

Primeramente en mi programa dentro del **namespace** *MooglesEngine* y utilizando la biblioteca **System.IO**; dentro de la clase estática *Moogles* busco la ruta dinámica desde donde se ejecuta el programa y en la carpeta *Content* almacenó los nombres de los archivos en un string [].

Luego creo un objeto estático de mi clase *DiccionarioReferencial*.

Esta clase *DiccionarioReferencial* crea un diccionario que contiene información sobre la ocurrencia de las palabras en documentos y su posición en una matriz de índices. El constructor toma una lista de archivos de texto como entrada y realiza varias operaciones en cada archivo para crear el diccionario. Primero, tokeniza el contenido del archivo en palabras separadas por delimitadores. Luego, cuenta la cantidad de palabras en cada documento y las agrega al diccionario *Cant_pal_DOC*. También asigna un índice de fila a cada documento en el diccionario *Indexador_Fila*. A continuación, para cada palabra en el archivo, agrega una entrada al diccionario *Indexador_Columnas* si no existe ya, y agrega la palabra al diccionario *Ocurrencia_de_i_en_documentos* si tampoco existe. Si la palabra ya existe en el diccionario, incrementa su valor de ocurrencia en 1 (una vez por cada documento donde aparezca). Finalmente, establece la cantidad de archivos procesados en la variable *Cant_arch*. También hay un método *Tokenizar_txt* que toma una ruta de archivo o una cadena de texto como entrada y devuelve un string [] de palabras tokenizadas.

Luego creo un objeto estático *Matriz_TF_IDF* de esta clase:

Esta clase matriz tiene dos métodos: el constructor y *Respuesta_query*. También tiene una propiedad matriz formada por un *Vector* []

Es importante explicar antes en que consiste la clase vector:

Vector representa un vector en la *Matriz_TF_IDF* como un **double** []. Tiene varios métodos:

- El constructor toma como entrada un objeto *DiccionarioReferencial* y una cadena que representa el archivo correspondiente al vector. Crea un nuevo vector y lo inicializa con ceros. Luego, itera sobre las palabras en el archivo y actualiza el vector con los valores TF-IDF correspondientes para cada palabra.
- El segundo constructor toma como entrada un **double** [] que representa un query y crea un nuevo vector con esos valores.
- El método *Calculo_TF_IDF* calcula el valor TF-IDF para una palabra en un documento específico, utilizando la fórmula correspondiente.
- El método *Multiplicar_por_Vector* calcula el producto punto entre dos vectores.
- El método *Norma* calcula la norma del vector.
- El método *Cosigno* calcula el coseno entre dos vectores.

En resumen, esta clase se encarga de representar y manipular vectores en la matriz TF-IDF.

En cuanto al cálculo de TF * IDF:

La fórmula que utilizo es:

$TF_{(i)} = \frac{\log_2(Freq(i,j)+1)}{\log_2(L_j)}$ <p>$Freq(i,j)$ = Frecuencia del término i en el documento j.</p> <p>$L_{(j)}$ = Número total de términos en el documento j.</p>	$IDF_i = \log_2(1 + \frac{N_D}{f_i})$ <p>N_D = Número total de documentos considerados.</p> <p>f_i = Número de documentos que contienen el término i.</p>
---	--

Para evitar indefiniciones con las funciones logarítmicas en el código realicé ajustes a los argumentos, o sea:

En vez de $L_{(j)}$ utilizo $L_{(j)} + 000.1$ y en vez de $f_{(i)}$ utilizo $f_{(i)} + 1$

Luego volviendo a la clase *Matriz_TF_IDF*

El constructor toma como entrada un objeto *Diccionario_Referencial* y una cadena que representa la carpeta donde se encuentran los documentos. Crea una matriz de objetos *Vector*, donde cada *Vector* representa una fila en la matriz. Luego, itera sobre el diccionario de índices de fila en el objeto *Diccionario_Referencial* y crea un nuevo objeto *Vector* para cada fila en la matriz. El objeto *Vector* toma como entrada el objeto *Diccionario_Referencial* y el índice de fila correspondiente. Finalmente, agrega el objeto *Vector* a la matriz.

El método *Respuesta_query* toma como entrada un objeto *Query_class* de la clase con el mismo nombre, un objeto *Matriz_TF_IDF* y un objeto *Diccionario_Referencial*. Itera sobre el diccionario de índices de fila en el objeto *Diccionario_Referencial* y calcula el coseno entre el vector de consulta (almacenado en el objeto *Query_class*) y cada fila en la matriz (almacenada en el objeto *Matriz_TF_IDF*). Agrega los resultados al diccionario *Solution*, que contiene los nombres de los documentos y sus puntajes de similitud con la consulta. Si *Solution* tiene más de 5 elementos, elimina el documento con el puntaje más bajo (utilizando el método *Menor*). Luego, elimina cualquier entrada en *Solution* que tenga un puntaje de similitud igual a cero. Finalmente, ordena *Solution* por puntaje de similitud y lo devuelve como un nuevo diccionario llamado *orderedSolution*.

El método estático *Menor* toma como entrada un diccionario de **string** y **double** y devuelve la clave del par con el valor más bajo.

Luego está el método estático *Query* el cual dado un query devuelve un objeto *SearchResult*.

Para esto primeramente se crea un *SearchItems [] items* así como un objeto que representa al query de mi clase *Query_class*.

Query_class que se utiliza para representar el query y trabajar con él. El constructor de la clase recibe como parámetros el diccionario referencial y el string query.

El método *Snippet* se encarga de generar un fragmento de texto que contenga las palabras más relevantes de la consulta en el contexto del documento. Para ello, primero se tokeniza el documento y se identifican las palabras que coinciden con las de la consulta. Luego se ordenan estas palabras según su relevancia (calculada mediante el método TF-IDF) y se seleccionan las dos más relevantes.

A continuación, se busca la posición más cercana de estas dos palabras en el documento (método *posicion_mas_cercana*) y se determina la vecindad de cada una (es decir, las palabras que la rodean, método *Vecindad*). Si la distancia entre ambas vecindades es mayor a 7 palabras, se retorna un fragmento que incluye ambas vecindades separadas por puntos suspensivos. Si la distancia es menor o igual a 7, se retorna un fragmento que incluye ambas vecindades sin puntos suspensivos.

En resumen, el método *Snippet* genera un fragmento de texto que contiene las palabras más relevantes de la consulta en el contexto del documento, lo que puede ayudar al usuario a entender mejor cómo se relaciona su consulta con el contenido.

Además están los métodos:

Busqueda_valida: Ve si el query es válido, sino devuelve algo así



Para la sugerencia usé los siguientes métodos:

LevenshteinsDistance: La distancia de Levenshtein o distancia entre palabras es el número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra. Se entiende por operación, bien una inserción, eliminación o la sustitución de un carácter. Este método calcula dicha distancia.

MostSimilarWord: Devuelve la palabra más parecida (menor distancia de Levensteins) a la que se ingresa.

Suggestion: Devuelve un query, sustituyendo las palabras invalidas por sus más parecidas.

Luego a través del método *Busqueda_valida* se verifica si el query es válido y si lo es se crea un diccionario con los documentos más relevantes y su respectivo score (este diccionario le da la dimensión a *items*) Luego se inicializa cada elemento de *SearchItems [] items* incluyendo su *Snippet*

Al final se hace una sugerencia (método *Suggestion*) del query y se devuelve un objeto *SearchResult* que se imprime en el **Moogole**.