


Proyecto Moogle

Ronald Provance Valladares C121

Today

 **Buscar**

Resumen

Todo lo básico que debe conocer para poder usar **Moogle!**:

- Moogle! es una aplicación que tiene como objetivo facilitar la búsqueda en un conjunto de documentos.
- Luego de implementar el motor de búsqueda he decidido redactar este informe con el propósito de explicar la realización del proyecto.

Para llevar a cabo la tarea que se me ha asignado he utilizado lo que se conoce como modelo vectorial, que no es más que un modelo algebraico utilizado para el filtrado, indexado, recuperación y cálculo de relevancia de la información y además, mediante este modelo se pueden explotar las relaciones geométricas entre dos vectores documento (y términos) a fin de expresar las similitudes y diferencias entre términos.

1. Funcionamiento

Toda información solicitada será extraída de la colección de documentos (con extensión .txt) que se ingrese en la carpeta 'Content'.

- El motor recibe una entrada por parte del usuario, llamado 'query', el cual puede contener algunos operadores que llegarían a optimizar la búsqueda, estos operadores son:

- I. * este operador el usuario lo puede ingresar tantas veces como quiera antes de una palabra sin poner espacio, esto le dará mayor importancia o relevancia a los documentos que contengan esta palabra.
- II. ! el usuario debe colocarlo junto a la palabra, este negará (no devolverá) aquellos documentos que contengan esa palabra.

2. Clases

A nivel estructural este proyecto cuenta con varias clases que van a tener determinadas funciones en la búsqueda de la información, estas son:

2.1. La clase Implementation:

Esta clase va a tener la declaración de variables ‘generales’ y varios métodos:

1. Worktxt: en este método voy a operar sobre los documentos en la carpeta ‘Content’ para obtener su dirección y su título.

```
ziferences
public static (string[], string[]) Worktxt()
{
    List<string> nombres = new();

    //Modifico la direccion(por la del padre(de donde estoy corriendo))
    string currentadress = Directory.GetParent(Directory.GetCurrentDirectory()).FullName;
    string[] adresstxt = Directory.GetFiles(currentadress + @"Content");

    //lo convierto a lista para remover el .gitignore
    var convertirenlista = adresstxt.ToList();
    convertirenlista.Remove(currentadress + @"Content/" + ".gitignore");
    adresstxt = convertirenlista.ToArray();

    //sacar los nombres de los txt de las rutas
    string[] titletxt = new string[adresstxt.Length];
    for (int i = 0; i < titletxt.Length; i++)
    {
        //gracias a Path puedo obtener directamente el nombre del txt
        titletxt[i] = Path.GetFileNameWithoutExtension(adresstxt[i]);
    }
    return (adresstxt, titletxt);
}
```

Figura 1: Método Worktxt

2. WorkWords: en este método voy a recorrer todos los documentos y voy a dejar solo las palabras y números (elimino los caracteres como '.' y ',') y guardo en un diccionario las palabras con su frecuencia y en listas guardo los documentos sin los caracteres y las palabras de todos los documentos sin repetir.

```

//reference
public static (Dictionary<string, int>[], List<List<string>>, List<string>) WorkWords()
{
    string[] addressdoc = docConan.doc;
    (variable local) int canttxt;
    int canttxt = addressdoc.Length;

    Dictionary<string, int>[] Dictionarydoc = new Dictionary<string, int>[canttxt];

    List<List<string>> listtxtAllWords = new();

    List<string> wordsdontrepeat = new();

    for (int i = 0; i < canttxt; i++)
    {
        string words = "";

        listtxtAllWords.Add(new());

        Dictionarydoc[i] = new();
        //esta línea es para trabajar con documentos ANSI, los codifico a UTF8
        StreamReader lector = new StreamReader(addressdoc[i], System.Text.Encoding.UTF8);

        string textototal = lector.ReadToEnd().ToLower();
    }
}

```

Figura 2: Método WorkWords

3. TFxIDF: en este método voy a calcular el TF-IDF de las palabras de los documentos, nótese en el código del calculo que le sumo una cantidad casi nula al contador, esto es una medida para impedir una indefinición dentro del cálculo que deriva en errores de búsqueda (aquí dejo una referencia, para más información, pude consultar las páginas de Wikipedia).

$$TF = \frac{FP}{N}$$

FP- Frecuencia de la palabra (Repetición) N- Número total de palabras.

$$IDF = \log \left(\frac{CD}{ND} \right)$$

CD- Cantidad total de TXT ND- Número de documentos que contiene la palabra

TF-IDF (frecuencia de término – frecuencia inversa de documento): Es una medida numérica que expresa cuán importante es una palabra para un documento en una colección. Esta medida se utiliza a menudo como un factor de ponderación en la recuperación de información y la minería de texto. El TF-IDF aumenta proporcionalmente al número de veces que una palabra aparece en el documento, pero es compensada por la frecuencia de la palabra en la colección de documentos, lo que permite manejar el hecho de que algunas palabras son más frecuentes que otras.

```

public static Dictionary<string, double>[] TFxIDF()
{
    //TF= FRECUENCIA DE LA PALABRA (REPETICION)/ NUMERO TOTAL DE PALABRAS
    //IDF= LOG(NUMERO TOTAL DE TXT/ NUMERO DE DOCUMENTOS QUE CONTIENEN LA PALABRA)

    Dictionary<string, int>[] dictionarydoc = diccionario_global.Item1;

    Dictionary<string, double>[] tfidf = new Dictionary<string, double>[diccionario_global.Item2.Count];

    int canttxt = dictionarydoc.Length;

    for (int i = 0; i < canttxt; i++)
    {
        //convierto el diccionario en una lista
        tfidf[i] = new();
        var listadicdoc = dictionarydoc[i].ToList();
        int temporal = dictionarydoc[i].Count;
        for (int j = 0; j < temporal; j++)
        {
            int contador = 0;
            var palabra = listadicdoc[j];

            for (int h = 0; h < canttxt; h++)
            {
                if (dictionarydoc[h].ContainsKey(palabra.Key))
                {
                    contador++;
                }
            }
        }
    }
}

```

Figura 3: Método TFxIDF

4. QueryNormalizar: en este método voy a eliminar del query todo los caracteres que no sean letras o dígitos para posteriormente hacer su cálculo de TF-IDF, utilizando un método llamado TF e IDF de la query del que voy a obtener un diccionario con las palabras y su valor de TF-IDF asociado.
5. QueryVectorizado: este método lo utilizo para guardar en un 'array' solo los valores de TF-IDF de las palabras del query.
6. Matriztfidf: con este método voy a formar una matriz en la que voy a almacenar los valores de TF-IDF de las palabras del query en los documentos en los que aparezcan, si no aparecen les pongo valor 0.
7. SimilitudCos: en este método voy a calcular lo que se llama 'similitud del coseno' utilizando el query vectorizado y la matriz de TF-IDF antes calculados, y voy a obtener el 'score' de las palabras de los documentos con respecto al query ingresado, (dejo una referencia extraída de Wikipedia, para más información puede visitar la pagina oficial).

Similitud del coseno: Es una medida de la similitud existente entre dos vectores e un espacio que posee un producto interior con el que se evalúa el valor del coseno del ángulo comprendido entre ellos. Esta función trigonométrica proporciona un valor igual a 1 si en ángulo comprendido es cero, es decir si ambos vectores apuntan a un mismo lugar. Cualquier ángulo existente entre los vectores, el coseno arrojaría un valor inferior a uno. Si los vectores fuesen ortogonales el coseno se anularía, y si apuntasen en sentido contrario su valor sería -1. De esta forma, el valor de esta

```

1 reference
public static string[] QueryNormalizar(string query)
{
    List<string> palabrasquery = new List<string>();
    string palabra = "";
    for (int i = 0; i < query.Length; i++)
    {
        if (query[i] == ' ' && palabra == "")
        {
            continue;
        }
        if (Char.IsLetterOrDigit(query[i]))
        {
            palabra += query[i];
        }
        else if (query[i] == ' ' || palabra != "")
        {
            palabra = palabra.ToLower();
            palabrasquery.Add(palabra);

            palabra = "";
        }
    }
}

```

Figura 4: Método QueryNormalizar

```

1 reference
public static double[] Queryvectorizado(Dictionary<string, double> tfidfquery, string[] query)
{
    double[] vectorquery = new double[query.Length];

    for (int i = 0; i < vectorquery.Length; i++)
    {
        vectorquery[i] = tfidfquery[query[i]];
    }
    return vectorquery;
}

```

Figura 5: Método QueryVectorizado

métrica se encuentra entre -1 y 1, es decir en el intervalo cerrado $[-1,1]$.

2.2. La clase Levenshtein:

Esta clase es enteramente para la creación de las sugerencias del buscador a través del cálculo de distancias de Levenshtein, (debajo referencia extraída de Wikipedia).

Levenshtein: Es el número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra, se usa ampliamente en teoría de la información y ciencias de la computación. Se entiende por operación, bien una inserción, eliminación o la sustitución de un carácter. Este cálculo es muy útil para determinar cuan similares son dos cadenas de palabras y la empleo en mi proyecto con el fin de dar una sugerencia lo más acertada posible en caso de errores ortográficos en el query.

```

1 reference
public static double[,] Matriztfidf(string[] query)
{
    //esta va a ser mi cant de filas en la matriz
    Dictionary<string, double>[] tfidfdoc = tf_global;
    double[,] matriztfidf = new double[tfidfdoc.Length, query.Length];

    for (int i = 0; i < tfidfdoc.Length; i++)
    {
        for (int j = 0; j < query.Length; j++)
        {
            if (tfidfdoc[i].ContainsKey(query[j]))
            {
                matriztfidf[i, j] = tfidfdoc[i][query[j]];
            }
            else
            {
                matriztfidf[i, j] = 0;
            }
        }
    }
    return matriztfidf;
}

```

Figura 6: Método Matriztfidf

Y aquí un pedazo de código de la clase que es el método calculo para calcular el porcentaje de cambio en la palabra

2.3. La clase Matriz

Aquí lleve las operaciones con matrices a la programación, además utilizo la multiplicación de matrices en el proyecto, ya que multiplico una matriz por un vector, que no es más que una matriz de una sola columna (fila)

2.4. La clase Operadores:

En esta clase se encuentran los métodos con los que voy a definir la modificación sobre el 'score' en base a los operadores ingresados

- ★ En el caso del operador (!) uso el método 'No Aparece' los documentos que contengan a la palabra con este operador decidí, para que no aparezcan por ningún motivo, hacer su score 0, y de esta forma no presentarlo en los resultados de búsqueda.
- ★ En el caso del operador (*) uso el método 'Asterisco' decidí aumentar la relevancia de los documentos que contienen esa palabra al multiplicar el Score por la cantidad de (*) ingresados aumentado en 1.

```

1 reference
public static float[] SimilitudCos(double[] vectorquery, double[,] matriztfidf)
{
    float[] Score = new float[matriztfidf.GetLength(0)];

    double sumatoriaquery = 0;
    for (int i = 0; i < vectorquery.Length; i++)
    {
        sumatoriaquery += Math.Pow(vectorquery[i], 2);
    }
    sumatoriaquery = Math.Sqrt(sumatoriaquery);

    for (int j = 0; j < matriztfidf.GetLength(0); j++)
    {
        double sumatoriamatriz = 0;
        double subscore = 0;

        for (int i = 0; i < vectorquery.Length; i++)
        {
            sumatoriamatriz += Math.Pow(matriztfidf[j, i], 2);
            subscore += vectorquery[i] * matriztfidf[j, i];
        }
        sumatoriamatriz = Math.Sqrt(sumatoriamatriz);
        Score[j] = (float)((double)subscore / ((double)(sumatoriamatriz * sumatoriaquery) + 0.0000001));
    }
    return Score;
}

```

Figura 7: Método SimilitudCos

2.5. La clase Snippet:

En esta clase voy a tener un único método que será el encargado de construir un fragmento del documento donde se encuentren la(s) palabra(s) del query, este fragmento servirá como una breve introducción al contenido del documento.

2.6. La clase Moogle:

En esta clase se encuentran los llamados a las otras clases y métodos antes mencionados para hacer funcionar el buscador, además de un algoritmo de ordenación de los títulos de los documentos y el score para que sean devueltos en orden de mayor a menor score.

Esta imagen muestra la llamada de métodos:

A rasgos generales estas son las características de mi proyecto, espero lo disfrute.

¡Muchas Gracias!

```

class Levenshtein
1 reference
private static double Calculo(string palabra1, string palabra2)
{
    int costo = 0;
    int m = palabra1.Length;
    int n = palabra2.Length;
    // d es una tabla con m+1 renglones y n+1 columnas
    int[,] d = new int[m + 1, n + 1];
    // Llena la primera columna y la primera fila.
    for (int i = 0; i <= m; d[i, 0] = i++) ;
    for (int h = 0; h <= n; d[0, h] = h++) ;
    /// recorre la matriz llenando cada uno de los pesos.
    /// i columnas, j renglones
    for (int i = 1; i <= m; i++)
    {
        // recorre para j
        for (int j = 1; j <= n; j++)
        {
            /// si son iguales en posiciones equidistantes el peso es 0
            /// de lo contrario el peso suma a uno.
            costo = (palabra1[i - 1] == palabra2[j - 1]) ? 0 : 1;
            d[i, j] = System.Math.Min(System.Math.Min(d[i - 1, j] + 1, //Eliminacion
                d[i, j - 1] + 1), //Insercion
                d[i - 1, j - 1] + costo); //Sustitucion
        }
    }
    /// Calculamos el porcentaje de cambios en la palabra.
    if (palabra1.Length > palabra2.Length)
        return ((double)d[m, n] / (double)palabra1.Length);
    else
        return ((double)d[m, n] / (double)palabra2.Length);
}

```

```

0 references
public static Matrix operator +(Matrix matriz1, Matrix matriz2)
{
    if (matriz1.filas != matriz2.filas || matriz1.columnas != matriz2.columnas)
    {
        throw new ArgumentException("Las matrices deben tener las mismas dimensiones");
    }

    int[,] resultado = new int[matriz1.filas, matriz1.columnas];

    for (int i = 0; i < matriz1.filas; i++)
    {
        for (int j = 0; j < matriz1.columnas; j++)
        {
            resultado[i, j] = matriz1.matriz[i, j] + matriz2.matriz[i, j];
        }
    }

    return new Matrix(resultado);
}

```

Figura 8: Clase Matriz


```

//operador ('!') su funcion es hacer 0 el score de los documentos donde esta la palabra con el operador
1 reference
public static void NoAparece(string palabra, float[] score, List<List<string>> documentos)
{
    for (int i = 0; i < documentos.Count; i++)
    {
        if (documentos[i].Contains(palabra))
        {
            score[i] = 0;
        }
    }
}

```

Figura 9: Operador "No Aparece"

```

//operador (**) mi forma de hacerlo es multiplicar el score por la cantidad de (**) mas uno para ir dandole mas relevancia
1 reference
public static void Asterisco(string palabra, float[] score, List<List<string>> documentos, int contador)
{
    for (int i = 0; i < documentos.Count; i++)
    {
        if (documentos[i].Contains(palabra))
        {
            score[i] = score[i] * (contador + 1);
        }
    }
}

```

Figura 10: Operador "Asterisco"

```

1 reference
public class Snippet
{
    //Este metodo me va a retornar un fragmento del txt donde aparecen las palabras del query
    1 reference
    public static string BuscarSnippet(List<string> documentos, string palabra)
    {
        string parte = "";
        string parte1 = "";
        string parte2 = "";
        for (int j = 0; j < documentos.Count; j++)
        {
            if (documentos[j] == palabra && parte == "")
            {
                int minpalabras = 15;
                for (int k = j; k >= 0; k--)
                {
                    parte1 = documentos[k] + " " + parte1;
                    minpalabras--;
                    if (minpalabras == 0)break;
                }
                int maxpalabras = 14;
                for (int k = j + 1; k < documentos.Count; k++)
                {
                    parte2 += " " + documentos[k];
                    maxpalabras--;
                    if (maxpalabras == 0)break;
                }
                parte = parte1 + parte2;
            }
        }
        return parte;
    }
}

```

Figura 11: Clase Snippet

```

1 reference
public static SearchResult Query(string query)
{
    string[] titulo_txt = Implementation.Worktxt().Item2;
    (Dictionary<string, int>[], List<List<string>>, List<string>) trabajo_con_palabras = Implementation.diccionario_global;
    string[] query_normalizado = Implementation.QueryNormalizar(query);
    Dictionary<string, double> tf_idf_query = Implementation.TFIDFQuery(query_normalizado);
    double[] vector_query = Implementation.Queryvectorizado(tf_idf_query, query_normalizado);
    double[,] matriz_tf_idf = Implementation.Matriztfidf(query_normalizado);
    float[] similitud_cos = Implementation.SimilitudCos(vector_query, matriz_tf_idf);
    string[] sugerencias = Levenshtein.Distancia(query_normalizado, trabajo_con_palabras.Item3);
    string sugerencia = "";
    for (int i = 0; i < sugerencias.Length; i++)
    {
        sugerencia += sugerencias[i] + " ";
    }
    float[] score = Operadores.verqueryoriginal(query, similitud_cos, trabajo_con_palabras.Item2);
}

```

Figura 12: Clase Moogole