PROYECTO DE PROGRAMACION I

FACULTAD

DE MATEMATICA Y COMPUTACION UH

¿De que se trata?

Como parte del I proyecto de la carrera de programación de la carrera Ciencias de la Computación ,que es una especie de buscador dentro de una base de dato de textos en formato “.txt” y esta es la explicación de mi proyecto Moogle! El sitio cuenta con una barra de búsqueda que forma parte de lo que le usuario este dispuesto a buscar , y el programa muestra en la misma pagina el nombre de los documentos que contienen al menos una coincidencia con las palabras de la cadena insertada y debajo un fragmento del texto actual que se relacionada con la búsqueda; todo estos documentos organizados de mayor a menor relevancia de acuerdo con criterio sofisticados de algebra lineal y algoritmos de programación.

Aunque las especificaciones iniciales están en archivo readme.md (justo en la misma carpeta donde está este pdf) es valido resaltar que la carpeta MoogleServer contiene otra llamada Content , que será la que almacenara los textos y que el vocabulario general de estos a su vez será principalmente para dominio del idioma español , y en otra medida de idiomas derivados del latín y anglosajones (todo se debe al vocabulario del teclado de este programador)

MoogleEngine

MoogleEngine es el nombre de la biblioteca que en diferentes clases se encarga de la implementación de la parte no grafica del proyecto y de la lógica de los algoritmos y la matemática detrás de la búsqueda. Todas las clases de la misma están implementadas en C#. Este proceso se desarrolla en dos fases imprescindibles, una primera encargada de leer los documentos, trabajar sobre sus respectivos vocabularios y encapsular toda la información posible para trabajar con la futura cadena , esto se ejecuta antes de que la app cargue la pagina para facilitar en medida una dinámica búsqueda. Esto es gracias a que se llama la clase Principal en el archivo Program.cs de MoogleServer.

Para una mejor explicación del proyecto analicemos un análisis de las clases de la de menor complejidad a la de mayor

CreatedVocabulary.cs

Esta clase es la encargada de manipular los vocabularios de cada documento especifico implementando. Algunas de la implementaciones que realiza son:

-Localizar la ruta de los documentos, almacenar sus títulos y leerlos.

-Crear el vocabulario de cada documento y uno mas global escaneando los documentos para sustituir tildes, evitar signos no alfanuméricos y no almacenar palabras repetidas. Para encapsular esta información y mucha mas especifica de cada documento he creado una clase File y considerar los documentos como objetos de la misma.

Files.cs

Esta clase como hemos explicado es la encargada de encapsular en un objeto toda la información relativa a cada texto tales como:

-Índice del documento

-Titulo del documento

-Un diccionario donde a cada palabra del vocabulario se le hace corresponder la cantidad de veces que aparece.

-El cuerpo del texto

-El mismo texto pero normalizado (definimos normalizar en la próxima clase)

-Un string snippet que es un fragmento del texto

- Una variable Score que será el peso de cada documento con respecto a la entrada

-Un diccionario con el respectivo peso de cada palabra con el query

Normalize.cs

Es la clase encargada de a través de varios métodos normalizar las cadenas de texto para su posterior trabajo, digamos tales como sustituir tildes y otros acentos de otros idiomas , eliminar signos no alfanuméricos, y otras acciones tales como suprimir saltos de líneas, espacios en blancos para tener una alineación dentro de los textos que manipulamos. Ademas hay implementada un método para trabajar con los títulos que se van a imprimir en la pagina.

AplicatedMath.cs

Esta clase realiza trabajos matemáticos tales como calcular tf*idf , normalizar vectores , y una que calcula similitud. He aquí donde empieza el primer de los principales análisis.

El principal algoritmo de búsqueda que efectúo es el calculo de tfidf . En el enfoque Frecuencia de Termino(TF) las coordenadas del vector documento d_j es representado como una función de conteo de términos con la longitud del documento

TF es 0 si la palabra no aparece

Sino: TF= (numero de veces que se repite dentro del documento)/(longitud del documento)

. La idea básica de la frecuencia inversa IDF es calcular en proporción a lo documentos donde aparece el termino t_j con respecto al numero total de documentos . IDF (t_i) = log10(N/(n_i -0.5)) donde:

-N es el numero de documento de la colección

-n_i es la cantidad de documentos donde aparece el termino t_i.(-0.5 es usado para evitar tener log10(1)=0;

Luego el peso de la palabra i dentro del documento j será la multiplicación del TF de la palabra dentro del documento y el IDF del valor de la palabra dentro del corpus de documentos.

.EL método de normalizar vectores es para darle a los vectores una longitud de 1. A través de la formulas $\sqrt{(\Sigma X_i^2)}$ donde X_i son los valores del vector.

El método de similitud es para calcular un score para el documento en función del producto de ambos vectores normalizados , y aquellos vectores que den mayor valor es porque su similitud es mayor con la cadena inicial.

$s(A,B)= \Sigma(A_i * B_i)/(\sqrt{(\Sigma A_i^2)} * \sqrt{(\Sigma B_i^2)} +1)$ siendo A y B dos vectores siendo A_i y B_i los componentes de ambos vectores.

QueryWork.cs

Este método es para trabajar con el query(cadena inicial)

-ProcessQuery es el método de crear el vocabulario en función de las palabras que pertenezcan al vocabulario global de los documentos.

-VectorQuery es el método que crea el vector query como un diccionario donde a cada palabra del vocabulario antes sacado su valor TF-IDF de acuerdo: TF(la frecuencia de la palabra dentro del query) y el IDF(como la cantidad de documentos en el que aparece la palabra) ,este ultimo ya lo teníamos calculado. Este vector es normalizado seguidamente.

-Operator es un método publico que se encarga se realizar operaciones de acuerdo a la presencia de operadores que no son mas que signos tales como:

! Operador que advierte de que ningún documento con la palabra que le sucede debe ser devuelto. A toda palabra con ese operador se le sustituirá dentro del vector query su valor TF-IDF por el valor -1

^ es el operador opuesto al anterior , solo deben ser devueltos documentos que tengan la palabra que le sucede al signo. A toda palabra con ese operador se le sumara 100 a su TF-IDF

* es operador prioridad que determina que la palabra que le sucede al operador aumenta en relevancia tanta como signos tenga , al TF-IDF se le agregara un

valor equivalente a la cantidad de * por 1000.

Se usan esos números de potencias de 10 para evitar confusión cuando se implemente el método que calcula esa similitud

~ este es un operador cercanía y a diferencia de los anteriores no es valor de una palabra sino del documento , o sea que se almacene dentro de un array de dobles donde en la posición i estará el valor correspondiente al incremento final del documento i

Para este operador se busca dentro del texto del documento actual la menor distancia (distancia es la cantidad de palabras que separan dos palabras dadas dentro de un texto) entre las palabras separadas por ~ y el valor incremento del documento i será el valor de dividir 0.5/ (esa distancia menor).

-el método Snippet que cuenta dentro de un rango de 100 palabras donde más veces aparezca más palabras del query y lo guarda en el campo snippet de la clase file asociado por supuesto a cada documento

-Un método que ayuda al usuario a localizar las palabras que quería buscar dentro del snippet entre emogis.

Principal.cs

Es el método encargado de unificar todos esos métodos y encaminar el trabajo para almacenar en un array los objetos SearchItem organizados por su score

-En su constructor se ejecutan los procesos relacionados con la creación de la matriz de documentos con su TF-IDF calculado y encapsulada esperando a trabajar con el query de ahí que se implemente en una primera fase antes de que el usuario pueda visualizar la página. Donde el método CalculateTF-IDF realizara el trabajo de asignarle a cada palabra un TF-IDF

- Una segunda fase que si dependerá del query y que se implementará una vez que el usuario de al botón buscar. Se crea el vocabulario del query; se calcula la similitud con cada uno de los vectores de la matriz de documentos pero no sin antes tener en cuenta los valores relativos a cada operador, si durante el proceso encuentro una palabra con TF_IDF= -1 hay que salir del documento actual, si me encuentro una con valor mayor a 100 pero no aparece en el vocabulario del documento actual pues también hay que salir del documento(se exige de la presencia de la palabra), En caso de si aparezca la palabra pues le quito 100 al valor TF_IDF de la palabra, y si la palabra tiene valor superior a 1000 pues cuento la cantidad de asteriscos y multiplico ese TF_IDF por esa cantidad.

Luego que están escogidos los documentos que pasan esa prueba de operadores se les calcula la similitud con el vector query y se suma al valor de la similitud de ese documento, el valor incremento realtivo al símbolo de cercanía ~ y la cantidad de palabras del vocabulario del query que coinciden con el vocabulario del documento. Se organizan los objetos de acuerdo a ese score total y se guardan en un array como objetos de SearchItem para luego la clase Moogle llame a la otra SearchResult con ese array de objetos SearchItem y los imprima en la página.

Espero que os guste y quede a la altura de lo que buscaban, aunque pienso modificarlo más porque me ha inspirado bastante. Gracias.

.