Informe sobre el proyecto Moogle

Abraham Romero Imbert

Julio, 2023

Resumen

En este informe se estará explicando de manera más detallada el funcionamiento del Proyecto Moogle!. Se interiorizará la ejecución del mismo así como en las clases y métodos que intervienen en el proceso para dar una respuesta al usuario, una vez hecha una consulta.

1. Introducción

Es el primer proyecto de Programación orientado a Primer Año de Ciencias de la Computación. Básicamente es una aplicación cuyo propósito es buscar inteligentemente un texto en un conjunto de documentos. Como cualquier buscador busca información en una base de datos a partir de palabras clave o frases que el usuario introduce en la barra de búsqueda. Utiliza algoritmos para encontrar archivos que contienen las palabras clave y las presenta al usuario en una lista de resultados teniendo en cuenta la relevancia de los mismos respecto a la búsqueda realizada. Adicionalmente este buscador tiene la capacidad de sugerir al usuario otras posibles búsquedas, en especial al no encontrar resultados en la base de datos semejantes a la petición inicial del usuario.

2. Funcionamiento

2.1. Ruta de ejecución

Dentro del `namespace` **MoogleEngine** y utilizando la biblioteca **System.IO**; dentro de la clase estática **Moogle** busco la ruta dinámica(desde donde se encuentra el proyecto) y se ejecuta el programa y analizando cada archivo de la carpeta **Content** (ubicada en la carpeta raíz) almaceno los nombres de los archivos en un `string [ ]`.

2.2. Clase: Diccionario-Referencial

Luego creo un objeto estático de mi clase **Diccionario-Referencial**. Esta clase 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 `string [ ]` 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.

2.3. Clase: Matriz-TF-IDF

Luego creo un objeto estático de la `clase` **Matriz-TF-IDF**. Esta 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**:

2.3.1. Clase: Vector

Vector representa un vector en **Matriz-TF-IDF** como un `double [ ]`. Tiene varios métodos:

1. El constructo: toma como entrada un objeto **Diccionario-Referencial** y un `string` 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.

2. El segundo constructor: toma como entrada un `double [ ]` que representa un *query* y crea un nuevo vector con esos valores.

3. El método **Calculo-TF-IDF** calcula el valor *TF-IDF* para una palabra en un documento específico, utilizando la fórmula siguiente:

$$TF \cdot IDF \tag{1}$$

Donde

$$TF_i = \frac{\log_2(Freq(i,j)+1)}{\log_2(L_j + 0{,}001)} \tag{2}$$

Freq(i,j)+1 = Frecuencia del término i en el documento j.

L_j = Número total de términos en el documento j. En este caso se le sumo a este valor 0.001 para evitar indefiniciones en la función

$$IDF_i = log_2(1 + \frac{N_d}{f_i + 1}) \tag{3}$$

N_d= Número total de documentos considerados
f_i = Número de documentos que contienen el término i. En este caso se le sumo a este valor 1 para evitar indefiniciones en la función

4. El *método* **Multiplicar-por-Vector** calcula el *producto punto* entre dos vectores.

5. **Norma** calcula la *norma* del vector.

6. **Cosigno** calcula el *coseno* entre dos vectores.

   Este viene determinado por la fórmula siguiente y representa la semejanza que existen entre dos documentos (incluido la *query*) este valor viene oscila entre 0 y 1 donde 0 representa que ambos documentos son *ortogonales* o sea no tienen palabras en común y 1 que son documentos con las mismas ocurrencias de palabras una vez normalizados, claramente mientras más cercano sea el valor de *coseno* a 1 más similares serán los documentos.

$$\cos(\theta) = \frac{A \cdot B}{\parallel A \parallel \parallel B \parallel} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{4}$$

2.4. Continuación Clase: Matriz-TF-IDF

Luego volviendo a la clase **Matriz-TF-IDF**: El *constructor* toma como entrada un objeto **Diccionario-Referencial** y la *ruta* donde se encuentran los documentos. Crea un **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. Este toma como entrada el *objeto* **Diccionario-Referencial** y el índice de fila correspondiente. Finalmente, agrega el 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 textbfSearchResult `[ ]` items así como un objeto que representa al *query* de mi clase **Query-class**.

Query-class 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:

Figura 1: Sugerencia en Moogle!

Para la sugerencia usé los siguientes métodos: **LevensteinsDistance**: La *distancia de Levenshtein* o distancia entre palabras es el número mínimo de operaciones requeridas para transformar un 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 inválidas 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 Moogle.

3. Conclusión

En este informe se ha explicado detalladamente el código implementado por mí para la ejecución del Moogle!. Se han detallado las clases y los métodos propios de cada una, así como la manera en que se ejecutan estos, o sea, cómo funcionan. En vista de la posibilidad de un posterior mejoramiento de la capacidad de procesamiento de este proyecto o la implementación de otras funcionalidades se expresa que los métodos anteriores pueden estar

sujetos a cambios. En mi experiencia particular este como mi primer proyecto en la Facultad de Matemática y Computación en la Universidad de la Habana considero que fue un gran paso en el tránsito del desconocimiento en la programación a este nuevo y amplio mundo donde TODO es posible. Quisiera agradecer a mis profesores y profesores de clases prácticas que cumplieron en su labor para aclarar mis dudas respecto al proyecto y me dieron herramientas para poder trabajar de manera más eficiente en su desarrollo, a ellos y demás personas que intervinieron muchas gracias.