Informe:

Para la realización del proyecto y poder cumplir con la condición de modularidad, se ha dividido minuciosamente en varias clases; éstas son: Data, DataTFIDF, Suggestion, Operator además de las ya existentes; Moogle, SearchResult y SearchItem.

Éstas clases tienen cada una distintas funciones. Pero antes de entrar en detalles quisiera dar una breve explicación en abstracto de como funciona el proyecto para así a la hora de entrar en detalles se lucide mejor el porqué de las cosas.

La clase Data se encargaría de recopilar toda la información necesaria para el funcionamiento del programa en su conjunto, toda esta información sería reunida en preprocesado, luego la clase DataTFIDF se encargaría de darnos los valores de TF-IDF(Método para puntuar las palabras; se explicará mas adelante) de cada palabra en su respectivo documento, luego Suggestion nos podría encontrar una sugerencia en caso de que alguna palabra a buscar no aparezca. Por último Operator se encargaría de verificar si se encuentra algún operador en la query y efectuar algunas otras tareas. De esta manera, de la colaboración armoniosa entre las distintas partes del proyecto, poder dar respuesta a una determinada query; devolviendo nombre del documento, un snnipet, además de los documentos ordenados por relevancia.

Bien, pues empecemos por el principio; el preprocesado, y lo primero sería la clase Data:

1-En dicha clase tenemos varios campos de clase, los cuales son:

1.1-string[] fileAdresses: un arreglo de tipo string que almacena las direcciones de los documentos.

1.2-string[] fileName: un arreglo de tipo string que almacena los nombres de los documentos.

1.3-string [] fileContent: un arreglo de tipo string que almacena el contenido de los documentos.

1.4-string[] normContent: un arreglo de tipo string que almacena el contenido de los documentos normalizado(solo letras y números). Se rellenará dentro del mismo constructor con un bucle for y utilizando el metodo NormalizeText(), no es necesario crear un método que haga esto; pues es extremadamente sencillo.

1.5-Dictionary<string, int[]> [] wordsPosition: un arreglo de tipo Dictionary donde cada posición corresponde a un documento, y en cada diccionario se toman como Key las

palabras del documento y se les asocia como Value un arreglo de int que tiene las posiciones donde aparece la palabra en dicho documento.

1.6-Dictionary<string, string>[] snnipets: un arreglo de tipo Dictionary donde cada posición corresponde a un documento y en cada diccionario se toman como Key un string con una línea del documento y se le asocia como Value un string con dicha línea normalizada(solo letras y números).

2-Luego para inicializar cada uno de los campos anteriores contamos con varios métodos:

2.1-public static string NormalizeText(string text){...}: éste método devuelve texto normalizado(solo letras y números), recibe como argumento un string llamado texto; lo que hace es utilizar la clase RegularExpressions, primero lleva todo a minúsculas, luego quita las tíldes, y por último solo se queda con letras y números. Será muy útil para otros métodos y para buscar.

2.2-private static string[] GetAdresses(){...}: éste método obtiene las direcciones de cada documento, para ello nos servimos de que sabemos que los documentos se encuentran en la carpeta "…/moogle/content/", y de que nos encontramos en "…/moogle/MoogleEngine/", luego utilizaremos el método Directory.GetFiles(@"../content/", "*.txt") para almacenar en un arreglo de string las direcciones de cada documento, y esto será lo que devolveremos para inicializar el campo de clase string[] fileAdresses.

2.3-private static string[] GetFileContent(string[] fileAdresses){...}: éste método devuelve un arreglo de string que tendrá la misma longitud que fileAdresses pues se corresponde con la cantidad de documentos, luego se declara un objeto de tipo StreamReader, y se procede a ir con un for por cada dirección en fileAdresses e ir leyendo su contenido y almacenándolo en su posición homóloga en el arreglo de string a devolver. Con lo devuelto inicializaremos el campo de clase string[] fileContent.

2.4-private static string[] GetFileName(string[] fileAdresses){...}: éste método devuelve un arreglo de string que tendrá la misma longitud que fileAdresses pues se corresponde con la cantidad de documentos, luego se procede a ir con un for por cada dirección en fileAdresses y se realiza un substring a partir del 11no caracter de forma tal que queden los nombres, pues recordemos que todas las direcciones son de la forma "../content/name.txt", de forma tal que a partir del 11no caracter tendríamos el nombre del documento. Con lo devuelto inicializaremos el campo de clase string[] fileName.

2.5-private static Dictionary<string, string>[] GetFileSnnipets(string[] fileContent){...}: éste método devuelve un arreglo de diccionario de Key tipo string y Value tipo string. Cada posición del arreglo representa un documento y en cada diccionario las Key serán las líneas del documento y los Value serán esas mismas líneas pero normalizadas. Con lo devuelto inicializaremos el campo de clase Dictionary<string, string>[] snnipets.

2.6-private static Dictionary<string, int[]>[] GetWordsPosition(string[] fileContent){...}: éste método devuelve un arreglo de diccionario, donde cada posición del arreglo representa un documento, cada diccionario tiene una Key de tipo string y un Value de tipo int[], donde las Key serían las palabras del documento y los Value serían las posiciones dónde aparecen en el documento. Con lo devuelto inicializaremos el campo de clase Dictionary<string, int[]>[] wordsPosition.

2.7-Luego tenemos ciertos métodos que son public, los cuáles no son static, pues se utilizarán para acceder a toda la información que contienen los objetos de tipo Data.

A continuacion tendríamos la clase DataTFIDF.

Antes de proseguir a explicar la estructura de la clase, primero debemos definir cuál es la función de esta clase; la idea es que para buscar en los textos necesito darles una puntuación para saber cuáles voy a devolver o no, para ello utilizaremos un método de puntuar, el cuál es el conocido TF-IDF; se divide en dos partes; la primera es el TF, que es la frecuencia del término en su respectivo documento, la cual la hallamos dividiendo la cantidad de veces que aparece dicho término entre la cantidad total de palabras en dicho documento, luego la segunda parte sería el IDF, el cual es la frecuencia de aparición del término entre todos los documentos; o sea que mientras más veces aparezca menos importante debe ser, para ello calculamos el logaritmo de la cantidad total de documentos entre la cantidad de documentos en que aparece el término: mientras en más documenos aparezca menor será su valor. Luego solo resta multiplicar el valor de TF por el de IDF de cada palabra y asociárselo a cada palabra en su respectivo documento, y tendríamos lista de antemano una base de datos acerca de los scores de los términos.

1-Lo primero es que esta clase solo tiene un campo de clase:

1.1-Dictionary<string, float>[] vectorTFIDF: un arreglo de diccionario donde cada posición corresponde a un documento y cada diccionario tiene una Key de tipo string y un Value de tipo float, las Key serían las palabras correspondientes a ese

documento y el Value vendría siendo su valor de TFIDF, el cual será la base del cálculo del score de los documentos en función de la búsqueda.

2-Luego los métodos que trabajan en armonía de forma tal de poder calcular el TF-IDF son:

2.1-private static Dictionary<string, int> GetPreIDF(string[] fileContent): éste método recolecta información acerca de en cuántos documentos aparece cada palabra del universo, luego en el diccionario que devuelve asocia cada palabra con la cantidad de documentos en que aparece. Esta información es muy importante para el cálculo del IDF.

2.2-private static Dictionary<string, float> GetIDF(Dictionary<string, int> preIDF, int fileNumber){...}: éste método devuelve un diccionario con Key de tipo string y Value de tipo float, donde las Key son las palabras y los Value son su valor de IDF. Para ello nos apoyaremos en el preIDF y en fileNumber(cantidad total de documentos), y preIDF es resultado del método GetPreIDF el cuál tiene en cuántos documentos aparece cada palabra.

2.3-private static Dictionary<string, int>[] GetPreTF(string[] fileContent){...}: éste método recolecta información acerca de cuantas veces aparece una palabra en su respectivo documento(necesario para el cálculo del TF),

2.4-private static Dictionary<string, float>[] GetTF(Dictionary<string, int>[] preTF, string[] fileContent){...}: éste método apoyandose en el metodo GetPreTF, calculará el TF de cada palabra en su respectivo documento. Devuelve un arreglo de diccionario donde cada posición representa un documento, y cada diccionario tiene por Keys las palabras de dicho documento que representa y de Value su valor de TF, el cual se calcula dividiendo la cantidad de palabras total del documento entre la cantidad de veces que aparece la palabra que esta por Key en el documento.

2.5-private static Dictionary<string, float>[] GetTFIDF(Dictionary<string, float> IDF, Dictionary<string, float>[] TF){...}: éste método devuelve un arreglo de diccionario, donde cada posición representa un documento y cada diccionario tiene por Keys las palabras del documento y por Value su valor de TF-IDF; el cual se obtiene mediante la mutliplicación de su TF en ese documento por su IDF.

2.6-Finalmente tenemos un método public y no static para obtener los valores de TFIDF desde fuera de la clase a través de un objeto de tipo DataTFIDF.

A continuación tendríamos la clase Suggestion:

1-La cual sólo presenta un campo de clase:

1.1-Dictionary<string, string> words: un diccionario que tiene por Key un string y por Value un string, dicha Key vendría siendo las palabras del universo de palabra sin normalizar y por Value dicha palabra pero normalizada. Esto será de gran valor para estructurar las sugerencias.

2-Luego proseguimos con los métodos:

2.1-private static Dictionary<string, string> GetWords(string[] fileContent){...}: éste método devuelve un diccionario que tiene por Keys las palabras del universo de documentos y le asocia como Value la misma palabra normalizada, para ello se apoya en el metodo NormalizeTex() de la clase Data. Es un método realmente sencillo.

2.2-private static int Levenshtein(string st1, string st2){...}: éste método devuelve un valor de tipo int, el cual representa la cantidad de modificaciones necesarias que debemos hacerle a una cadena para llegar a otra, las cuales pueden ser reemplazo, inserción o eliminación. Su funcionamiento es utilizando una matriz, donde las columnas representarían los caracteres de una de las palabras y las filas los caracteres de la otra.

Luego se rellena la fila y la columna de la matriz, sin contar la posicion (0;0), con numeros desde el uno hasta la longitud máxima de cada cadena. Y se procede a verificar si los carácteres son iguales, en caso de que si, se pone un cero en la diagonal, en caso de que no, se calcula en función de la operación cual es el costo y se coloca en la diagonal, este se va acumulando hasta terminar en la esquina inferior derecha, siendo este valor el costo en operaciones para convertir una cadena en otra.

2.3-public string Suggestion(string wordQuery){...}: éste método recibe una palabra e intenta encontrarle una sugerencia, primero verifica que la palabra no se encuentre en el universo y luego invoca el método Levenshtein para verificar si hay alguna posible sugerencia.

Proseguimos con la clase Operator:

1-Los campos de clase son:

1.1-char[] oper: un arreglo de tipo char, tendrá la misma dimensión que la cantidad de palabras de la query, tendrá en cada posición un char que representará al operador de la palabra de la query correspondiente en caso de que tenga.

1.2-private int[] countAsterisc: un arreglo de tipo int, tendrá la misma dimensión que cantidad de palabras de la query, y tendrá la cantidad de veces que aparece el

operador “*” en la posición correspondiente con la palabra de la query que presente dicho operador.

2-Luego los métodos de la clase son:

2.1-private static char[] CheckOperator(string[] splitQuery){...}: éste método devuelve un arreglo de tipo char, el cual tendrá la misma dimensión que la cantidad de palabras de la query, tendrá en cada posición un char que representará al operador de la palabra de la query correspondiente. Para verificar si aparece algún operador, lo que hacemos es ir verificando el primer caracter de cada palabra de la query y se lo vamos asociando, una vez encontrado o no alguno en la primera posicion se rompe el ciclo y se va a la siguiente palabra. El único operador que tiene una condición especial sería el de cercanía, pues no puede aparecer en la primera posicion de splitQuery, pues no habría una palabra anterior para calcular su distancia con la siguiente.

2.2-private static int[] GetAsterisc(string[] splitQuery){...}: éste método devuelve un arreglo de tipo int, el cual tendrá la misma dimensión que la cantidad de palabras de la query. Verificará si alguna palabra de la query tiene el operador “*” y procederá a contar cuántos tenga, luego lo almacenara en la posicion correspondiente del arreglo de tipo int.

2.3-public float GetDistance(string w1, string w2, Dictionary<string, int[]> wordsPosition){...}: éste método devuelve un valor de tipo float que se corresponde con el valor a multiplicar por el score de un documento si presenta las 2 palabras relacionadas por el operador de cercanía. Este es un algoritmo no trivial, por lo que procederé a explicarlo en detalle;

Como sabemos previamente obtuvimos un arreglo de diccionario en la clase Data llamado wordsPositions, éste contiene en cada diccionario las palabras de ese documento y le asocia un arreglo de tipo int, que contiene las posiciones de las palabras, pues a la hora de llamar a este método solo le pasaremos el del documento en el que estemos trabajando, el cual será el diccionario wordsPositions que se pasa en el argumento.

Ahora procederemos a verificar que las 2 palabras aparezcan en el documento, sino lo hacen devolvemos un 1, pues no habrá que modificar el valor del score del documento debido a este operador. En caso de que ambas aparezcan lo que haremos será recorrer con un bucle for los valores del primer arreglo de tipo int con las posiciones de w1, y iremos restando este valor con todos los valores del arreglo de tipo int de las posiciones de w2, e iremos almacenando la que tenga menor módulo,

de esta forma tendremos lo más cercanas que están las palabras en dicho documento.

Procedemos a dividir a una constante “c” con un valor de 10, y la dividimos por la distancia entre las dos palabras y este valor de tipo float será el que devolveremos para multiplicar por el score del documento.

2.4-Luego los métodos que restan son para obtener información de los campos de clase.

Finalmente verificaremos la clase Moogle:

1-Entre sus campos de clase se encuentran:

1.1-public static Data data: un objeto de tipo Data.

1.2-public static DataTFIDF vTFIDF: un objeto de tipo DataTFIDF.

1.3-public static Suggestion sgt: un objeto de tipo Suggestion.

1.4-static bool [] matchQuery: una máscara booleana, será utilizada para verificar a la hora de buscar sugerencias si la palabra se encuentra en el universo de palabras, en caso de que sea false no se busca sugerencia.

2-A continuación vienen los métodos de la clase:

2.1-private static float [] GetScore(int fileNumber, string [] splitQuery, Dictionary<string, float>[] TFIDF, Operator oper){...}: éste método devuelve un arreglo de tipo float, donde cada posición es un documento y el valor que contiene será su score. Para el cálculo de dicho score este método recibe varios argumentos; fileNumber(cantidad de documentos), splitQuery(la query), TFIDF(los valores de tfidf de cada palabra en su documento) y oper(un objeto de tipo Operator para verificar si hay que modificar los scores).

Como el algoritmo para el cálculo del score no es trivial, entraré en detalles:

Primero inicializamos un arreglo de diccionario(wordsPositions) que obtendremos a través del método GetWordsPositions() de la clase Data.

Luego inicializamos la máscara booleana la cual tendrá un tamaño igual a la cantidad de palabras de la query.

Cargamos los operadores para su verificacion en un arreglo de tipo char llamado operQuery, a traves del método GetOperator() de la clase Operator.

Procedemos a inicializar float[] docScore el cual será el arreglo a devolver, le daremos un tamaño acorde a fileNumber pues es la cantidad total de documentos y la máxima cantidad de valores que podríamos calcular.

A continuación inicializamos float distanceValue = 0; por si se da el caso que hay operador de distancia; “~”.

Luego hacemos un bucle for para ir por cada documento y luego otro bucle for para ir revisando cada palabra de la query, luego vendrían una serie de condicionales, donde verificamos si la palabra aparece en ese documento(verificamos si aparecen en TFIDF en la posicion I, pues aqui están las palabras sin repetirse del documento I) y que el string no es vacío. En caso de que no aparezca, verificamos si no es porque era un operador de cercanía, en cuyo caso procedemos a darle a distanceValue el valor correspondiente en función de la distancia entre las palabras j+1 y j-1, este valor se guardará para cuando termine el cálculo del score del documento multiplicarlo por el mismo. Luego si no aparece la palabra y no es un operador de cercanía, verificamos en operQuery[j] si es un operador de “debe aparecer”(^), luego si la condición es true, multiplicamos el score del documento por cero, pues este documento no debe ser devuelto. Volvamos al caso en que la palabra aparezca en el documento, entonces procedemos a verificar si hay operador de importancia (*) y en caso positivo procedemos a obtener la cantidad de asteriscos a traves del método GetAsterisc() de la clase Operator y almacenarlo en una variable de tipo int llamada count, luego multiplicamos el valor de TFIDF de esa palabra por count y lo acumulamos en docScore[i]. En caso de que no encontremos dicho operador, verificamos si es operador de “no debe aparecer”(!), en cuyo caso le damos cero al valor del score y hacemos un break para ir al siguiente documento. En el caso de que no aparezcan ninguno de estos dos operadores, procedemos a simplemente acumular el valor de TFIDF de la palabra en el score del documento.

Cada vez que se termine de calcular el score de un documento se verifica si distanceValue es mayor que cero, en cuyo caso se multiplica el valor del score del documento por dicho valor.

2.2-private static KeyValuePair<float, int>[] GetScorePosition(float[] docScore){...}: éste método devuelve un arreglo de tipo KeyValuePair<float, int>, donde cada posicion del arreglo será un valor de score, donde la Key sera el valor de score y el Value será su posicion en scorePosition, la cuál se correspondera con la posición del documento en el resto de estructuras, despues de eso procedemos a utilizar ScorePosition = ScorePosition.OrderByDescending(x=>x.Key).ToArray(); para ordenar los scores de mayor a menor por la Key y luego devolver ScorePosition.

2.3-private static string [] GetFinalSnnipets(Data data, DataTFIDF vTFIDF, string [] normSplitQuery, KeyValuePair<float, int>[] scorePosition, Operator oper){...}: éste método devuelve un arreglo de tipo string, donde cada posición es la mejor línea en respuesta a una determinada query. Para su funcionamiento, el método aprovecha los scores del TFIDF, la idea será tomar cada documento como el universo, cada línea como documento, y los términos de la línea como las palabras del documento, y utilizar el mismo algoritmo de cálculo de score utilizado para los documentos, de esta forma, garantizaremos devolver una línea con sentido en función de la query.

2.4-private static SearchResult FinalResult(string query, int counter, string [] fileName, KeyValuePair<float, int> [] ScorePosition, string [] unSplitQuery, string[] splitQuery, Operator oper){...}: éste método devuelve un objeto de tipo SearchResult, que sería la respuesta a la consulta de la query, para ello recibe muchos argumentos, como es un algoritmo no trivial lo explicaré en detalle:

Primero que nada declaramos el arreglo de tipo SearchResult a devolver; items.

Luego procedemos a una serie de condicionales; primero verificamos si counter es mayor que cero, esto significa que hay respuesta a la búsqueda; pues counter es la cantidad de documentos con score mayor que cero. En caso positivo utilizamos el método GetFinalSnnipets() para rellenar la variable string[] snnipets con los snnipets que vamos a devolver.

Entonces utilizamos un bucle for para rellenar items con los nombres, snnipets y scores, para ello nos servimos de scorePosition, el cual tiene como Key los scores y como Value la posición del documento, debido a que el tamaño de items es igual a la cantidad de documentos con score mayor que cero, se tomaran todos los documentos que cumplan esta condición ya que están ordenados de mayor a menor en scorePosition, cuando llegue el documento cuyo score es cero, ya se habrá detenido el rellenado de items.

Ahora declaramos un bool llamado existSuggestion, el cual será true en caso de que tengamos alguna sugerencia, y declaramos la variable string suggestion y la

inicializamos como “”. Acto seguido procedemos a recorrer la máscara booleana matchQuery en búsqueda de un false(palabra que no aparece en el universo). En caso de que no aparezca dicha palabra, buscamos una sugerencia y de existir la incorporamos y le damos true a existSuggestion. Y una vez terminada la búsqueda, si existSuggestion es true, devolvemos items y suggestion, en caso contrario solo devolvemos items.

Luego suponiendo que counter sea cero, pero la query no este vacía, procedemos a intentar encontrar una sugerencia para el usuario, en caso de que se encuentre se devuelve solamente suggestion. Por último si la query esta vacía y counter es cero, no devolvemos nada.

Notas: para determinar si devolver items y suggestion, a la vez o uno de los dos solamente, cree mas constructores en la clase SearchResult.

Las clases cuya información es necesaria que esté de ante mano, se inicializan en “…/MoogleServer/Program.cs” antes de que levante la página.

También realicé modificaciones en ../MoogleServer/Pages/Index.razor, modificando el comportamiento de la acción de dar click en la sugerencia y si se muestra el mensaje de “Quisiste decir ….?” en función de si hay o no sugerencia.