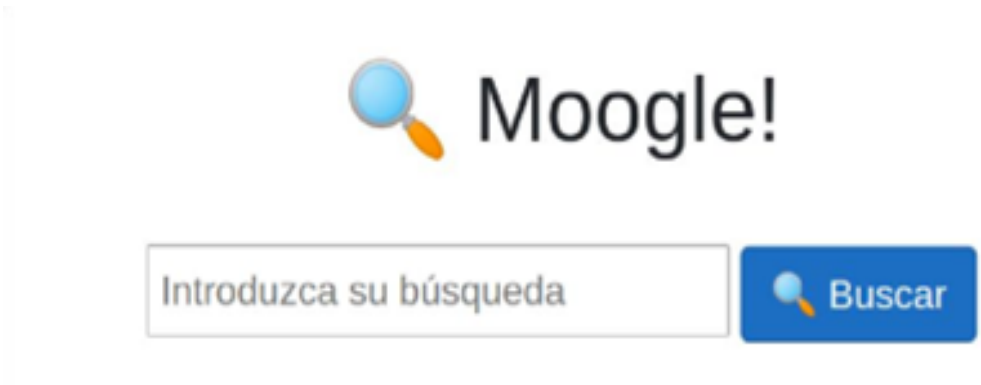
Facultad de Matemática y Computación

PRIMER PROYECTO DE PROGRAMACIÓN



Autor: **Joel Aparicio Tamayo**

Grupo: **C-121**

Moogle !

Especificaciones y particularidades:

1. El proyecto ha sido probado con un repositorio de alrededor de 15 mil documentos cuyo peso era de 170 MB (En GitHub estará subida la carpeta Content con solo 10 documentos). Con estos documentos el proyecto demora alrededor de 1 minuto inicialmente para cargar (solo la primera vez, luego si se cierra y se abre nuevamente, demora tan solo 10 segundos), y la búsqueda es casi instantánea.

2. La búsqueda muestra los 10 primeros resultados como máximo. (Pudieran ser más si hay scores repetidos)

3. El buscador funciona tanto al seleccionar el botón "Buscar", como al presionar la tecla "Enter".

4. El snippet mostrado en pantalla de cada documento muestra una vecindad de la primera aparición de la palabra de más peso de la query con respecto a dicho documento.

5. Se ha implementado el operador de relevancia ($*$), el operador de obligatoriedad ($\wedge$), el operador de inexistencia (!) y el operador de cercanía ($\sim$). (Dicho funcionamiento será explicado en la sección "Estructura de Clases Operators")

6. Los operadores $\wedge$ y ! funcionan solo para la primera palabra de la query que los tenga. (Ej: "!hola !Claudia" solo funcionará para "hola", mientras que a "Claudia" la tratará como una palabra normal)

7. Pueden existir combinaciones de los operadores. (Uno en cada palabra distinta, no puede haber dos juntos en una misma palabra)

8. El operador de cercanía se debe usar de la forma: palabra1 ($\sim$) palabra2 (espacio entre el operador y las palabras).

Estructura del Proyecto (Dentro de la carpeta Moogle Engine):

El proyecto posee 8 clases. De ellas hay dos que ya estaban creadas y que no fueron modificadas, que son: SearchItem y Searchresult. Por tanto, no concierne explicar su funcionamiento en este informe. Tampoco nos interesa demasiado aquí el funcionamiento de la clase *Matriz*, ya que no tiene aplicación alguna en el buscador (es solo para la parte de Álgebra I). No obstante, esta clase está bien comentada en el código. Las clases que nos importa explicar son:

- **Repository:** Es la clase que genera el repositorio de documentos. (Ver sección *"Estrsuctura de Clases"*)
- **Document:** Es la clase que tiene los métodos para trabajar con los documentos. (Ver sección *"Estrsuctura de Clases"*)
- **RankingVector:** Es la clase que contiene el método para asignarle un score a cada documento. (Ver sección *"Estrsuctura de Clases"*)
- **Operators:** Es la clase que tiene los métodos de los operadores implementados. (Ver sección *"Estrsuctura de Clases"*)
- **Moogle:** Es la clase principal del proyecto y donde se generan los resultados de las búsquedas. (Ver sección *"Funcionamiento en tiempo de búsqueda"*)

Estructura de Clases:

- **Repository:**

Esta clase estática, como su nombre indica, carga el repositorio, y guarda los datos necesarios en sus campos estáticos. Los más fundamentales son el campo `wordsFrequency` (una lista de diccionarios que tendrán como claves cada palabra del documento i, y como valor su TF) y `docsFrequency` (un diccionario con todas las palabras del corpus como keys, y como valor la cantidad de documentos en que aparecen).

Además, esta clase tiene el método `CreateRepository` que recibe como parámetro un string predefinido como "Content", que es la carpeta donde están los documentos del repositorio. (En caso de querer cambiar el repositorio de lugar debe pasarse el nombre de dicha carpeta al método) Este método es el encargado de guardar la información necesaria en los campos apreciados anteriormente, para ser utilizadas por otras clases. Pero, ¿cómo funciona? A continuación, se explica brevemente.

Primero almacena las rutas de los documentos en su campo, e inicializa los demás campos. (En la declaración de los campos se inicializaron con longitud 1 para evitar advertencias por posibles valores null)

Luego, por medio de un primer bucle for, va a iterar sobre cada documento. Como primera parte de este bucle, se va a guardar el nombre del documento correspondiente en su campo, se agregará el texto al campo correspondiente y se le hará Split al texto normalizado (utilizando el método `Normalize` de la clase *Document* -Ver *Estructura de Clases.Document* posteriormente-) al texto para separarlo por palabras (y se eliminarán las cadenas vacías también con `StringSplitOptions.RemoveEmptyEntries`) y guardarlo en otro campo. Como segunda parte de este bucle, se itera en un segundo for (esta vez utilizamos foreach) para recorrer cada palabra del texto (ya que tenemos las palabras separadas ya en uno de los campos). Este bucle interno va a permitir tener almacenado en otro campo la cantidad de veces que se repite cada palabra en el documento, y a la vez, en otro campo, en cuántos documentos han ido apareciendo dichas palabras. A continuación, se aprecia visualmente cómo funciona dicho bucle interno:

```
52          wordsFrequency.Add (new Dictionary<string, float>());
53
54          /* Iteramos sobre cada elemento de filesWords para saber cuántas veces se repite una palabra
55          en el documento  y en cuántos documentos está*/
56          foreach (string word in filesWords[i])
57          {
58              /* Si la palabra no está en el diccionario la añadimos como key y se le hace corresponder
59              valor 1 (la cant de veces que se ha repetido hasta el momento). */
60              if (!wordsFrequency[i].ContainsKey(word))
61              {
62                  wordsFrequency[i][word] = 1;
63              /* En caso de que el diccionario ya contenga la palabra como key, entonces sumamos
64              uno a su valor (o sea que se está repitiendo una vez más de lo que ya lo hacía). */
65              } else  wordsFrequency[i][word]++;
66
67              /* Si la palabra no está en el diccionario docsFrequency la agregamos y le damos valor 1
68              (aparece al menos en un documento)*/
69              if (!docsFrequency.ContainsKey(word)) docsFrequency[word] = 1;
70
71              /* En caso de que la palabra aparezca en el diccionario entonces quiere decir :
72              1. O la palabra está repetida en el mismo documento.
73              2. O la palabra se encuentra en otro documento.*/
74              else if (wordsFrequency[i][word] == 1)
75              {
76              /* Como el valor de frecuencia de la palabra es 1 entonces es primera vez que la
77              palabra aparece en el documento, pero como ya estaba en docsfrequency entonces significa
78              que ha aparecido en otro documento distinto. Entonces sumamos 1.*/
79                  docsFrequency[word]++;
80              }
81          }
```

Como tercera parte del bucle for externo, y una vez conlcuido el segundo for (foreach), existe un nuevo for (otro foreach), que itera sobre las palabras sin repetir de cada texto (que en este caso son las llaves de un diccionario y tienen como valor la cantidad de veces que se repiten) y teniendo la cantidad de veces que se repiten, se intercambia ese valor (en su diccionario correspondiente) por su TF (utilizando el método TF de la clase *Document* –Ver *Estructura de Clases.Document* posteriormente-). Y de esta manera ya el for externo está completo en su funcionalidad, y con él, el método también concluye. Una vez que el método es llamado, ya se podrá obtener la información de los documentos llamando a los campos estáticos de esta clase. Aclaración: La clase *Repository* es estática para que al ser llamado su método en otra parte puedan ser llenados sus campos sin necesidad de tener que crear un objeto. Esto será profundizado en la pequeña sección *"¿Cómo cargar el repositorio?"*.

- **Document:**

Esta clase tiene varias funcionalidades, pero todas relacionadas al trabajo con los documentos o las palabras de los documentos. Está constituida por 4 métodos: `Normalize, NormalizeQuery, TF, IDF, GetSnippet`.

El método `Normalize` tiene la función de normalizar un string. Primero lo convierte a minúsculas y elimina las tildes.En una segunda parte se utiliza el método Replace de la clase *Regex* de *System.Text.RegularExpressions* para convertir en espacios en blanco los caracteres innecesarios. A continuación, se muestra cómo funciona:

```
8       // Método para normalizar los documentos.
        2 references
9       public static string Normalize(string texto)
10      {
11          texto = texto.ToLower();
12          texto = texto.Replace("á","a");
13          texto = texto.Replace("é","e");
14          texto = texto.Replace("í","i");
15          texto = texto.Replace("ó","o");
16          texto = texto.Replace("ú","u");
17
18          /* Con el método Replace de la clase Regex vamos a cambiar por " " todos los caracteres que
19          no sean "ñ" o los caracteres de la "a" a la "z" o los caracteres del "0" al "9". */
20          texto = Regex.Replace(texto, @"[^ña-z0-9]", " ");
21
22          return texto;
23      }
```

Finalmente, el método devuelve el string normalizado.

El método `NormalizeQuery` funciona análogamente al método anterior, la única diferencia es que se mantienen los operadores si los tiene.

El método `TF` recibe dos parámetros: la frecuencia de la palabra en un documento y la cantidad de palabras de ese documento. Con estos datos devuelve entonces el TF de la palabra que no es más que el peso de la misma en el documento, y tiene su fórmula específica. El método `IDF` recibe dos parámetros: la cantidad de documentos y la cantidad de documentos que poseen la palabra. Con estos datos devuelve entonces el IDF de la palabra que no es más que el peso de la misma en el cuerpo de documentos, y tiene su fórmula específica. Por último, se encuentra el método `GetSnippet` que es el que va a generar el snippet a mostrar en pantalla de cada documento. Recibe como parámetro el índice de un documento. Primeramente, dentro de un bucle while (true) se guarda el índice de la palabra de la query con mayor IDF (utilizando los valores almacenados en el campo `wordsIDF` de la clase *RankingVector* –Ver *Estructura de Clases.RankingVector* posteriormente-) en la variable `index1`. Luego se utiliza este índice para obtener la palabra de la query y buscarla en el documento normalizado. (que estará guardado en la variable `text`). Si el índice es válido y la palabra está en el documento, pues se rompe el bucle, sino continúa con la siguiente palabra de mayor IDF:

```
        1 reference
56      public static string GetSnippet(int i) // Le pasamos el índice del documento a buscar.
57      {
58          int index2 = 0;
59          string snippet = "";
60          int count = 0; // esto llevará la cuenta de las palabras que se han iterado más adelante.
61          float[] arrayIDF = new float[RankingVector.wordsIDF.Length];
62          RankingVector.wordsIDF.CopyTo(arrayIDF,0);
63
64          // Con el bucle comprobamos si la palabra de más IDF está en el documento, sino la que le sigue.
65          while (true)
66          {
67              // En index1 guardamos la posición de la palabra con más IDF.
68              int index1 = Array.IndexOf(arrayIDF,arrayIDF.Max());
69
70              //Normalizamos el documento.
71              string text = Normalize(Repository.allTexts[i]);
72
73              /* En index2 guardamos la posición de la palabra en el texto. Se señala que debe ser la palabra
74              entre espacios, para que por error no devuelva el índice de una secuencia de caracteres que
75              contenga dicha palabra dentro */
76              index2 = text.IndexOf(" " + RankingVector.wordsQuery[index1] + " ");
77
78              if (index2 == -1)
79              {
80                  arrayIDF[index1] = float.MinValue; // Reducimos el valor para obtener el siguiente máximo.
81                  if (count == RankingVector.wordsIDF.Length) break; /* Si se ha iterado sobre todas las palabras se
82                  rompe el bucle */
83                  count++; // pasamos a la siguiente palabra.
84              }
85              else break;
86          }
```

Luego de este bucle, el método procede a retornar el snippet dependiendo de si cumple o no ciertas condiciones:

```
88          /* En caso extremo (cosa que no debería ocurrir) si existe alguna falla en los índices entonces
89          pasamos un valor por defecto 0 a index2 para evitar una excepción. */
90          if (index2 == -1) index2 = 0;
91
92          /* Si el snippet que queremos imprimir es válido (no se sale del rango del documento) entonces
93          lo imprimimos. */
94          if (index2 - 150 >= 0 && index2 + 150 < Repository.allTexts[i].Length)
95          {
96              snippet = Repository.allTexts[i].Substring(index2 - 150, 300);
97          }
98
99          /* En caso de que el snippet se salga de rango por un índice mayor al largo del documento entonces
100         imprimimos 150 caracteres anteriores a index2 y detenemos la impresion en el último caracter del
101         texto. */
102         else if ((index2 > Repository.allTexts[i].Length - 151) && (index2 - 150 >= 0))
103         {
104             snippet = Repository.allTexts[i].Substring(index2 - 150);
105         }
106
107         /* Si se sale de rango por índice menor a 0 entonces imprimimos desde index2 hasta 150 caracteres
108         posteriores. */
109         else if (Repository.allTexts[i].Length > index2 + 150)
110         {
111             snippet = Repository.allTexts[i].Substring(index2, 150);
112         }
113
114         /* Si se sale de rango por ambos lados quiere decir que el texto es muy pequeño, luego lo imprimimos
115         en su totalidad. */
116         else snippet = Repository.allTexts[i];
117
118         return snippet;
```

- **RankingVector:**

Esta clase está creada para confeccionar el score de los documentos. Posee dos campos estáticos, uno para guardar la query separada por palabras y poderla utilizar en la clase *Document* para crear el snippet; y otro para almacenar los IDF de cada palabra de la query y también utilizarlo para crear el snippet.

La clase posee un método llamado `GetScore` que recibe un array con la query separada por palabras para pasárselo al campo correspondiente en esta clase, un array con los IDF de las palabras de la query también para pasárselo al campo correspondiente, y el array con los valores de cercanía. Primeramente, inicializamos algunas variables que serán necesarias como, por ejemplo, el array final que se va a devolver y una variable que va a ir almacenando por iteraciones la suma acumulada de TF x IDF.Luego con un bucle for más externo iteramos sobre la cantidad de documentos. Dentro de este utilizamos un for (foreach) interno para iterar por las palabras de la query. Utilizando condicionales verificamos si la palabra está en cada documento. Si está, entonces añadimos a `suma` (inicializada en cero) la multiplicación del TF de la palabra en ese documento por su IDF.En caso de que la palabra no esté en el documento, entonces restamos 10 veces la longitud de la query, a *suma*, para asegurar que los documentos que contengan más palabras de la query tengan mayor score. De esta manera, por ejemplo, si la query contiene 3 palabras y un documento tiene un score de -90, significa que no contiene ninguna palabra. (Se ha restado 3 veces –una por palabra- la longitud de la query multiplicada por 10) Así, cuando termine de iterar por las palabras de la query dividimos la suma entre el valor de cercanía (que será automáticamente 1 si el operador no está, o si una de las dos palabras no están en el documento) y copiamos el resultado en la posición i del array final y volvemos a hacer cero la variable *suma* para iterar sobre un nuevo documento en el bucle externo.

- **Operators:**

Esta clase contiene los métodos para que la búsqueda admita operadores. Todos los métodos reciben la query separada por palabras, pero sin normalizar.

Tiene un primer método `ItHasToBeInText` que devuelve el índice de la palabra en al array de palabras de la query que contenga $\wedge$ (hacer constar que el método trabaja sobre el hecho de que sea una sola palabra la que empiece con $\wedge$, en otro caso funcionará, pero solo correctamente para la primera palabra que contenga $\wedge$). Para ello, inicializamos una variable con valor -1 que será la que vamos a devolver. Iteramos sobre las palabras de la query y si una palabra empieza con $\wedge$, entonces copiamos su índice en la variable y termina la iteración. (Si la query tiene la palabra con $\wedge$ pero también la tiene sin el operador, el código también funciona, pues reconoce que son la misma palabra) Por tanto, el método devolvería -1 si ninguna palabra posee el operador. Si se usa el operador de cercanía el método analiza si es necesario variar el índice de la palabra con el operador $\wedge$.

El segundo método es `ItCantBeInText`, que funcionará análogamente al método anterior lo que sobre el operador !.

El tercer método es `ValueOfRelevance`, que devolverá un diccionario con cada palabra de la query asociada a su valor. Inicialmente cada valor será 1 (excepto la palabra con el operador !, pues en este caso la palabra no debe influir en los rankings), pero por medio de una iteración sobre las palabras de la query y una más interna sobre los caracteres de la palabra (si empieza con *. Si no, saltamos a la siguiente palabra), entonces multiplicamos por 10 ese valor por cada * que tenga la palabra delante.

El cuarto método es `Closeness` que devuelve en un array la cercanía de las dos palabras en cada texto si está el operador. Si no está, copia 1 a cada elemento del array.

¿Cómo cargar el repositorio?

La carga del repositorio de documentos tiene un funcionamiento bastante sencillo. Solamente se llama al método `CreateRepository` de la clase *Repository* en la línea anterior a la que ejecuta la aplicación en el archivo *Program* de la carpeta *Moogle Server*:

```
26  MoogleEngine.Repository.CreateRepository();
27  app.Run();
```

En este llamado al método, se llenan los campos de la clase *Repository* y se puede usar su información en las otras clases. Si esta clase no fuera estática habría que crear un objeto en vez de llamar a la función y entonces los campos quedarían guardando la información de ese objeto y no habría forma de llamarlos en otras clases fuera de *Moogle Server*.

Funcionamiento en tiempo de búsqueda:

El funcionamiento en tiempo de búsqueda del programa está regido por la clase *Moogle*. En esta clase se recibe la query como parámetro del método `Query` y se trabaja con ella.

Primeramente, creamos dos arrays donde vamos a guardar la query separada por palabras. En el primero las guardamos normalizadas manteniendo los operadores (solo lo utilizaremos para los operadores) y nos aseguramos de no tener palabras repetidas; y en el segundo las guardamos ya normalizadas. Guardamos además en variables auxiliares (`ObligatoryWordIndex` ($\wedge$), `NotContainsWordIndex` (!), `RelevanceValues` (*), `ClosenessValues` ($\sim$)) los valores que resultan de evaluar cada método de la clase *Operators*. También creamos un array para guardar los valores de IDF de cada palabra de la query, un diccionario para relacionar más adelante los scores con el documento correspondiente, un diccionario para relacionar cada palabra de la query y la cantidad de veces que se repite en la misma, y también el array `wordsQuery`, con las palabras de la query normalizadas, pero sin repetir. Luego con un foreach llenamos el diccionario poniendo cuántas veces se repite cada palabra en la query.

En un segundo momento, se crea un primer bucle for para calcular los IDF de las palabras de la query. Para ello se utiliza una condicional, de manera que, si la palabra está en algún documento, entonces calculamos su IDF y lo multiplicamos por su valor de relevancia (dado por el operador *) y también por la cantidad de veces que está la palabra en la query. En caso contrario pues mantenemos el cero.

Luego de tener los valores de cada palabra de la query, entonces se crea un array `score` donde se va a guardar el ranking de los documentos utilizando el método `GetScore` de la clase *RankingVector*. Después, con un pequeño bucle for, le asociamos a cada valor almacenado en `score` el documento asociado a él (puede haber varios con un mismo valor de score, por eso el diccionario recibe como valor una lista). Luego ordenamos el ranking de mayor a menor.

Una vez ordenado el ranking, entonces se crea una lista de objetos *SearchItem* (título, snippet, score) que representan cada documento a mostrar en pantalla. Luego existe una condición por si no existen resultados de búsqueda:

```
81      /* Si el mayor score posee ese valor significa que no hay resultados, pues se restó 10* words.Length
82      cada vez que no se encontró una palabra de la query. O bien si todo es 0, es porque es una palabra
83      que está en todos los documentos y no nos interesa*/
84      if ( (score[0] == -10 * wordsQuery.Length * wordsQuery.Length) || ((score[0] == score[score.Length -1]) && score[0] == 0) )
85      {
86          items.Add (new SearchItem("No hay resultados", "El repositorio no contiene la búsqueda deseada", 0F));
87          return new SearchResult(items.ToArray(), query);
88      }
```

En caso de que sí existan resultados, entonces eliminamos primero los valores repetidos en `score` (pues no son necesarios, ya que si existen repetidos ya uno solo está asociado a una lista con todos los documentos que poseen dicho ranking), y luego se procede a colocar los diez documentos con más coincidencias con la búsqueda realizada por el usuario, en la lista mencionada en el párrafo anterior. Esto se muestra a continuación:

```
        /* Iteramos sobre los primeros 10 resultados de la búsqueda, que son los que imprimiremos.
        En caso de que el repositorio contenga menos de 10 documentos, la iteración se detiene al
        llegar al último. */
        int stop = 10;
        for (int i = 0; (i < stop) && (i < score.Length); i++)
        {
            // En cada iteración nos aseguramos que exista algún resultado más.
            if (score[i] != -10 * wordsQuery.Length * wordsQuery.Length)
            {
                // Iteramos sobre la lista de documentos asociada al score.
                foreach (string title in rankingDocs[score[i]])
                {
                    if (rankingDocs[score[i]].IndexOf(title) < stop)
                    {
                        // Obtenemos la posición del documento en el repositorio.
                        int index = Array.IndexOf(Repository.titles, title);

                        // Condición por si está el operador ^.
                        if (ObligatoryWordIndex != -1 && !Repository.filesWords[index].Contains(wordsQuery[ObligatoryWordIndex])) {
                            stop++;
                            continue;
                        }

                        // Condición por si está el operador !
                        if (NotContainsWordIndex != -1 && Repository.filesWords[index].Contains(wordsQuery[NotContainsWordIndex])) {
                            stop++;
                            continue;
                        }

                        // Con la función GetSnippet de la clase Document obtenemos el snippet.
                        string snippet = Document.GetSnippet(index);

                        // Agregamos a la lista el resultado encontrado para ser impreso en pantalla.
                        items.Add(new SearchItem (title, snippet, score[i]));
                    }
                }
            } else break;
```

Para finalizar, nos aseguramos de que la lista de objetos *SearchItem* contenga al menos 1. Si es así pues ya termina prácticamente el método. En caso contrario, se le añade lo siguiente y se muestra en pantalla:

```
        /* Si "items" está vacío es porque se emplearon operadores y nigún documento cumple con las
        especificidades. */
        if (items.Count() == 0) items?.Add(new SearchItem("No hay resultados", "Pruebe eliminar las especificidades", 0F));
        return new SearchResult(items?.ToArray(), query);
    }
```