Reporte del Moogle!

Luis Alejandro Arteaga Morales

Julio, 2023

Resumen

Moogle! es un motor de búsqueda sencillo que corre desde el navegador y permite al usuario a realizar búsquedas de documentos en formato .txt que contengan un input en una base de datos local pequeña. Está implementado en C# y utiliza como criterio de búsqueda el TFIDF sobre los documentos y similitud de cosenos.

Keywords: motor de búsqueda, navegador, .txt, local, C#, TFIDF, similitud de cosenos.

1. Introducción

En el presente artículo se tocarán las ideas principales que hay tras la implementación de Moogle! así como un tour por sus funcionalidades básicas.

2. Ideas principales

La implementación del Motor de Búsqueda está basada en el modelo vectorial, tomando cada consulta(query) y cada texto como un vector de palabras de $\mathbb{R}^n$, representados con el valor de TF-IDF de cada palabra. Luego se emplea la similitud de cosenos para ver qué tan cerca de ser iguales están dos vectores. De esta manera se tiene una noción de qué documentos son los más similares a lo que buscamos.

3. Funcionalidades básicas

3.1. Búsqueda simple

Se le mostrará una lista de documentos que contengan las palabras que aparecen en su búsqueda. Si escribe una consulta vacía el buscador se lo informará. Se mostrarán primero los documentos que contienen palabras más raras (aparecen en la consulta pero aparecen en menos documentos). Las palabras que son demasiado comunes (aparecen en todos los documentos) no serán relevantes y por tanto ignoradas por la búsqueda

3.2. Operador !

Los documentos que contengan las palabras que tengan un signo ! delante en la consulta, no serán sugeridos.

3.3. Operador ^

Solo podrán ser sugeridos documentos que contengan las palabras que tengan un signo ^delante en la consulta.

3.4. Operador *

Cualquier cantidad de * delante de una palabra aumenta la relevancia de esta para la búsqueda

3.5. Operador ~

Para dos o más palabras unidas por ~, entre más cerca estén esas palabras en un documento más relevante será este.

3.6. Sugerencias

Si no se encuentra ningún resultado para su búsqueda se mostrará una sugerencia de lo que debería escribir en su consulta para tener resultados. Esta sugerencia está basada en palabras ya contenidas en los textos que sean lo más parecidas a la consulta inicial.

4. Estructura del proyecto

4.1. Estructura general

El proyecto está dividido en dos partes principales: MoogleServer, que contiene el servidor e implementación del Frontend del motor y MoogleEngine, que contiene la lógica que hay detrás del motor. Nos centraremos en describir esta última en el reporte. Durante el análisis algunos detalles de implementación serán omitidos. De lo contrario el documento sería demasiado extenso. Para ver estos se pueden revisar los comentarios en el código fuente del programa.

MoogleEngine está compuesto de las siguientes clases, que iremos desglosando en las secciones que siguen:

- Document
- DocumentCatcher
- SearchItem
- SearchResult

- TFIDFAnalyzer
- StringMethods
- Moogle
- Trie (para futuras mejoras del buscador)
- Matrix (por propósitos de evaluación de la asignatura)

4.2. Document

Representa objetos de tipo documento, con las propiedades que se utilizan de ellos para cada una de las consultas.

- public string Title(título del documento)
- public string Text (texto del documento)
- public string LowerizedText (texto en minúsculas, para solo preocuparnos por la semántica de los datos y no de la capitalización, esta propiedad es usada en la construcción de snippets para la búsqueda en la clase Moogle)
- public string[] Words (contiene un arreglo con todas las palabras del texto separadas, esta separación está dada por los caracteres que no constituyen letras o números, ej: signos de puntuación)
- public Dictionary < string, int > WordFrequency (contiene cuantas veces aparece cada palabra en el texto)
- public Dictionary < string, float > TF (representa el vector con los valores de TF para cada palabra, en el desglose de la clase TFIDFAnalyzer veremos que significa el TF y el IDF en cada documento)
- public Dictionary < string, float > TFIDF (vector de TF * IDF de cada palabra, estos son los vectores que utilizaremos en nuestra búsqueda)
- public Dictionary < string, List<int> > WordPos (posiciones de cada palabra en el documento para la implementación del operador ˜en la clase Moogle)

4.3. DocumentCatcher

Esta clase estática contiene los métodos y constantes necesarios para procesar los archivos y convertirlos a nuestra clase Document.

Métodos:

- public static string IgnoreASCII() genera todos los caracteres que ignoraremos al procesar los textos y actuarán como separadores de las palabras. Internamente recorre todos los caracteres ASCII y considera todos los que no son letras o números.

- public static string IgnoreASCIIQuery() este método funciona como el anterior pero no elimina los caracteres especiales para los operadores de búsqueda, véase: ! * ^ ~
- public static bool IsAlphanumeric(char c) comprueba si un carácter es letra o número (esta función realiza la misma función que char.IsLetterOrDigit y sólo permanece en el código por motivos de seguridad, en versiones posteriores puede ser refactorizada y eliminada).
- public static bool IsOperator(char c) comprueba si un caracter es uno de los cuatro operadores especiales (es una función de utilidad empleada en varios lugares del código).
- public static bool InvalidWord(string word) comprueba si una palabra es inválida o no basándose en si contiene letras o números y no símbolos extraños.
- public static Document[] ReadDocumentsFromFolder(string folder) acepta un directorio en forma de string como parámetro y devuelve un arreglo de Document, construido a partir de los .txt contenidos en este directorio. Esta es la función que utilizamos para construir los documentos sobre los que realizaremos la query.

Propiedades:

- public static char[] Delims contiene todos los caracteres que actúan como separadores para las palabras de los documentos (es construido a partir de IgnoreASCII())
- public static char[] DelimsQuery contiene todos los caracteres que actúan como separadores para la query (es construido a partir de IgnoreASCIIQuery())

4.4. SearchItem

Representa los elementos devueltos en la query, e implementa de la clase IComparable para poder ordenar usando Array.Sort() Propiedades:

- public string Title (título del documento devuelto)
- public string Snippet (fragmento de texto del documento que contiene al menos una palabra de la query)
- public float Score (puntuación del documento empleada para poderlos ordenar de mayor a menor relevancia)

Nota: El método CompareTo permite comparar dos SearchItem en base a sus puntuaciones.

4.5. SearchResult

Contiene los resultados de la búsqueda y una sugerencia de cómo podríamos obtener mejores resultados de la búsqueda (esta será la misma que la query si encontramos resultados relevantes)

- private SearchItem[] items (arreglo de SearchItem que contiene los resultados de la búsqueda)
- public string Suggestion (string que contiene una nueva query con la que podríamos mejorar los resultados)

4.6. TFIDFAnalyzer

Esta clase estática contiene los métodos necesarios para poder construir los vectores de TF-IDF necesarios para procesar las consultas(queries). Un vector de TF-IDF está formado digamos por N componentes(donde cada una representa a una palabra), que son números reales. Estos números reales que representaremos dan una cierta noción de que tan raras e importantes o no son cada una de las palabras que estamos procesando en nuestras queries y documentos, y están dadas por las fórmulas siguientes:

$$TF(Palabra, Documento) = \frac{cntPalabra}{cntDoc} \tag{1}$$

Donde $cntPalabra$ es la cantidad de ocurrencias de la palabra en el documento y $cntDoc$ es la cantidad de palabras en el documento

$$IDF(Palabra) = \ln \frac{Docs}{PalabraDocs} \tag{2}$$

Donde $Docs$ es la cantidad de documentos y $PalabraDocs$ es la cantidad de documentos donde aparece la palabra

Luego el TF-IDF es el producto de estas dos medidas. Notemos que si una palabra aparece en todos los documentos su IDF será $\ln 1 = 0$, y el producto del TF-IDF será 0 tambíen, por lo que la palabra no será relevante.

Este tipo de palabras no relevantes son llamadas StopWords y no nos interesan en nuestra búsqueda.

Propiedades:

- public static void CalculateDocumentsWithTerm() recorre todas las palabras del vocabulario y determina cuantos documentos contienen a cada una, esta información es almacenada en Dictionary < string,int > DocumentsWithTerm en Moogle.cs
- public static void CalculateIDFVector() calcula el vector de IDF para todos los documentos y lo almacena en IDFVector en Moogle.cs
- public static void CalculateTFVector(Document doc) calcula el vector de TF para cada documento
- public static void CalculateTFIDFVector(Document doc) una vez calculado el vector de IDF y los vectores de TF construye el vector de TFIDF para cada documento.
- public static float Norm(Dictionary < string, float > vec) recibe un Dictionary < string,float > vec, que no es más que un vector de TF-IDF y calcula su norma, que es el valor por el cual tenemos que dividir cada valor en el vector para tener un vector unitario, de cierta manera esto es parecido a la Fórmula de Pitágoras pero generalizada a N-dimensiones, ya que nos da algo así como el tamaño del vector.

Este método será útil a continuación cuando veamos ComputeRelevance(). Su valor es la raíz cuadrada de la sumatoria de los cuadrados de cada componente.

- public static float ComputeRelevance(Document query, Document doc) permite determinar que tan relevante es un documento. Para esto como ya tenemos representados los vectores de TF-IDF por decir así como vectores del espacio $\mathbb{R}^N$ entonces queremos ver que tan cerca de ser colineales están esos vectores, porque esto nos dice que tan cerca de ser iguales (de contener lo mismo) están, para ello usaremos la siguiente fórmula:

$$Relevance = \frac{v \times w}{\|v\| \times \|w\|} \tag{3}$$

para el cálculo del numerador recorreremos cada componente del vector de la query verificando que también esté en el documento y haremos producto de vectores con estos. Luego para el denominador usaremos la función Norm(). En la implementación se tiene cuidado con los denominadores nulos, para no causar indefiniciones y errores en tiempo de ejecución.

4.7. StringMethods

Esta clase estática contiene los métodos necesarios para poder construir la sugerencia en Moogle.cs cuando no encontramos resultados

- public static int EditDistance(string s1, string s2) devuelve la distancia entre dos strings que recibe como parámetro, es decir la mínima cantidad de caracteres que hay que insertar, remover o cambiar para que los dos strings sean iguales
- public static int LongestCommonSubsequence(string s1, string s2) devuelve la subsecuencia común más larga entre dos strings que recibe como parámetro.

Ambos métodos son calculados usando Programación Dinámica para lograr mayor eficiencia.

4.8. Moogle

Este es el núcleo de nuestro proyecto y donde procesamos las consultas a mostrar. Tiene un constructor estático el cual es accedido por Index.razor en MoogleServer para precalcular la información necesaria para las queries.

- static void Preproccess() es llamado por static Moogle() para precalcular la información de los documentos, construye los documentos y el cuerpo de vectores de TFIDF(corpus) utilizando los métodos de TFIDFAnalyzer, Document y DocumentCatcher.
- static void ResetGlobalVariables() es llamado cada vez que hacemos una nueva query para limpiar la información de la query anterior y poder calcular la nueva sin errores.

- static void CalculateNewRelevance() incrementa la relevancia de las palabras en el vector de TF-IDF de la query multiplicando por 2.0f por cada asterisco encontrado en el prefijo de la palabra.
- static void UpdateNewRelevance() actualiza la nueva relevancia en el vector de la query
- static void FindSearchResults(List < SearchItem > results) inserta a la lista de resultados los documentos más relevantes para la búsqueda. Considera los 4 operadores especiales. Recorre todas las palabras de la query y analiza todos los documentos, solo considerando aquellos que tienen palabras obligatorias (de haberlas) y que no tienen palabras prohibidas, sugiriendo los que mayor score tienen, y de todos estos en caso de existir el operador ˜en la query se incrementa la relevancia según las palabras cercanas. Luego se construye el snippet del documento y se añade a la lista. Se usan expresiones regulares (RegEx) para construir el snippet, osea para encontrar el fragmento de texto que contiene palabras de la query. En caso de no poder construir el snippet por algún símbolo extraño haberlo impedido, entonces no se sugiere.
- static string BuildSuggestion() devuelve un string que es la sugerencia de lo que debería ser escrito en el input para obtener un resultado en la búsqueda. Recorre todas las palabras de la query y las sustituye por la palabra con menor EditDistance, de haber varias, escoge la de menor LongestCommonSubsequence.
- static string BuildSnippet(string[] words, string originalText, string lowerizedText) construye el snippet a partir de un documento dado (recibe como parámetros las palabras del documento, el texto original, y el texto en minúscula) y recorre las palabras de la query, cuando encuentra una de ellas intenta mostrar 20 caracteres a la izquierda y 200 a la derecha y devuelve este intervalo en el texto original.
- static List < Tuple < string, string > > FindWordsToBeNear() construye una lista que relaciona que palabras tienen que estar cerca de otras para el operador ˜. Devuelve un objeto de tipo List<Tuple<int,int>>.
- public static SearchResult Query(string query) recibe un string query que es obtenido desde Index.razor y encuentra todos los documentos relevantes ordenados utilizando los métodos anteriores. Los devuelve como un SearchResult.

4.9. Trie

Esta clase se encuentra en el programa para poder ser usada en funcionalidades futuras. Ej: Sugerencia de palabras en tiempo real, búsqueda de palabras con el mismo prefijo (esta funcionalidad ya se encontraba en el programa pero tuvo que ser removida porque es demasiado costosa en memoria)

4.10. Matrix

Esta clase está en el proyecto solo para evaluar los elementos básicos de álgebra (multiplicación de matrices y demás)

5. Conclusiones

Ante cualquier duda visitar el repositorio público del proyecto en `https://github.com/Sekai02/moogle-1`

Índice