oogle!

Manual de Usuario e Informe del Proyecto de Programación Moogle! del curso 2022

Daniel Machado Pérez

Estudiante de Licenciatura en Ciencia de la Computación, Facultad de Matemática y Computación, Universidad de La Habana

daniel.machado@estudiantes.matcom.uh.cu

Resumen

En el año 2022, como parte de la asignatura de Programación de Primer Año de la carrera Ciencia de la Computación de la Universidad de la Habana, fue orientado un proyecto a realizar por los estudiantes como parte de su evaluación del curso. Nuestra misión (si deseábamos aceptarla) consistía en implementar la lógica del motor de búsqueda **Moogle!** (sí, el nombre es así, con ! al final), aplicación *totalmente original* cuya función es buscar inteligentemente un texto en un conjunto de documentos.

Palabras Clave: TF-IDF, Similaridad de Cosenos, Modelo Vectorial, Query, Score, Distancia de Levenshtein, Content.

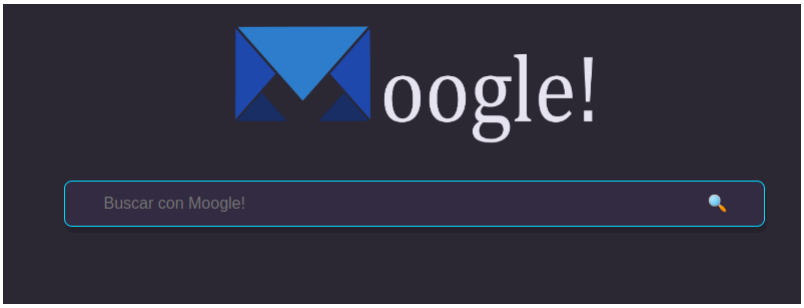

Figura 1: Barra de Búsqueda

1. Manual de Usuario

Para utilizar el motor de búsqueda **Moogle!** el usuario debe introducir un texto en la barra **"Buscar con Moogle!"** y presionar **Enter** o hacer **click** en el botón **"Buscar"**. Dicho texto debe constituir la **Query** a partir de la cuál se ofrecerán resultados ordenados de forma descendente según un coeficiente **(score)** de relevancia, calculado durante el proceso de la búsqueda mediante un método matemático llamado **Similaridad de Cosenos**. El usuario tiene la posibilidad de utilizar 4 operadores para ejecutar una búsqueda más inteligente. Estos influirán sobre los resultados, modificando el score o desechando ciertos documentos que no cumplan con lo exigido.

Estos operadores son:

- **Operador de "no presencia de la palabra" (!)**: Este operador se debe utilizar escribiendo el caracter **(!)** inmediatamente antes de la palabra sobre la que se va a ejecutar, sin un caracter en blanco por el medio **(!palabra)**. Garantiza que no se muestre ningún documento en el que esté *presente* la palabra señalada.

- **Operador de "presencia de la palabra" (^)**: Este operador se debe utilizar escribiendo el caracter **(^)** inmediatamente antes de la palabra sobre la que se va a ejecutar, sin un caracter en blanco por el medio **(^palabra)**. Establece la obligatoriedad de que la palabra señalada esté *presente* en los documentos resultados.

- **Operador de "cercanía" (∼)**: Este operador se debe utilizar escribiendo el caracter **(∼)** inmediatamente después de la primera palabra sobre la que se va a ejecutar e inmediatamente antes de la segunda palabra, sin un caracter en blanco por el medio **(palabra1∼palabra2)**: Calcula la *cercanía* de ambas palabras en los documentos (en los que aparezcan ambas a la vez) y aumenta el *score* de dichos documentos, ordenados según la *cercanía* (mientras más cercanas las palabras, más aumenta el score).

- **Operador de "importancia" (*)**: Este operador se debe utilizar escribiendo el caracter **(*)** inmediatamente antes de la palabra sobre la que se va a ejecutar, sin un caracter en blanco por el medio **(*palabra)**. Modifica la *importancia* de una palabra en una búsqueda, aumentando su **TF-IDF**(por sus siglas en inglés, Frecuencia del Término e Inverso de la Frecuencia de los Documentos). Puede escribirse más de un caracter **(*)** delante de la palabra, lo que denota la cantidad de veces que se aumenta la *importancia* de la misma. (n caracteres **(*)** delante de una palabra, aumenta su *importancia* n veces).

Si el usuario realiza una búsqueda en la que alguna de las palabras de la **Query** no se encuentra en ningún documento (no se encuentra en el **Content**), el programa buscará la palabra más cercana

presente en los archivos mediante un método matemático llamado **Distancia de Levenshtein**, y ofrecerá una sugerencia de posible **Query** *arreglada*. Se mostrarán resultados para dicha sugerencia.

En cada búsqueda se muestra el número de resultados y el tiempo empleado en el proceso.

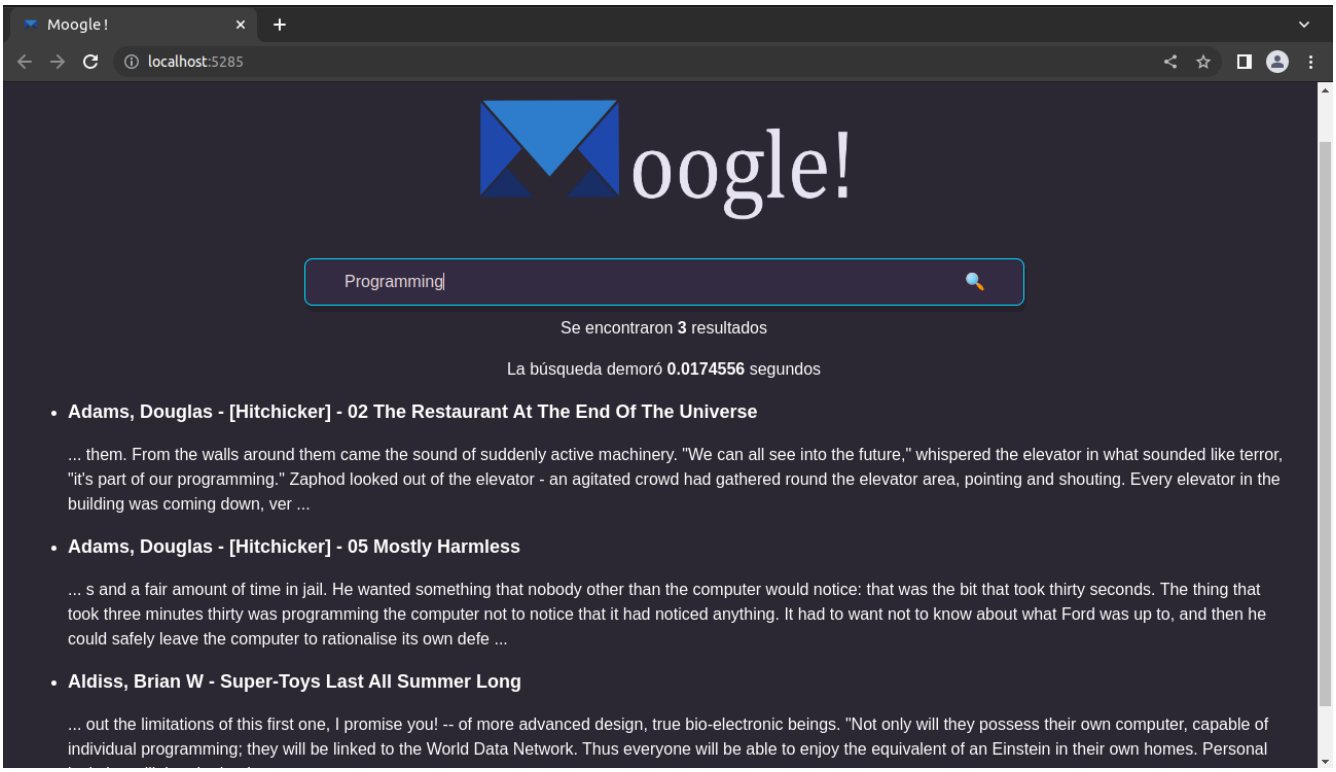

Figura 2: Menú de Búsqueda

2. Acerca de la ejecución del proyecto

2.1 Estructura

El proyecto consta de dos componentes principales:

- **"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.

"MoogleEngine" está constutuida por siete clases, cuatro de ellas fueron creadas durante la realización del proyecto. La clase **SearchItem** se mantiene con el diseño original. Las clases **SearchResult** y **Moogle** fueron modificadas, siendo esta última la clase principal. A **SearchResult** se le añadió un método **RepairSuggestion()** que elimina la sugerencia ofrecida, una vez sustituida por el usuario. Completamente nuevas son **DocumentProcess**, **ModelSpaceVector**, **LevenshteinDistance** y **SearchOperators**.

Moogle: 1 método [Query()]

DocumentProcess: 4 métodos [LoadData(), Normalize(), TF-IDF(), SnippetMaker()].

LevenshteinDistance: 4 métodos [NotFoundWord(), EditDistance(), FixedWords(), VectorQueryFixed()].

ModelSpaceVector: 10 métodos [SelectVector(), NormalizeVectors(), DotProducts(), ModuleVector(), ModuleProducts(), SortCosines(), CosinesAfterNearbyWords(), VectorsAfterHighlightWords(), CosineSimilarity(), MostImportantWord()].

SearchOperators: 5 métodos [WordsWithOperator(), UnwishedWords(), NecessaryWords(), NearbyWords(), HighlightWords()].

2.2 Flujo de Datos

2.2.1 Preprocesamiento

Primeramente ocurre el preprocesamiento de los datos del **Content** que son independientes de la búsqueda, es decir, se ejecuta el método **LoadData()** de la clase **DocumentProcess**, que inicialmente carga todos los textos del **Content** y luego hace un llamado a los métodos **Normalize()** y **TF-IDF()** de la misma clase y en ese orden. El método **Normalize()** procesa todos los archivos, recorre el **string** del texto del documento, forma las palabras, guarda en una variable **WordIndex** de tipo $\boldsymbol{List < Dictionary < string, List < int >>>}$ los índices de su posiciones en el **string**, donde cada elemento de la **Lista** "de afuera" es un **Diccionario** que corresponde a un documento, que a su vez contiene todas ls palabras del documento sin repetición como parte de los **Keys** relacionada con una **Lista** de índices de dichas palabras en el texto. Esta variable será usada posteriormente para el trabajo con el operador de *cercanía* y la elaboración de los **Snippets**. En otra variable se guarda la cantidad de veces que se repiten las palabras dentro del mismo documento. Luego, el método **TF-IDF()** se encarga de calcular y guardar para cada palabra su coeficiente de *importancia* utilizando la fórmula

$$\frac{\boldsymbol{nd}}{\boldsymbol{Cd}} \cdot \log\left(\frac{\boldsymbol{T}}{\boldsymbol{N}}\right),$$

donde
$\boldsymbol{nd}$ es la cantidad de ocurrencias de una palabra en un documento,
$\boldsymbol{Cd}$ es la cantidad total de palabras en el documento,
$\boldsymbol{T}$ es la cantidad total de documentos,
$\boldsymbol{N}$ es la cantidad de documentos en los que aparece la palabra.

2.2.2 Procesamiento de la Query

A continuación se realiza una primera búsqueda para detectar las posibles palabras que no se encuentran en ningún documento del **Content**. Para ello se invoca el método **NotFoundWord()** de la clase **LevenshteinDistance**. En caso de que fueran encontradas todas las palabras, para calcular su **TF** se ejecuta el método **TF-IDF()** con la **Query** normalizada como parámetro. Si existen palabras que no fueron encontradas, se llama al método **VectorQueryFixed()** de la clase **LevenshteinDistance**, para obtener la **Query** *arreglada*, que se pasa como parámetro al método

TF-IDF(). **VectorQueryFixed()** llama al método **FixedWords()** y este a su vez a **EditDistance()** para obtener el término más *cercano* a cada palabra no encontrada en los textos. Una vez tenido esto **VectorQueryFixed()** se encarga de reelaborar la **Query** sustituyendo los términos no encontrados por los más *cercanos*. Inmediatamente se elabora la **Sugerencia** de búsqueda que será ofrecida al usuario.

2.2.3 Modelo de Espacio Vectorial

Después de esto, todo está listo para hallar los resultados de la búsqueda. Esto se hace a través del método principal de la clase **ModelSpaceVector**, **CosineSimilarity()**. Dentro del mismo se ejecutan varios llamados a otros métodos de la misma clase. Primeramente a **SelectVector()**, que se encarga de realizar un filtrado previo de los documentos, buscando los que contengan al menos una palabra de la **Query**, desechando los que contengan palabras *no deseadas* obtenidas con el método **UnWishedWords()**, y desechando también las que no contengan alguna de las palabras *necesarias* obtenidas con **NecessaryWords()**. Posteriormente se construyen representaciones de los documentos filtrados para ser tratados como vectores y poder realizar operaciones con ellos, mediante el método **NormalizeVectors()**. Ahora se modifican los coeficientes de *importancia* de las palabras obtenidas en **HighlightWords()**, sumándole la cantidad ($\boldsymbol{k}$) de caracteres **(*)** asociados a la palabra. Esto se realiza en el llamado al método **VectorsAfterHighlightWords()**. Seguidamente se procede al cálculo de los cosenos entre cada vector resultado y la **Query**. Primero se calculan los **Productos Punto** mediante **DotProducts()**, y luego los **Productos de los Módulos** mediante **ModuleProducts()**, que a su vez utiliza el método **ModuleVector()** para calcular el módulo de cada vector. La división entre estos dos resultados es la que devuelve los cosenos entre vectores. La fórmula sería

$$\cos(\boldsymbol{A}, \boldsymbol{Q}) = \frac{\boldsymbol{A} \cdot \boldsymbol{Q}}{|\boldsymbol{A}||\boldsymbol{Q}|},$$

donde A es un vector que representa un documento y Q es el vector que representa la **Query**.

Luego, estos resultados de los cosenos se modifican atentiendo al operador de *cercanía* a través del método **CosinesAfterNearbyWords()**, que suma un número $\boldsymbol{p}$ a cada coseno que involucre un documento donde se encuentren los pares de palabras de **NearbyWords()**, donde $\boldsymbol{p}$ es el lugar que ocupa el documento en la lista ordenada ascendentemente de los textos, atendiendo a la cercanía de las palabras en cuestión. Este método contiene uno de los algoritmos más complicados de la implementación lógica de la aplicación. Para facilitarlo se decidió la utilización de la variable **WordIndex**, aprovechando que contiene los índices ordenados que representan la posición de cada palabra dentro de cada documento. De esta forma lo que hace el algoritmo es, en un texto que contenga el par de palabras de **NearbyWords()**, mediante un ciclo, ir calculando diferencias entre índices de ambas palabras y guardando siempre la menor diferencia. Auxiliándose del orden de los índices, es posible evitar probar todas las combinaciones. Inicialmente se efectúa la resta entre los primeros índices de cada **Lista** y el algoritmo continúa avanzando por los siguientes índices de una u otra **Lista** en dependencia de cuál de los dos sea el menor. El ciclo termina cuando se llega al final de ambas **Listas**. Teniendo menor distancia para cada documento, se ordenan los documentos y entonces se suma el número $\boldsymbol{p}$ del que hablábamos anteriormente.

Es en este momento que se calculan los *scores*. La fórmula general sería la siguiente:

$$\boldsymbol{score} = \frac{\sum_{i=1}^{n}(\boldsymbol{a_i} + \boldsymbol{k_i})\boldsymbol{q_i}}{|\boldsymbol{A}||\boldsymbol{Q}|} + \boldsymbol{p},$$

donde $\boldsymbol{n}$ es la cantidad de palabras de la **Query**,
$\boldsymbol{a_i}$ es la componente i-ésima del vector documento (**TF-IDF**),
$\boldsymbol{q_i}$ es la componente i-ésima del vector **Query** (**TF**),
$\boldsymbol{k_i}$ es el coeficiente de *importancia* que se suma como resultado del operador de *importancia*.

Por último, utilizando el método **SortCosines()** se ordenan los resultados descendentemente atendiendo al *score* y esto es lo que retorna el método **CosineSimilarity()**.

2.2.4 Resultados

Una vez obtenidos los resultados ordenados, se elaboran los *snippets*. Para ello, primeramente se llama al método **MostImportantWord()** que retorna, para cada documento, de las palabras que coinciden entre la **Query** y dicho texto, la que tiene mayor coeficiente de *importancia*. Este resultado se pasa como parámetro a un método **SnippetMaker()** que lo utiliza para elaborar los *snippets*. Para concluir se elaboran los objetos de tipo **SearchItem**, se almacena la cantidad de resultados, se detiene el cronómetro y se retorna el objeto de tipo **SearchResult**.

Aquí termina la ejecución de una búsqueda. Seguidamente se muestran los resultados en la interfaz web y se espera la interacción del usuario. Si se realiza otra búsqueda se repite el proceso a partir de la ejecución del método **Query** de la clase **Moogle**.

Referencias

[1] Information retrieval document search using vector space model in R (https://www.r-bloggers.com/2017/11/information-retrieval-document-search-using-vector-space-model-in-r/).

[2] Reguera Villar, R. y Solana Sagarduy, M. *Geometría Analítica*. Editorial Pueblo y Educación, La Habana, 1982.