



Universidad de La Habana

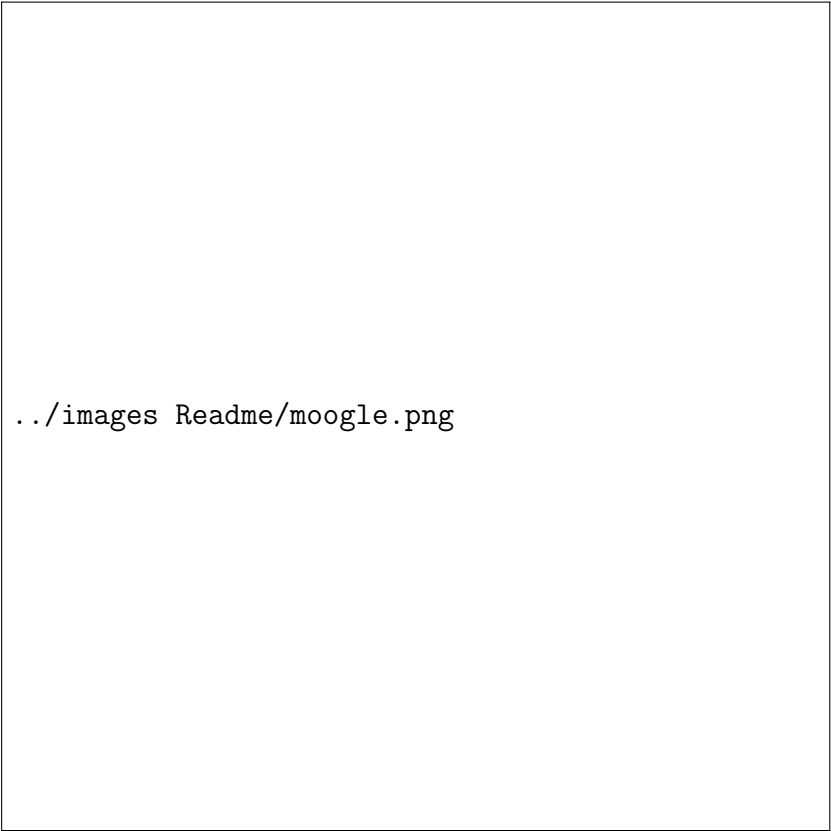
FACULTAD DE MATEMÁTICA Y COMPUTACIÓN

MOOGLE!

Proyecto de Programación I

Autor:
Melissa Maureen Sales Brito

2023



```
../images Readme/moogle.png
```

Moogle! es una aplicación totalmente original cuyo propósito es buscar inteligentemente un texto en un conjunto de documentos. Es una aplicación web, desarrollada con tecnología .NET Core 6.0, específicamente usando Blazor como *framework* web para la interfaz gráfica, y en el lenguaje C#.

La aplicación está dividida en dos componentes fundamentales:

- **MoogleServer** es un servidor web que renderiza la interfaz gráfica y sirve los resultados.
- **MoogleEngine** es una biblioteca de clases donde está implementada la lógica del algoritmo de búsqueda.

Necesitas tener instalado algún navegador y tener a **dotnet** disponible, o equivalente a esto tener .NET CORE 6.0 instalado, para ejecutarlo te debes ubicar en la carpeta del proyecto y ejecutar en la terminal de Linux:

```
make dev
```

Si estás en Windows, debes poder hacer lo mismo desde la terminal del WSL (Windows Subsystem for Linux), el comando ultimate para ejecutar la aplicación es (desde la carpeta raíz del proyecto):

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

- Abrir en el navegador `localhost:5000/`
- Introducir la búsqueda en la “entrada” y presionar el botón ‘`Buscar`’
- En primer lugar, el usuario puede buscar no solo una palabra sino en general una frase cualquiera.
- Si no aparecen todas las palabras de la frase en un documento, pero al menos aparecen algunas, este documento también será devuelto, pero con un score menor mientras menos palabras aparezcan.
- Algunas palabras excesivamente comunes como las preposiciones, conjunciones, etc., serán ignoradas por completo ya que aparecerán en la inmensa mayoría de los documentos. (En este sentido se recomienda un content de archivos en un mismo idioma.)

Nota: el uso de los siguientes operadores debe ser igual al de los ejemplos, para cada palabra que se desee negar/fijar/priorizar.

- Un símbolo `!` delante de una palabra (e.j., `hardin !tessa`) indica que esa palabra no debe aparecer en ningún documento que sea devuelto.
- Un símbolo `^` delante de una palabra (e.j., `hardin ^tessa`) indica que esa palabra tiene que aparecer en cualquier documento que sea devuelto.
- Cualquier cantidad de símbolos `*` delante de un término indican que ese término es más importante, por lo que su influencia en el score debe ser mayor que la tendría normalmente; este efecto será acumulativo, (e.j., `pan` con `**chocolate` indica que la palabra `chocolate` tiene dos veces más prioridad que `pan`).

La clave del algoritmo es el empleo del TF-IDF y el Modelo Vectorial el cual se explica detalladamente en el `Readme.md` del proyecto.

Estructura y funcionamiento:

```
class Utils
```

Esta clase contiene 4 métodos:

EnumerateFiles: como su nombre indica emplea la función `Directory.EnumerateFiles(string)` que devuelve una colección enumerable de nombres completos (con sus rutas de acceso) para los archivos en el directorio especificado por la ruta de acceso absoluta o relativa al directorio que se va a buscar. Esta cadena no distingue entre mayúsculas y minúsculas. Dicha colección es convertida en array.

TokenizeWord: recibe una palabra y la normaliza utilizando `Char.IsLetterOrDigit` que indica si un carácter Unicode pertenece en alguna categoría de letras o dígitos decimales.

LoadDocuments: emplea la función `Directory.GetCurrentDirectory()` que devuelve una cadena que contiene la ruta de acceso absoluta del directorio de trabajo actual y no termina con una barra diagonal inversa (`\`), se le pasa la ruta de nuestro contenedor de archivos. Este método retorna un llamado al método `Engine` de la clase `VectorialModel`.

Sort: método para ordenar los objetos de tipo `SearchItem`, que representan documentos coincidentes con la consulta, según su score final disponiéndolos de forma descendente.

`class DataFile`

Cuenta con las siguientes propiedades:

- **Name:** nombre del documento.
- **Root:** ruta del contenedor de documentos.
- **tfs:** diccionario cuyas llaves son las palabras de un determinado documento, asociadas a su TF.
- **NumWord:** total de palabras de un documento en específico.

En el constructor de la clase primeramente se guarda la ruta, se inicializa el diccionario, seguido se extrae el nombre del documento. Luego se lee el documento empleando el método `Stream Reader`. Añado a un array cada palabra del documento utilizando el método `String.Split`, hecho esto se llama al método `TokenizeWord` de la clase `Utils` para normalizar cada palabra.

Hallo el TF según la fórmula, calculando la frecuencia de cada palabra en el documento y luego dividiendo entre la cantidad de palabras del documento (`NumWord`), este valor será añadido al `Diccionario tfs`.

`class DataFolder`

- **Files:** array del tipo `DataFile`, es decir, que contiene las instancias de los documentos y las propiedades de estos.
- **idfs:** Diccionario con los IDF de cada palabra con respecto a la totalidad de documentos del contenedor.
- **norma:** propiedad que permite acceder al vector normalizado de un documento.

Se inicializa el diccionario en el constructor, luego se obtiene la ruta de cada documento llamando al método `EnumerateFiles` de `Utils`. Hallo el IDF de cada palabra calculando la frecuencia de la misma en la totalidad de documentos y luego el logaritmo natural del total de documentos dividido entre la frecuencia de la palabra.

Posteriormente procedo con la norma de cada documento, accedo a las palabras y sus tfs en dicho documento, para calcular la relevancia que sería el producto TF-IDF y finalmente la raíz cuadrada de la sumatoria de las relevancias de las palabras de ese documento, cada una al cuadrado.

Además cuenta con dos métodos:

- **TFS:** devuelve una matriz con los tfs dado un array de palabras que serán las de la consulta, cada fila corresponde a un documento y cada columna a una de estas palabras.
- **IDFS:** devuelve un array de idfs dado un array de palabras que serán en nuestra funcionalidad, las palabras de la consulta.

`class Engine en VectorialModel`

Primeramente, se declara e inicializa la propiedad `Folder` de tipo `DataFolder`.

Implementa varios métodos:

- **TokenizeQuery:** procede con el mismo tratamiento que la normalización de las palabras de un documento, pero en este caso para el arreglo de palabras de la consulta.
- **BuildDocumentsVectors:** dada la matriz con los tfs de la consulta y el arreglo con los idfs, este método devuelve una matriz con la relevancia de las palabras de la consulta.

- **QueryTf**: dadas las palabras de la consulta y el diccionario asociado al operador de prioridad de la clase `OperatorsQuery`, se calcula el tf de cada palabra que por conveniencia adopta la forma del inverso del total de palabras de la consulta y además aumento el numerador tantas veces como prioridad tenga la palabra.
- **QueryNorm**: devuelve la norma del vector consulta.
- **VectorialModel**: dados el vector documento y el vector de la consulta se implementa la fórmula para la similitud del coseno, el numerador es la sumatoria del producto escalar de los vectores y el denominador es el producto de las normas. De esta forma se obtiene el score de cada documento.
- **Snippet**: devuelve dado un documento un fragmento de este. Será el resultado que arroje la búsqueda.
- **Query**: Este método devuelve un objeto de tipo `SearchResult`. Este objeto contiene los resultados de la búsqueda realizada por el usuario. En particular llama a los métodos: `TokenizeQuery`, `WordsYes`, `WordsNot`, `WordsPriority`, `IDFS`, `TFS`, `BuildDocumentsVectors`, `QueryTf` y `VectorialModel` respectivamente. Finalmente se multiplican los scores por el valor devuelto por `NeedWords` y `RemoveWords`, si es mayor que 0 busco su snippet. Ordeno con `Sort` los scores y doy los resultados según similitud con la consulta.

`class Operators`

- **WordsNot**: devuelve una lista con las palabras que según la aparición del operador ! no deben ser devueltos documentos que las contenga.
- **RemoveWords**: retorna 0 si el documento contiene la palabra para luego en la clase `Engine` multiplicar el score de ese documento por 0 y que no sea devuelto como similar.
- **WordsYes**: devuelve una lista con las palabras que según la aparición del operador ^deben ser devueltos documentos que las contenga.
- **NeedWords**: retorna 0 si el documento no contiene la palabra para luego multiplicar el score de ese documento por 0 y que no sea devuelto como similar.
- **WordsPriority**: calcula la cantidad de * que le dan prioridad a una palabra en la consulta y guarda este valor en un diccionario que luego es devuelto asociando las palabras con prioridad a la cantidad de * que tenían.