Informe escrito del proyecto de programación de 1er año de Lic. en Ciencia de la Computación

Nombre: Emrys Cruz Viera
Grupo: C111
Nombre del Proyecto: Moogle!
Enlace de GitHub del proyecto: https://github.com/emryscv/MoogleProject

Índice:

0. Notas.

-Nos referiremos a los métodos como “**Nombre_de_la_clase.Nombre_del_método()**”

1. Arquitectura básica del proyecto.

Durante el desarrollo del proyecto fueron implementados en su totalidad una serie de clases y métodos y fueron modificados otros tantos.

Creados por mi:
-Tools.cs
-Search.cs
-TextProcessor.cs

Modificados:
-Program.cs
-Moogle.cs
-Index.razor
-Index.razor.css

2. Flujo de ejecución del proyecto.

Preprocesamiento:

Una vez se ejecuta el proyecto se ejecutan una serie de métodos para preprocesar la información guarda en los documentos que están en la carpeta “**Content**” sobre los cuales se realizara la búsqueda mas tarde. En primer lugar se ejecuta el método “**Tools.LoadDocuments()**” que nos devuelve la información que hay en cada documento y las direcciones de estos. Luego con esta información ya utilizable, se crea una instancia de la clase “**TextProcessor**” que es en la cual esta implementado todo el procesamiento del texto, en el que se le da peso a las palabras usando la medida numérica “**TfIdf**”, y por último cargamos el diccionario de sinónimos mediante el método “**Tools.LoadAndCreateSynonymsDictionary()**” que nos devuelve un diccionario que relaciona cada palabra con sus sinónimos.

Búsqueda:

Cuando el botón "**Buscar**" es presionado se llama al método "**Moogle.Query()**" este a su vez verifica si esta "**query**" fue realizada con antelación en caso positivo de vuelve directamente el mismo resultado que ya fue calculado la vez anterior para evitar realizar los cálculos otra vez y en caso negativo crea una instancia de la clase "**Search**" que es en la cual se realiza la búsqueda, incluyendo el procesamiento de los operadores(en caso de aparecer alguno en esta "**query**") y los sinónimos. Luego de realizada la búsqueda en el objeto de tipo "**Search**" tenemos la información que necesitamos devolver(los documentos que satisfacen total o parcialmente, el "**score**" de cada uno ordenas de manera descendente por el "**score**" y la "**sugerencia de búsqueda**"). Para mostrar los documentos en pantalla es necesario un "**snippet**" el cual es hallado con el método "**Tools.FindSnippet()**". Por ultimo este resultado es guardado para en caso de repetirse esta búsqueda evitar realizar los cálculos otra vez y después se retorna el resultado de la búsqueda.

3. Funcionalidad de la clase "TextProcessor".

Al instanciar un objeto de tipo " **TextProcessor**" el constructor recibe como parámetros el contenido de los documentos a procesar.

Lo primero que se realiza es calcular la frecuencia de cada palabra en cada documento("**TF**") con el método "**TextProcessor.CalcTF()**" para lo cual lo primero que se hace es normalizar el documento con el método "**Tools.Normalize()**" para quedarnos solo con letras minúsculas y sin tildar y con los números además con el documento palabra a palabra, que luego sera utilizado para hallar el "**snippet**" relativo a cada "**query**". Durante este proceso iremos añadiendo cada palabra a nuestro vocabulario(conjunto de todas las palabras diferentes en nuestros documentos), asignándole un indice en la matriz en la cual se relaciona palabras con documentos.
y guardando la posición en la que aparece en el texto para luego poder hacer uso del operador "**~**".

Luego se procede a calcular el valor de "**TFIDF**" de cada palabra con respecto a cada documento con el método "**TextProcessor.CalcTFIDF()**". Primero es necesario calcular la frecuencia inversa de documentos para cada una de las palabras con el método "**TextProcessor.CalcIDF()**" que se calcula "$\mathbf{log_{10}((cantidad\ total\ de\ documentos)/(documentos\ en\ que\ aparece\ la\ palabra))}$".¿Por qué de esta forma? Pues mientras en más documentos aparezca la palabra es menos discriminante para nuestra "**query**" entrando un poquito mas en análisis matemático "**log10(x)**" tiene a 0 cuando x tiende a 1. En este caso "**(cantidad total de documentos)/(documentos en que aparece la palabra)**" tiende a 1, cuando la palabra aparece en mayor cantidad de documentos, entonces mientras en mas documentos aparezca la palabra, menor sera el valor de "**TfIdf**". Finalmente se multiplica el valor de "**Idf**" por el de "**Tf**" de cada palabra en cada documento y así culmina el preprocesado del texto.

4. Funcionalidad de la clase "Search".

Al instanciar un objeto de tipo "**Search**" el constructor recibe como parámetros la "**query**", los documentos procesados en un objeto de tipo "**TextProcessor**" y el diccionario de sinónimos. Lo primero que se hace es normalizar la "**query**" con el método "**Tools.Normalize**()" sin quitarle los operadores para luego poder guardar que operadores y en palabras aparecen en la "**query**" con el método "**Tools.FindOperator()**" y luego se vuelve a normalizar para quitar los operadores.

Para realizar la búsqueda se verifica que cada una de las palabras de la "**query**" exista en nuestro vocabulario. En caso positivo, al "**score**" de los documentos en los que esta aparece se le suma el peso valor de TfIdf de dicha palabra con respecto a este documento y se añade la palabra a la "**sugerencia de búsqueda**". En caso contrario se busca la palabra que lexicográficamente más se parece utilizando el algoritmo "**Distancia Levenshtein**" implementado en el método

“**Tools.EditDistance()**” el cual es llamado por el método “**Tools.ClosestWord()**” y se añade a la “**sugerencia de búsqueda**”. Luego se buscan los sinónimos de la palabra en cuestión y si aparecen en nuestro vocabulario su valor de “**TfIdf**” divido a la mitad(pues no son la palabra buscada sino una relacionada) es añadido al “**score**” de los documentos en que aparece. Durante este proceso también son aplicados, en caso de aparecer, los cálculos relativos al operador “*****”, los cuales son multiplicar el valor de “**TfIdf**” de la palabra y sus sinónimos por *“2^(cantidad de veces que aparece el operador)„*.

Luego son aplicados los cálculos referentes a los restantes tres operadores con el método “**Search.ApplyOperators()**”, llegado a este punto ya le fue otorgado su “**score**” a cada documento tal cual si no hubiera operadores a excepción de “*” en caso de aparecer. A partir de aquí en caso de existir operadores de los restantes tres tipos entonces se procede de la siguiente manera para el operador “**!**” se hace 0 el ¨**score**¨ del documento que contenga a la palabra que tiene el operador y con respecto a “^” se hace 0 el “**score**” del documento que no contenga a dicha palabra. Para el operador “**~**” se busca usando el método “**Tools.MinDistance()**” las distancias mínimas de las dos palabras relacionadas por el operador en cada uno de los textos y es almacenada para luego ser procesada. A continuación se busca el “**score**” máximo para esta query y se le suma a los documentos cuyo “**score**” no sea 0 el que le asigna la función de proporcionalidad inversa “ “***(máximo score)/ (distancia mínima)***” .

Por ultimo los documentos son ordenados de manera descendente por el valor del “**score**” y son eliminados del resultado los elementos cuyo “**score**” es 0.

5. Funcionalidad de la clase “Tools”.

En esta clase están agrupados los métodos en los que están implementadas las funcionalidades que necesitamos para complementar los dos flujos de ejecución importantes del proyecto “**el preprocesamiento**” y “**la búsqueda**”.

Métodos:

1. “**Tools.EditDistance()**”: en este método está implementado el algoritmo llamado “**Distancia de Levenshtein**”. La idea general es calcular cual es la menor cantidad cambios que hay que aplicar sobre una cadena de caracteres para que sea igual a otra, dichos cambios son: añadir, eliminar o sustituir una letra y todos tienen valor de 1, osea que dicha cantidad de cambios sera la distancia entre dos cadenas que es un numero entero.

2. “**Tools.Normalize()**”: El texto recibido es dividido en palabras, incluso si hubiera un error como (palabras,incluso), también estas dos son separadas. Luego por cada palabra iteramos sobre los caracteres que contiene, el caracteres es llevado a minúsculas y luego se comprueba: si es una vocal con tilde, se cambia por la correspondiente sin tildar, si es ‘ü’ se cambiar por ‘u’, si no es una letra o un digito, se desecha, si es una “**query**”, evitamos quitar los caracteres que representan a los operadores y por ultimo desechamos las palabras que después de normalizadas, no contengan ninguno de los caracteres que necesitamos.

3. “**Tools.ClosestWord()**”: se evalúa al “**Distancia de Leveshtein**” entre la palabra que tenemos y todas las palabras del universo de palabras que tenemos y devolvemos la que menor distancia tenga con esta.

4. “**Tools.MinDistance()**”: para cada documento, se itera por las posiciones de las dos palabras con las que vamos a trabajar empezando por la posición de la segunda palabra(la que aparece a la derecha del operador ‘~’) en el texto, se calcula la distancia entre la posición de esta palabra y las posiciones de la otra palabra que estén antes que esta. Luego

tomamos la primera posición de la primera palabra que es mas grande que la primera posición de la segunda palabra y repetimos el proceso anterior y así vamos alternando entre ambas palabras hasta llegar al final del texto y nos quedamos con la menor de esas distancias calculadas.

5. "**Tools.FindSnippet()**": la idea es quedarnos con el "**snippet**" que mas satisface a la "**query**". El "**score**"que se le da a cada posible "**snippet**" se calcula iterando por las palabras que contiene y comprobando si estas están en la "**query**", en caso positivo se añade al "**score**" el valor de "**TFIDF**" de la palabra en ese documento. Primero se calcula para las primeras 100 palabras del documento(en caso de que el documento tenga menos de 100 palabras el "**snippet**"es el propio documento) ,se calcula su "**score**" y luego se va desplazan el posible "**snippet**" una palabra a la vez añadiendo al "**score**" el "**TFIDF**" de la palabra si esta pertenece a la "**query**" y restando el de la que estaba al inicio del texto del "**snippet**". Si este nuevo "**snippet**", es mas favorable que el anterior guardamos la posición de la ultima palabra que lo conforma y al final reconstruimos el texto.

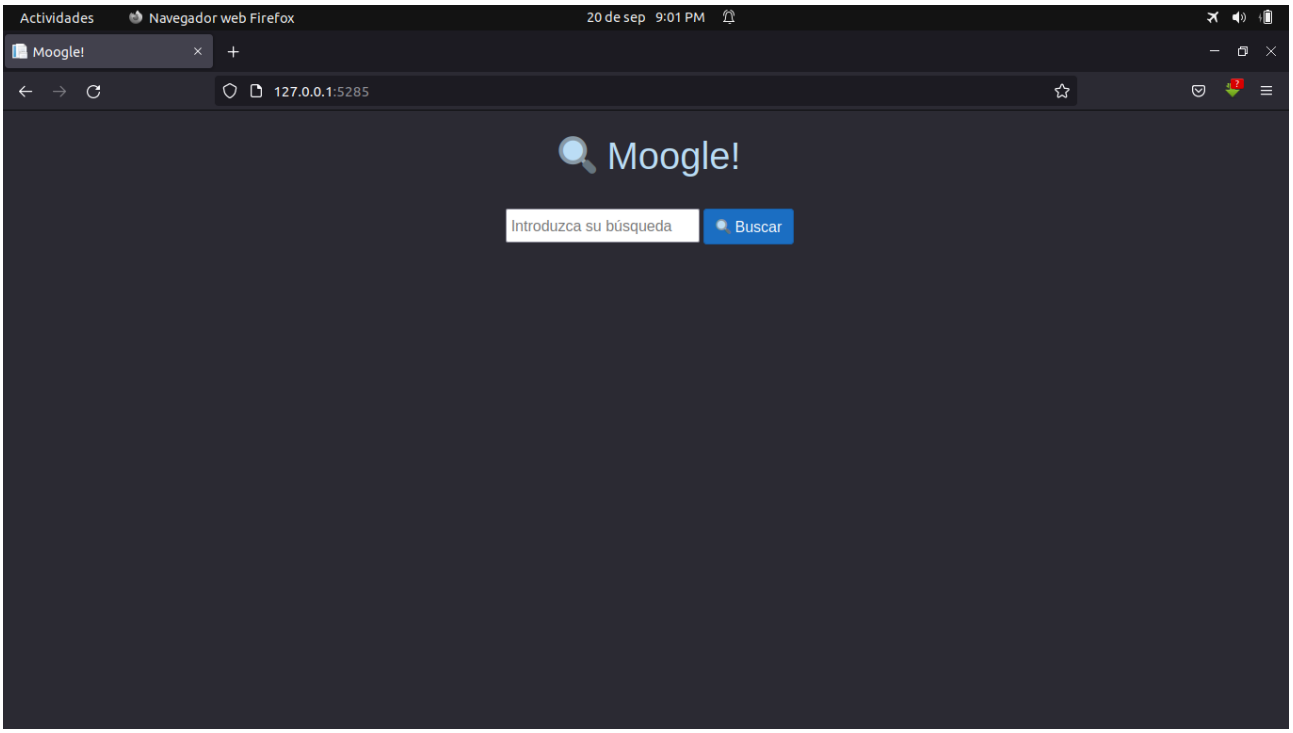
6. "**Tools.FindOperators()**": en cada palabra de la "**query**": se busca si contiene algún " "**operador**", en caso de tener '**!**' se desechan los demás, en caso de tener '*****', se adiciona al conjunto de operadores(esto se hace cada vez que este aparece; pues es importante tener la cantidad de ellos que modifican a la palabra), en caso de '^' se pone de primero y por ultimo para el caso '~' si la palabra que esta a la izquierda no la modifica '!' entonces se añade '~' a su conjunto de operadores, y a la que esta a la derecha igual se le añade.

7. "**Tools.LoadDocuments()**": se obtiene el directorio de la carpeta "**../Content**", luego para cada documento de la carpeta si no es vacío entonces se añade al universo de documentos.

8. "**Tools.LoadAndCreateSynonymsDictionary()**": se obtiene el diccionario de sinónimos guardado en un archivo .***json***, para ser procesado. Para cada palabra de cada conjunto de sinónimos si esta aun no ha sido relacionada se añade al diccionario y se le hacen corresponder todos sus posibles sinónimos.

Manual de Usuario:

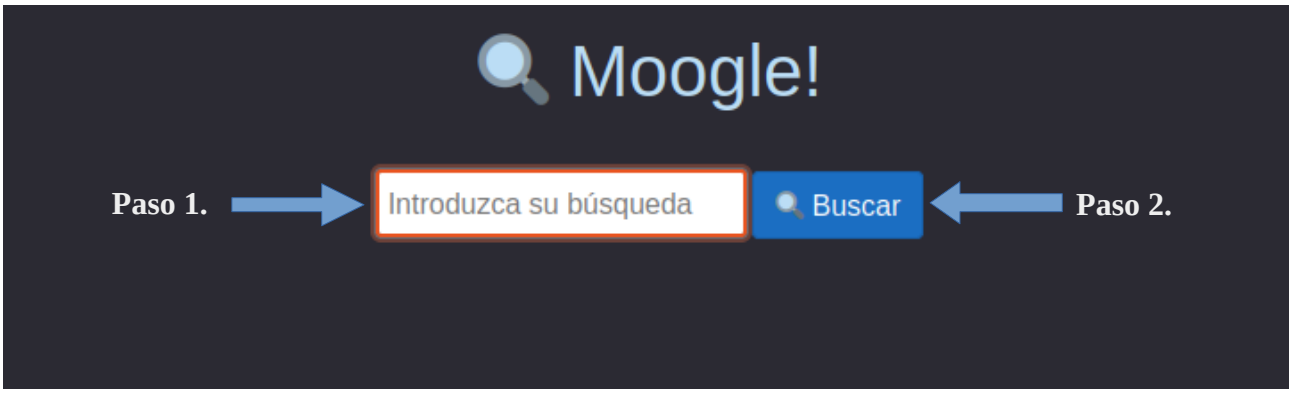
Cuando se accede en la pagina web de Moogle! Se nos muestra de la siguiente forma:



¿Como realizar una búsqueda(query) con Moogle!?

Para realizar una búsqueda solo tenemos que seguir dos sencillos pasos:

1. Escribir lo que deseamos buscar en la caja de texto que dice "**Introduzca su búsqueda**".
2. Presionar el botón que dice " **Buscar**".



3. Finalmente los resultados de la búsqueda se mostraran de la siguiente manera:
   3.1. Primeramente aparecerá una sugerencia de búsqueda resaltada de la misma manera que son resaltados los enlaces. Esta aparece de bajo de la caja de texto y el botón buscar.
   3.2. Luego aparecerán en recuadros los resultados de la búsqueda osea los documentos que obtuvieron mejor score ordenados de manera descendente.

Operadores. Uso y propiedades.

El buscador posee cuatro operadores para hacer mas eficiente la búsqueda, estos son: ‘*****’, ’**!**’, ’^’ y ‘**~**’ . Para hacer uso de un operador solo es necesario escribirlo en una palabra(a excepción de ‘**~**’ este tiene que estar entre dos palabras y separado de estas por un espacio). Para mas claridad a la hora de redactar el criterio de búsqueda se recomienda al usuario colocar el/los operador/es al principio de la palabra aunque el buscador también los procesara si estos aparecen en el medio o al final de la misma.

Ejemplos: “**Harry !Potter**”, “**Harry Po!tter**” y “**Harry Potter!**” son equivalentes.

Funcionalidad de los operadores.

!: Este operador indica que la palabra no puede aparecer osea que no muestra documentos que contengan esa palabra sin importar que contenga otras palabras importantes para la búsqueda.

^: Este operador indica que la palabra tiene que aparecer en el documento para que esta sea mostrado como resultado de la búsqueda.

*****: Este operador indica que la palabra es mas importante que las demás en la búsqueda y puede ser usado de manera repetida osea que mientras mas veces aparezca en una palabra mas importante sera esta y este aumento de importancia sera de manera exponencial osea $2^{(\text{cantidad de ‘*’})}$.

~: Este operador indica que los documentos en los cuales estas palabras estén mas cercanas serán mas importantes para la búsqueda.

Propiedades.

Al poder usarse mas de un operador por palabra a la misma vez para poder tener un criterio de búsqueda aun mas preciso estos cumplen con con ciertas propiedades de relación en este sentido:

1. Si en una palabra aparece “**!**” todos los demás operadores son descartados, incluso “^”.
2. “**~**” solo sera procesado si aparece entre dos palabras separado por un espacio entre estas.