Informe del Moogle en LaTeX

Pablo Gómez Vidal, C-111

Fecha de entrega:25/7/2023

Resumen

El Moogle es un motor de búsqueda de archivos en formato txt. de forma local, es el primer proyecto de programación. En caso de testearlo en un sistema operativo diferente de Windows se debe introducir el path de manera manual en el código en el script "Reader". En caso de editar algún txt o quitarle/adicionarle uno deberá volver a cargar el programa para que este se actualice. El proyecto en si tiene comentarios en diferentes lugares a petición de la tutora.

1. ¿ Cómo funciona ?

La idea para hacer mi Moogle la confeccioné de la siguiente forma: Lo primero de lo que me di cuenta fue que necesitaba un método para poder leer los .txt de mi carpeta Content, necesitaba esos valores para poder trabajar con ellos y poder empezar el proyecto como tal. Entonces creé un script nuevo llamado Reader donde creé una clase de igual nombre y usé una función específica para conseguirlo, usé (Directory.GetCurrentDirectory) para conseguir el path de mi carpeta content y (Directory.GetFiles) para conseguir los archivos y creé variables para poder trabajar luego con ellos como (string [ ] textos) que contiene a todos mis archivos ya una vez tenía esa primera parte resuelta necesitaba empezar a limpiar mi string texto para empezar a trabajar con él. Entonces cree un nuevo script llamado "tf idfz un primer metodito al que llamé "Tokenizar". Con el (Regex.Replace) me encargué de depurar el texto dejando solo los caracteres que necesitba y quitando cualquier carácter extraño, luego le hice .ToLower ya que ahí no necesitaba las mayúsculas y luego le hice .Split. Ya una vez tenía mi método para tener a "textos" limpio me encargué de hacer los métodos de TF e IDF y un 3cer método para calcular el TF * IDF, esa parte puede parecer un poco redundante, pero hace que se entienda a la perfección lo que hago. TF (Term frequency): El TF me da la cantidad de veces que aparece una palabra en un texto como un valor numérico; ósea si la palabra "Japón" aparece 4 veces en un texto de 40 palabras la palabra Japón tendría un valor de TF = 0,1. IDF (Inverse document frequency): El IDF tiene una fórmula para calcularse que es log10(cantidad de textos / cantidad de textos en los que aparece la palabra). Lo que hace el IDF es prácticamente quitarles puntuación a las palabras más comunes ya que si tenemos una palabra que aparece en casi todos lo textos y otra palabra que solo aparece en 1, la palabra más rara consigue una mejor puntuación, que .es lo que se busca. Ya una vez ya tengo un método donde puedo calcular el TF*IDF de las palabras no queda más que hacerlo. En este punto tuve muchas ideas sobre como tratar a mi string texto, hablé con mi tutora y me dio ideas

de como hacerlo y tal, pero en la concreta esta parte tenía que ser mía, lo que se ocurrió hacer fue convertir mis textos en diccionarios, específicamente dos tipos, un diccionario que contendría a todos mis textos y otro/s que contendrían a los textos por separados. Me decidí a trabajar con diccionarios porque ya conocía sobre ellos y porque consideré que no iba a ser lento el proceso de calcular el TF*IDF de cada palabra, además trabajar con matrices lo veía muy engorroso. Creé los diccionarios de la forma ¡string , float¿ya que quería tener a la palabra y a su TF*IDF al lado.

Entonces básicamente lo que hice fue crear un " for " que contenía a otro " for " que a su vez tenía dentro a 2 " if " y 2 " else ", uno de cada tipo para mi diccionario de todos los textos y el de textos específicos, el primer for me itera por los textos y el segundo me itera por cada palabra del texto[i]. Lo que hice fue básicamente llenarlos, si no contenían a la palabra que la adicionaran y le pusieran 1 al contador y en el diccionario de textos específicos lo que dije fue que en caso de volvérsela a encontrar le sumara 1 al contador (para ir sabiendo cuantas veces la tenía en ese texto).

Una vez tenía esto hecho creé otro diccionario.KeyCollection llamado keys y lo igualé a mi DiccionarioDeTodosLosTextos.Keys y creé un foreach y dentro del mismo creé un for con un if que lo que hace es que si mi diccionario de textos específicos contiene a la palabra (ContainsKey) le calculara su TF*IDF y de esta maravillosa forma fue que logré tener un diccionario para cada texto con la palabra y su TF*IDF al lado. Ahora este método que acabo de explicar se llama " initiate " y es el método que mas se demora en ejecutarse de todo mi programa (alrededor de 13 segundos con una base de 50 mb de txt). Este método lo volví estático y lo inicialicé en la parte visual del Moogle de forma que solo se carga una sola vez al inicio de correr el programa y ya luego las búsquedas las hace instantáneas. Luego de hacer esto me puse en función ya de mi query en el script Moogle.cs Antes de modificar el método que me dan por defecto creé uno antes llamado ContarVector que lo que hace es convertirme mi query en un diccionario así de esa manera puedo calcular su TF*IDF como se lo hice a los otros diccionarios. Ya luego dentro del método principal lo primero que hago es calcular el TF*IDF de mi query con un foreach y un if y un else dentro. Una vez hecho esto empecé a desarrollar mi método SimilitudCoseno que me fue un poco engorroso para entender como implementarlo en el código. Similitud de coseno se basa en una división donde mi numerador va a tener la multiplicación escalar de los vectores o multiplicación punto a punto y el denominador tiene a la longitud de cada vector multiplicadas, para hallar la longitud de cada vector se utiliza algo parecido al Teorema de Pitágoras pero con una buena abstracción ya que estamos hablando de n vectores pero el principio es el mismo que es la longitud del vector va a ser igual a la raíz cuadrada de la suma de los cuadrados de los elementos de cada vector, y la división de esos elementos de va a dar mi score de cada texto, donde mientras más cercano a 1 sea más similitud tendrá con el query. Una vez creo este método ya tengo todo listo para imprimir en pantalla mis mejores 5 documentos con respecto a la query (línea 52-94) del Moogle.cs. La idea fue que creé una lista para ir metiendo mis documentos por orden decreciente respecto al score, lo hice utilizando el .Add y el .Insert de la lista, una vez hice eso creé un array de tamaño 5 y le copié los primeros 5 documentos de mi lista.

2. Opcionales

2.1. Snippet

El snippet que creé es uno sencillo que funciona cogiendo la mejor palabra de la query con respecto a cada uno de los 5 textos, y devolviendo la primera aparición de esa palabra en el texto con un rango de aproximadamente 100 char, pero me encargué de que nunca cortara palabras haciéndole dos while, uno para cada rango, izquierdo y derecho, que decía que si cuando cayera en la posición 100 era un digito o una letra brincara a la siguiente posición o regresara una para atrás en dependencia si era izquierda o derecha, pero antes de eso declaré los lugares del texto donde podía aparecer mi mejor palabra de la query, al inicio, en medio o al final y como proceder en cada caso. Para coger mi mejor palabra solo tuve que meter un if dentro del primer foreach de mi método similitud coseno, fue una idea bastante buena ya que si no hubiese tenido que hacer un método para encontrarla.

2.2. Sugerencia

La sugerencia la hice con el algoritmo del cálculo de la distancia de Levenshtein el cual funciona de la siguiente forma:

	.	C	A	S	A
.	0	1	2	3	4
P	1	1	2	3	4
O	2	2	2	3	4
C	3	2	3	3	4
O	4	3	3	4	4

En este caso la distancia entre “casa” y “poco” es de 4 El algoritmo funciona de esa forma poniendo las palabras en una matriz y comparando: Si las letras son iguales se coge el valor de la diagonal y si las letras son diferentes se escoge el mínimo de los 3 valores que la rodean y se le suma 1, recorriendo por filas. Una vez hago este método creo otro método llamado sugerencia que se encarga de implementar el algoritmo, que lo que hace es comparar cada palabra de mi query con cada palabra de mi diccionario de todos los textos y hacerle el calculo de Levenshtein y yo lo que hago es irlo actualizando hasta quedarme con el más pequeño que será la palabra más parecida a la de la query.

2.3. Hipervínculo

Este opcional se lo hice yo a parte de lo que pedían con System.Diagnostics.Process con un método llamado Start.Info... el pedazo de código está en el Index.razor. No necesita mucha explicación, simplemente al tocar en el título del documento que resulte en la búsqueda se abrirá automáticamente el .txt que lo contiene