Informe del primer proyecto de programación

Michell Viu Ramirez

Julio, 2023

Contents

1 Introducción

Moogle! es una aplicación totalmente original cuyo propósito es buscar inteligentemente un texto en un conjunto de documentos. Es una aplicación web, desarrollada con tecnología .NET Core 6.0, específicamente usando Blazor como framework web para la interfaz gráfica, y en el lenguaje C#. La aplicación está dividida en dos componentes fundamentales:

- *'MoogleServer'* es un servidor web que renderiza la interfaz gráfica y sirve los resultados.
- *'MoogleEngine'* es una biblioteca de clases donde está implementada la lógica del algoritmo de búsqueda.

2 Clases

El proyecto modificado cuenta con cuatro clases diferentes:

2.1 Clase MétodosAdicionales

Es una clase estática que consta de dos métodos:

- **Normaliza** que recibe como parémetro un string query y devuelve un array de tipo string, su función en el proyecto es normalizar el string query así como convertirlo en un array de tipo string y eliminar términos que no son relevantes para la búsqueda como preposiciones, conjunciones y artículos. (**OJO**:solo se han eliminado estas palabras para el idioma español e inglés).
- **SubString** que recibe como parámetros un array de tipo string, un int "inicio", un int "fin"y un string "termino", y retorna un string. Su función es crear el "snippet" del "término" buscado en caso de que aparezca en el documento, ya que su funcionamiento se basa en convertir desde la posicion "inicio" del array hasta la posicion "fin" del array en un string que será la porción donde aparece dicho término en el documento(el "inicio" y el "fin" se introduce en dependencia del tamaño que se quiera tener dicho snippet,más adelante se explica más detallado su calculo).

2.2 Clase Matriz

Consta de una propiedad de tipo double[,], nueve métodos que cada uno de ellos realiza una operación entre matrices y/o vectores. En su constructor recibe un matriz de tipo double (double[,]).

2.2.1 Métodos de la clase Matriz

- **SumaMatriz** recibe como parámetros dos matrices de tipo "Matriz", y devuelve una "Matriz" ,su función es relizar la suma entre dos matrices. En caso de que las matrices no cumplan las condiciones para poder realizar dicha operación se lanza una excepción advirtiendolo.
- **RestaMatriz** recibe como parámetros dos matrices de tipo "Matriz", y devuelve una "Matriz" ,su función es relizar la resta entre dos matrices. En caso de que las matrices no cumplan las condiciones para poder realizar dicha operación se lanza una excepción advirtiendolo.
- **MultiplicaMatriz** recibe como parámetros dos matrices de tipo "Matriz", y devuelve una "Matriz",su función es relizar la multiplicación entre dos matrices. En caso de que las matrices no cumplan las condiciones para poder realizar dicha operación se lanza una excepción advirtiendolo.
- **EscalarMatriz** recibe como parámetros una "Matriz" y un int "escalar" y devuelve una "Matriz", su función es multiplicar dicho escalar por la Matriz.
- **VectorMatriz** recibe como parámetros un array de tipo double y una "Matriz" y devuelve un array de tipo double, su función es multiplicar un vector por una matriz.
- **MultiplicaVectores** recibe como parámetros dos arrays de tipo double y devuelve un array de tipo double, su función es multiplicar dos vectores.
- **SumaVectores** recibe como parámetros dos arrays de tipo double y devuelve un array de tipo double, su función es sumar dos vectores.
- **RestaVectores** recibe como parámetros dos arrays de tipo double y devuelve un array de tipo double, su función es restar dos vectores.
- **MultiplicaEscalarVector** recibe como parámetros un array de tipo double y un int "escalar" y devuelve un array de tipo double, su función es multiplicar un escalar por un vector.

2.3 Clase Documentos

Esta clase consta de cinco propiedades y de diez métodos, la propiedad 'nombreDocumento' de tipo string que represeta el nombre del "Documento",la propiedad 'score' de tipo double que representa la relevancia de ese "Documento" para una búsqueda dada, la propiedad 'contenido' de tipo string[ ] que representa cada palabra del "Documento" y la propiedad 'snippet' de tipo string que representa una porción del "Documento" para una búsqueda dada y la propiedad 'DiccSnippet' de tipo Dictionary(string,string) que representa el snippet predeterminado de cada palabra del documento . En su constructor se recibe como parámetros un string 'nombreDoc' y un string[ ] 'contenido',

las propiedades nombreDocumento y contenido reciben en el constructor los valores de nombreDoc y contenido respectivamente,score se inicializa en 0, snippet se inicialeza en ‘””‘ y DiccSnippet en un nuevo Dictionary(string,string). La propiedad ‘nombreDocumento‘ consta de un método ‘get‘ llamado ‘getNombre‘, la propiedad ‘score‘ consta de un método ‘get‘ y un método ‘set‘ llamados ‘getScore‘ y ‘setScore‘, además de otros dos métodos llamados AddScore y ResetScore; AddScore se encarga de añadir score al documento cuando este se este calculando y ResetScore se encarga de hacer el score de dicho documento 0 nuevamente , la propiedad ‘snippet‘ consta de un método ‘get‘ llamado ‘getSnippet‘, el método ‘AddSnippet‘ recibe un string ‘snippet‘ como parámetro y simplemente lo modifica por tanto devuelve void, para ello utilizamos una condicional if preguntando si ‘this.snippet‘== ”” de ser así significa que es el primer snippet agregado y por tanto ‘this.snippet‘ = ‘snippet‘ de lo contrario significa que ya existe otro ‘snippet‘ de otro término buscado y por tanto se lo agregamos con tres puntos suspensivos para diferenciar cada uno y un método llamado ResetSnippet que hace que el snippet de un documento sea vacio, y la propiedad ‘DiccSnippet‘ consta de un método ‘get‘ llamado ‘getDictionary‘.

Ej:

”””
if (this.snippet == ””)
this.snippet = snippet;
else
this.snippet += ”...” + snippet;
”””

2.4 Clase Datos

Esta clase es de gran importancia porque es la que se encarga del procesamiento de todos los documentos para recopilar los datos necesarios que se utilizarán posteriormente cuando el usuario introduzca la búsqueda.
La clase cuenta con 7 propiedades: archivos que es un array de string y representa cada uno de los documentos .txt, resultadoDoc que es un array de Documentos del mismo tamaño que archivos en el cual se almacenará la información de cada uno de esos archivos, terminosunicos que es una lista de string y representa cada término único de todo el corpus de los documentos, matrizTfIdf que es una matriz donde cada fila se le asocia un término único y cada columna representa un documento, en cada celda de esa matriz estará representado el peso de cada término único según el archivo correspondiente. Además cuenta de la propiedad relativePath de tipo string que representa la ruta relativa donde se encuentran los archivos .txt y la propiedad fullPath que representa de igual manera la ruta de dichos archivos pero usando el metodo Combine de la clase Path para que dichos archivos se puedan cargar en cualquier sistema operativo y por ultimo cuenta con una propiedad privada llamada datos del tipo Datos
En su constructor la clase no recibe parámetros, simplemente se inicializan sus propiedades y se llama al método Procesamiento de dicha clase para que el objeto se inicialice con todos los datos necesarios, además su constructor es pri-

vado.
A este objeto se le creó un método llamado getInstance que devuelve un objeto de tipo Datos que controla que en todo el proyecto solo haya una instancia de dicho objeto, por eso el constructor del objeto es privado y es este método el encargado de crear la instancia en caso de que no exista y si ya existe devolver la existente.

2.4.1 Método Procesamiento

Este método es el encargado de rellenar cada una de las propiedades del objeto Datos por tanto es un método void. Primeramente en este método de crea una lista de Diccionarios llamada ListDicc que es la encargada de que mientras se itera para buscar terminosunicos guardar para cada documento un diccionario que contiene como clave las palabras de documento y como valor las veces que aparece dicha palabra en ese documento. Este metodo se basa en dos ciclos for, el primero itera por los archivos.txt y el segundo por cada una de las palabras de dicho documento; por tanto cuando se entra en el primer ciclo se crea un diccionario llamado Dicc que representa el documento en cuestión y que posteriormente se añadirá a ListDicc, luego se crea una variable de tipo string llamada contenido y representará el contenido del documento en cuestión usando el método ReadAllText de la clase File, también se crea una variable doc de tipo Documentos donde se va a guardar los datos necesarios de ese documento y por último se crea un array de tipo string llamado alltext que es el resultado del método Normaliza de la clase MetodosAdicionales al pasarle como parámetro la variable contenido y representa cada una de las palabras del documento despues de ser normalizado. Posteriormente se entra el en segundo ciclo for donde se itera por alltex donde se crea una variable de tipo int llamada cantveces y representa la cantidad de veces que esta una palabra en su documento, luego pasamos por una condicional if que retorna true si dicha palabra está contenida en Dicc y de ser así se le suma uno a la cantidad de veces que aparece esa palabra y se guarda en el diccionario. Ej:
if (Dicc.TryGetValue(alltext[j], out cantveces))

cantveces++;
Dicc[alltext[j]] = cantveces;

En caso de que dicha condicional retornara false quiere decir que es la primera vez que aparece la palabra y por tanto la agregamos al diccionario con valor 1, además como para cada palabra queremos asignarle un snippet aprovechamos esta ocasión para hacerlo porque en esta zona solo se entrará cuando sea la primera vez que encontramos una palabra y así evitamos estar modificando el snippet de la palabra cada vez que esta se encuentre, ese trabajo lo hacemos mediante el método SubString de la clase MetodosAdicionales para ello necesitamos cuatro parámetros, el primero será el array de string de donde se quiere extraer el ‘snippet‘ que en este caso es ‘alltext‘, el segundo será el int inicio que en este caso va a estar representado por el método ‘Max‘ de la

clase ‘Math‘ que recibirá como parámetros 0 y j-4(*queremos quedarnos para el ‘snippet‘ con 4 palabras antes y 4 palabras despues del ‘terminoBuscado‘*) utilizamos este método para asegurarnos que si el ‘terminoBuscado‘ se encuentra en las primeras posiciones no se vaya de rango ya que si es asi comenzará en la posición 0, análogamente para el tercer parámetro necesitamos un int ‘fin‘ que estará representado por el método ‘Min‘ de la clase ‘Math‘ que va a recibir como parámetros ‘doc.getContenido().Lenght()-1‘ y j+4, igual que para el parámetro anterior utilizamos este método para evitar irnos de rango, y nuestro cuarto parámetro será ‘terminoBuscado‘. Ej:
”’
int snippetStart = Math.Max(0, j - 4);
int snippetEnd = Math.Min(doc.getContenido().Length - 1, j + 4);
string snippet = MetodosAdicionales.SubString(alltext, snippetStart, snippetEnd, terminoBuscado);
”’
Luego guardamos dicho ‘snippet‘para el termino en el documento en cuestión(‘doc‘) utilizando los métodos getDictionary y Add. Ej:
doc.getDictionay().Add(alltext[j], snippet);
Luego como estamos iterando término a término realizamos otra condicional para saber si la lista terminosunicos contiene al término, en caso de no tenerlo se lo agregamos.
Luego cuando salimos del segundo for antes de pasar a otro archivo agregamos la variable doc al array resultadoDoc y además la variable Dicc a la lista ListDicc, y pasamos a analizar el siguiente archivo.
El siguiente trabajo de nuestro método es crear la Matriz de términos por documentos(TF) y contar en cuantos documentos aparece cada término único para posteriormente calcular el Idf, para ellos se crea una matriz llamada matrizTF donde cada fila representa un término único y cada columna un documento, un array de double llamado idf y un array de int llamado cantdocumentos, los dos arrays serán del tamaño de la lista terminosunicos. Para la creación de dicha matriz necesitamos dos ciclos for, el primero recorrerá archivos para poder tener acceso a los diccionarios de cada archivo que guardan la cantidad de veces que se repite cada una de las palabras del archivos y el segundo recorrerá terminosunicos para asignarle valor a la matriz en la posición correcta, creamos una variable int llamada cont inicializada en 0 y pasamos por una condicional que pregunta si el diccionario del archivo contiene a terminosunicos[j] y de ser asi que su valor se lo asigne a cont, luego estamos seguros de que ese término aparece en el documento y al array cantdocumentos[j] sumamos 1, y por último le asignamos el valor de dividir cont entre la cantidad de palabras del documento(fórmula de TF) a la matriz en la posición [j,i]. Por último queda darle valor a la propiedad matrizTfIdf y se realiza multiplicando la frecuencia de término por documentos por la frecuencia inversa. Para ello usamos dos ciclos for, el primero recorre terminosunicos y el segundo recorre archivos, primeramente se calcula la frecuencia inversa de la siguiente manera:
idf[i] = Math.Log2((double)archivos.Length + 1 / cantdocumentos[i]);
Luego se entra al segundo ciclo for y se le asigna el valor de multiplicar la ma-

trizTF[i,j] por idf[i] a la matrizTfIdf en la posición [i,j].Ej:
matrizTfIdf[i, j] = matrizTF[i, j] * idf[i];

3 Ejecución y funcionamiento del método Query

Lo primero que se le realiza a este método es agregar a él un nuevo parámetro de tipo Datos en el cual introducimos una instancia de dicho objeto dónde se llama al método Query desde el proyecto MoogleServer para que a la hora de realizar la búsqueda este contenga los datos necesario para calcular la similitud con los archivos. Al string query se le realiza el mismo procedimiento que a los archivos en la clase Datos, con ello nos referimos a normalizar la query así como crear un array llamado vectorconsulta del tamaño de la lista terminosunicos, este array va a contener el peso de cada uno de los elementos de la búsqueda, para ello se realiza el mismo procedimiento que en los documentos y la misma fórmula de TF.
El siguiente paso es calcular la similitud entre la consulta y cada uno de los documentos, para ello se utilizó la fórmula de similitud coseno la cuál se calcula como el cociente de el producto escalar y la multiplicación de la longitud del documento por la longitud de la query. Lo primero que se hace es crear una Lista llamada "a" que en ella se guardarán los documentos con score mayor que 0. Para esto se necesitan dos ciclos for, el primero recorre archivos y el segundo terminosunicos. Lo primero que se hace es llamar al método ResetScore y ResetSnippet que garantizan que cada vez que el usuario haga una nueva búsqueda no se vaya acumulando el score y el snippet de los documentos y se crean tres variables de tipo double llamadas productoEscalar, longitudDocumento y longitudQuery. Luego se entra al segundo ciclo for y se realizan los cálculos pertinentes para darle valor a cada una de esas variables. Luego de calculado la similitud para cada documento se agregan a la lista "a" los documentos con score mayor que 0 y posteriormente se organiza dicha lista de mayor a menor para dar los resultados.
Seguidamente creo un array de tipo 'SearchItem' llamado 'items' que tendrá como longitud el tamaño de la lista 'a', luego mediante un ciclo for que recorre 'a' le asigno a 'items' los valores de 'nombre', 'snippet' y 'score' mediante los métodos 'getNombre', 'getSnippet' y 'getScore' respectivamente todos de la clase 'Documentos'.
Por último el método 'Query' retorna un nuevo objeto de tipo 'SearchResult' que recibe como parámetros el array 'items' y el string 'query'.

4 Observaciones

- La demora de carga del proyecto depende de la cantidad de documentos con que se pruebe además de la cantidad de términos únicos con los que

cuente el corpus de documentos. El proyecto con una base de datos de 20MB carga aproximadamente en 5 minutos.

****Espero que este informe haya sido útil y que el proyecto haya alcanzado las espectativas.****