Informe sobre programa *Moogle!*

Sheila Roque Alemán

20 de junio de 2023

Introducción

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 se implementa la lógica del algoritmo de búsqueda.

El Algoritmo de Búsqueda

Para comprender la lógica detrás del algoritmo de búsqueda, primero se deben conocer algunos conceptos que estaremos utilizando. Explicaremos conceptos y procedimientos como:

- De qué se tratan los *Documentos de Búsqueda*
- Qué es un *Resultado de Búsqueda*
- Cómo se implementa el *modelo vectorial* de esta aplicación con una matriz de términos por documentos
- Qué es y cómo funciona el *TF-IDF*, además de cómo calcularlo
- Cómo influyen los *operadores* en los valores del vector del query, y cómo influyen luego en el cálculo del *score* posteriormente
- Cómo se calcula la *similitud* entre los vectores de documentos y la query
- Cómo se calcula el *snippet*
- Cómo se calcula la *sugerencia* y en qué casos decidir que aparezca en el *Resultado de Búsqueda*

Los *Documentos de Búsqueda*

Moogle! devuelve un conjunto/array de *Documentos de Búsqueda* que tengan mayor o menor similitud con la búsqueda ingresada.

Cada documento de búsqueda contiene:

- **Nombre:** el nombre del documento correspondiente.
- **Snippet:** un pedazo de texto del documento.
- **Score:** el valor de la similitud del documento con la búsqueda/query.

El *Resultado de Búsqueda*

Al conjunto de *Documentos de Búsqueda* que se devuelve se le denomina *Resultado de Búsqueda* . Este queda ordenado de mayor a menor por el valor del Score de los *Documentos de Búsqueda* que contiene.

Al *Resultado de Búsqueda* también se le añade una *Sugerencia* en caso de que los *Documentos de Búsqueda* no hayan sido suficientes.

El Modelo Vectorial

Para poder reunir y devolver toda la información de los documentos se ha implementado un modelo vectorial de recuperación de información. Este consiste en agrupar un conjunto de vectores que representan la cantidad de términos de cada documento, formando así una matriz de términos por documentos.

La Caché

Como el modelo de búsqueda va a ser siempre el mismo y lo único que cambiaría sería la query, entonces al cargar la matriz de términos por documentos siempre saldrá igual que la vez anterior. Por lo tanto, cargar todos los términos de los documentos en dicha matriz sería un desperdicio de recursos y de tiempo de procesamiento.

Aquí es donde conviene crear una **caché** donde se guarde toda esta información que relativamente no debe cambiar.

Por lo tanto, guardaremos esta matriz junto con los demás grupos de información, que se describirán más adelante, que no deberían cambiar (o al menos no lo harán frecuentemente)

*En caso de que se **modifique la cantidad de documentos** (se añadan nuevos documentos) o se **modifique el interior de alguno de ellos** (cosa que no es común que suceda), la mejor solución sería actualizar el servidor de la aplicación para que cargue la **nueva información***

Implementación del Modelo Vectorial

Para crear por primera vez la matriz del modelo vectorial se debe saber su tamaño. Conocemos cuántos documentos existen, pero falta conocer la cantidad de términos que van a contener.

Para esto se crea una variable que reunirá todos los términos que aparecen en cada documento. Se pudiera decir que esta variable será nuestro **diccionario de términos**, será el léxico que se estará utilizando para nuestra búsqueda de información.

Pero para calcular este diccionario de términos primero debemos recorrer todos los términos de todos los documentos. Para esto utilizaremos un array de diccionarios de *keys* enteros y *values* strings (ya que queremos que los términos se repitan según aparezcan en los documentos; además, los enteros son para determinar el orden en el que aparecen en cada documento), y ya que los necesitaremos más adelante, los guardaremos en la caché. Cada diccionario de este array representa el texto entero de su documento correspondiente (mantendremos el orden de documentos también en los diccionarios de textos).

Una vez que tenemos los diccionarios del texto de cada documento es mucho más fácil calcular todos los términos que aparecen en todos los documentos sin que se repitan. Para esto, esta vez los strings serán los *keys* del diccionario, y los enteros serán sus *values* que servirán para calcular la posición i-ésima de la matriz. Así formamos nuestro diccionario de términos *globales*. Como la cantidad de términos no debe variar, y si varía sería mejor actualizar la aplicación, entonces esta variable la podemos guardar como parte de nuestra caché.

Calculando el TF-IDF

Una vez obtenidos todos los términos podemos crear nuestra matriz, ahora solo falta implementar el modelo vectorial dentro de esta.

Para tener una mayor precisión con los términos más importantes y prácticamente obviar las palabras más utilizadas y menos relevantes, se implementa un TF-IDF (Term Frequency - Inverse Document Frequency). Este consiste en calcular la frecuencia apariciones del término en el documento y multiplicarla por el inverso de la frecuencia del término en todos los documentos (o sea, en cuántos términos aparece este término).

Para calcular la frecuencia del término en el documento utilizaremos el array de diccionarios de texto de cada documento. En este se guardaron todos los términos que aparecen en el documento y en el orden en que aparecen y, por supuesto, las veces que aparecen en el mismo documento.

Utilizando este diccionario para calcular la cantidad de veces que aparece cada término en cada documento y la cantidad de términos en total de cada documento, podemos proceder a calcular el TF:

Como la matriz de vectores de documentos en este caso particular es una matriz de filas de términos y columnas de documentos, primero iteramos por cada columna pasando por cada fila de esta, teniendo así el i-ésimo térnino en el j-ésimo documento. Una vez recorriendo cada término por cada documento, podemos calcular la cantidad de veces que se repite cada término en un documento, y luego dividirlo por la cantidad de términos en total que existen en dicho documento (ya que nuestro TF va a ser la frecuencia del término en el documento dividido por la cantidad de téminos en total del documento).

Así obtenemos el TF de cada término con respecto a cada documento guardado en la matriz en su casilla correspondiente. Ahora es momento de calcular el IDF.

Para el idf se necesita saber la frecuencia del término en el conjunto de documentos, la frecuencia de documentos. Esto se calcula fácilmente con la función auxiliar DF (Document Frequency) que aparece en el código, para poder completar la fórmula del IDF de un término.

Como ya tenemos guardado el TF de cada término dentro de cada documento en la matriz del modelo vectorial, para poder tener el TF-IDF en la matriz solo hace falta multiplicar TF*IDF. Así que es muy conveniente ir iterando por cada casilla de la matriz y multiplicar el valor guardado

por el IDF calculado del i-ésimo término en el j-ésimo documento.

El IDF se calcularía por la fórmula $\log(D/DF) + 1$, siendo D la cantidad de documentos en total y DF el Document Frequency que habíamos explicado anteriormente.

El Query

Lo siguiente será calcular el vector del query. Para esto utilizamos el diccionario de términos que guardamos anteriormente como léxico para saber el largo de nuestro vector, determinar si un término del query es parte del léxico o no (si no lo es puede significar que es una palabra que puede existir pero que no nos interesa porque no aparece en ningún documento, o probablemente es una palabra mal escrita) y, en qué posiciones guardar los valores de los términos que aparecen en el query. Calculando los valores del query con el algoritmo de Term Frequency (no necesitamos IDF porque el query es un único elemento; se pudiera considerar como si fuera un documento más, *el documento de búsqueda*) ya tendríamos todo listo para calcular la similitud vectorial entre el quey y los vectores columna de los documentos.

Los Operadores en el Query

Para enriquecer más la búesqueda se han implementado algunos operadores de búsqueda (como petición opcional de nuestro cliente). Estos operadores tienen diferentes funcionalidades y pueden modificar los valores de la búsqueda a petición del usuario. Los operadores implementados hasta ahora son:

- **!:** El término que suceda a este operador no deberá aparecer en ningún documento devuelto.

  Si algún documento que contiene a este término es similar al query por razón de similitud con otro término que contiene este documento, como contiene un término "baneado"se modificará su score para que sea 0 y así no se devuelva este documento en el Resultado de Búsqueda *.*

  Ej: algoritmos !ordenación (queremos buscar los documentos que hablen de algoritmos pero no queremos que aparezca el término ordenación en ellos

- **∧:** El término que sucede a este operador deberá aparecer en todo documento devuelto.

  Para esto solo hemos implementado el aumentarle el valor considerablemente a dicho término en el vector del query, para que todos los documentos que contengan dicho término aparezcan con mayor similitud que los que no lo contienen. Y por supuesto, si contienen al término necesariamente deben aparecer en el Resultado de Búsqueda *.*

  Ej: ∧algoritmos ordenación (queremos que aparezcan Documentos de Búsqueda *que contengan estos dos términos, pero le queremos dar mayor importancia a todos los que contengan al término que utiliza este operador).*

- **∼:** Los términos en los que este operador aparezca entre dichos dos términos deberán aparecer cercanos en el documento.

  Mientras más cercanos, mayor deberá ser el score de dicho documento que contenga a ambos términos.

  Ej: algoritmos ordenación (queremos que tengan un mayor score y por tanto aparezcan de primeros los Documentos de Búsqueda *que más cercanos tengan estos términos).*

- ***:** El término que suceda a este operador tendrá una mayor importancia a lo que normalmente tendría.

  Este operador es acumulativo, así que mientras más cantidad de operadores existan junto al término, mayor será su importancia. En este caso, se implementó el aumentar su importancia como duplicar el valor que tenía anteriormente el término; y mientras mayor sea la cantidad de operadores de este tipo, más veces se duplicará su valor (tendría una fórmula de $query[i]* = 2^k$*, donde* k *es la cantidad de operadores, e* i *es la posición del término en el vector del query).*

  *Ej: algoritmos **búsqueda (queremos darle una mayor importancia a este término para que nos aparezcan primero los* Documentos de Búsqueda *que principalmente contengan el término de mayor importancia).*

Puede añadirse algún otro operador en el futuro, que tenga una nueva funcionalidad que ayude a enriquecer la búsqueda deseada de forma más eficiente y cómoda para el usuario.

Siempre recordar que los operadores se colocan ***precediendo al término que los vaya a utilizar*** *(excepto el operador* $\sim$*, que utiliza dos términos y se coloca entre estos dos) y* ***sin espacios****.*

Cálculo de la Similitud Vectorial

Calcular la similitud vectorial de cada documento con la búsqueda consiste en calcular el coseno del ángulo que está entre el vector del query y el vector del documento seleccionado. Este valor nos sirve para calcular el ranquing de cada uno de los *Documentos de Búsqueda* , con unas pocas modificaciones en caso de que se hayan usado operadores en el query.

El cálculo de la similitud entre los dos vectores se calcula mediante la siguiente fórmula:

$$\cos(q, d_j) = \frac{\langle q, d_j \rangle}{\|q\| \|d_j\|}$$

Donde q *representa el vector del query y* d_j *representa el vector del* j*-ésimo documento.*
$\langle x, y \rangle$ *representa el producto escalar de dos vectores* x*, y* ($\langle x, y \rangle = x_1 y_1 + x_2 y_2 + \ldots + x_n + y_n$).
$\|x\|$ *representa la norma/longitud de un vector* ($\|x\| = \sqrt{\langle x, x \rangle}$).

Mientras más pequeño sea el ángulo entre los dos vectores, el coseno de dicho ángulo se acercará más al valor 1; y mientras más grande sea el ángulo, el coseno se acercará más al valor 0.

Luego el valor del coseno del documento j será el valor del *Score* del *Documento de Búsqueda* corresponidente a este documento (Solo se modificará un poco en caso de que los operadores tengan que hacer algún cambio en el *Score* de este documento).

Los *Snippets*

Una vez obtenidos los *Score* de cada *Documento de Búsqueda*, ya casi está todo listo para devolver un *Resultado de Búsqueda* válido. Solo faltaría crear todos los *Documentos de Búsqueda* que no posean un *Score* nulo, y de paso ordenarlos por el *Ranquing* del *Score*.

Para crear cada instancia de *Documentos de Búsqueda* debemos tener calculado un *Snippet* adecuado con respecto al query, además del título del documento y su score que ya los tenemos.

Como a los *Documentos de Búsqueda* cuyo score sea nulo no necesitamos incluirlos al array que queremos devolver, no será necesario calcular un snippet para estos (lo que nos agilizará el trabajo y nos ahorrará gastar recursos y procesamiento).

Se verifica una vez más el uso de los operadores para definir por completo el *Score* que se utilizará en el *Ranquing* de los *Documentos de Búsqueda* y luego se procede a calcular los *Snippets* de cada uno y agregarlos al array de *Documentos de Búsqueda* en orden decreciente con respecto a su *Score*.

Como inicialmente se crearía un array de *Documentos de Búsqueda* con el tamaño de la cantidad de documentos que existen, independientemente de si tienen relación con la búsqueda o no, es muy probable que algunos o muchos de estos obtengan un *Score* nulo. Luego, como queremos saber la cantidad exacta de *Documentos de Búsqueda* válidos y no queremos dejar ninguna casilla del array nula, entonces procederíamos a crear un array con la cantidad correcta de *Documentos de Búsqueda* que vamos a devolver y copiamos el contenido del array anterior en el nuevo para luego devolver este último.

La Sugerencia

Como ahora tenemos un array con la cantidad real de *Documentos de Búsqueda* que vamos a devolver, solo nos falta decidir si nuestro *Resultado de Búsqueda* debería llevar una sugerencia o no.

Cuando la cantidad de *Documentos de Búsqueda* del array sea menor que la cantidad mínima satisfactoria (digamos en este caso 5), entonces es conveniente utilizar una sugerencia para ayudar al usuario a introducir un query que sea más adecuado para obtener una mayor cantidad de *Documentos de Búsqueda* .

Para calcular la sugerencia se recorren todos los términos del query, sin prestar atención a símbolos extraños como los operadores o signos de puntuación. Luego, por cada término del query se calcula la distancia de Levenshtein con cada término de nuestro *diccionario de términos*, y se coloca el que más similitud tenga con el término que originalmente aparecía en el query.

El algoritmo de la Distancia de Levenshtein *consiste en calcular la cantidad de modificaciones que llevaría cambiar de una* palabra orígen *a una* palabra destino.

Futuros cambios

Como siempre, no toda aplicación está siempre completa, y siempre pueden existir algunos cambios que ayuden a hacer más óptimo su funcionamiento o que mejore la calidad de la experiencia del usuario o simplemente se agreguen nuevas funciones.

El programa hasta ahora es completamente funcional, pero se le pueden agregar nuevas funciones que mejoren la calidad de los resultados que genera. Estén pendientes de nuevas actualizaciones y mejoras en el sistema.