Facultad de Matématica y Computación
Universidad de la Habana

Proyecto de Programación I
Moogle!

Ernesto Abreu Peraza
Grupo: C-121
Curso: 2023

Descripción

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... ehem... casi implementada la lógica del algoritmo de búsqueda.

Correr y usar el proyecto

Para correr el proyecto debes usar el comando dotnet watch run –project MoogleServer en Windows y make dev en Linux. En la carpeta Content deberán aparecer los documentos (en formato *.txt) en los que el usuario va a realizar la búsqueda. En la casilla donde aparece *Introduzca la búsqueda* el usuario va a escribir que desea buscar y basta con apretar el botón *Buscar* para que Moogle! haga su trabajo.

Arquitectura del proyecto

Características de clases

- Moogle: procesar consulta.
- TF_IDF: calcular tf-idf para los documentos y para la query, calcular coseno del angulo entre dos vectores.
- DocumentReader: leer, normalizar y obtener de los documentos, título, texto y palabras.
- StringUtil: procesar, calcular, modificar y obtener informacion a partir de texto util para alguna de las funcionalidades de la aplicación.
- Matrix: Definir y multiplicar matrices
- Vector: Definir, calcular modulo y multiplicar vectores

Precálculo

Term frequency-Inverse document frequency (TF_IDF) es una medida numérica que expresa cuan relevante es una palabra para un documento en una colección de documentos.

$$TF\_IDF[t,d] = TF[t,d] * IDF[t]$$
$$TF[t,d] = \frac{f[t]}{maxFrequency}$$
$$IDF[t] = \log_{10} \frac{numberOfDocuments}{Df[t]}$$

Siendo t una palabra y d un documento. $f[t]$ la frecuencia de la palabra t en el documento d y $maxFrequency$ es el máximo de los $f[t]$. $numberOfDocuments$ el total de documentos y $Df[t]$ es la cantidad de documentos en los que aparece la palabra t.

- Cuando el programa empieza a correr se ejecuta el método `TF_IDF.Compute()` del archivo `TF_IDF.cs`. Este método lee el texto de cada documento que aparece en la carpeta Content y de el extrae todas las palabras. Se calcula el `TF_IDF` para cada palabra en cada documento

```
public static void Compute()
{
    /* Precalcula el TF_IDF */

    /* Arreglo con los nombres de los documentos */
    documentsName = DocumentReader.DocumentsNameList("..\\Content");

    int numberOfDocuments = documentsName.Length;

    /* Diccionario [nombre de documento => indice] */
    for (int i = 0; i < numberOfDocuments; i++)
    {
        Document[documentsName[i]] = i;
    }

    /* Arreglo con las palabras del los documentos */
    words = DocumentReader.WordsList(documentsName);

    int numberOfWords = words.Length;

    /* Diccionario [palabra => indice] */
    for (int i = 0; i < numberOfWords; i++)
    {
        Word[words[i]] = i;
    }

    Matrix TF = ComputeTF();
    tf = TF;

    Vector IDF = ComputeIDF();
    idf = IDF;

    Matrix TF_IDF = new Matrix(numberOfWords, numberOfDocuments);

    /* Multiplicar TF por IDF */
    for (int j = 0; j < numberOfDocuments; j++)
        for (int i = 0; i < numberOfWords; i++)
            TF_IDF[i, j] = TF[i, j] * IDF[i];

    tf_idf = TF_IDF;
}
```

- Los métodos `ComputeTF()` y `ComputeIDF()` calculan el TF y el IDF respectivamente.

Consulta

- Luego que la aplicación inicie, cuando se realize una consulta (búsqueda), el proyecto llama al método `Moogle.Query()` del archivo `Moogle.cs` donde se calcula el TF_IDF para la consulta a traves del método `TF_IDF.ComputeQueryTF_IDF()` y se compara con el TF_IDF de cada documento dandole una puntuación a cada uno y devolviendo una lista con los nombres de los documentos que más relevancia tienen. La puntuación sería el coseno del ángulo entre el vector formado con el TF_IDF de la consulta y el TF_IDF del documento. La similitud de los documentos depende es mayor mientras mas se acerca a 1 el coseno del angulo entre ellos, visto de otra forma mientras mas pequeño es el angulo entre ambos. Para esto se utilizan dos fórmulas de dot product. Para dos vectores a y b: $dotProduct = a_1 * b_1 + a_2 * b_2 + ... + a_n * b_n$, $dotProduct = \cos(a, b) * |a| * |b|$.

```
public static (float, int)[] VectorialModel(Vector queryTF_IDF)
{
    /* Devuelve un arreglo que contiene para cada documento el coseno del angulo
    entre el vector de la query y el del documento */
    (float, int)[] vectorialModel = new (float, int)[tf_idf.columns];
    for (int j = 0; j < tf_idf.columns; j++)
    {
        Vector v = new Vector(tf_idf.rows);
        for (int i = 0; i < tf_idf.rows; i++)
        {
            v[i] = tf_idf[i, j];
        }
        float score = 0;
        if (queryTF_IDF.Module() * v.Module() != 0)
        score = Vector.Dot_Product(queryTF_IDF, v) / (queryTF_IDF.Module() *
        v.Module());
        vectorialModel[j] = (score, j);
    }
    return vectorialModel;
}
...
...
...
/* Definicion de producto escalar entre dos vectores */
static public float Dot_Product(Vector a, Vector b)
{
    if (a.Dimensions != b.Dimensions)
        /* Exception */
        return 0;

    float dot_product = 0;
    for (int i = 0; i < a.Dimensions; ++i)
        dot_product += a[i] * b[i];

    return dot_product;
}
...
...
...
/* Definicion de modulo de un vector */
```

```
public float Module()
{
    float module = 0;
    for (int i = 0; i < this.Dimensions; i++)
    {
        module += this[i] * this[i];
    }
    return (float)Math.Sqrt(module);
}
```

- También se muestra un fragmento por cada documento donde se puede apreciar en este una relación del documento con la consulta.

Funcionalidades extras

Otra funcionalidad del Moogle! es que dará una sugerencia de búsqueda en caso de que el usuario quizás cometió un error al escribir la consulta. Para esto usamos un algoritmo de Edit Distance conocido como Levenshtein distance y a este costo le dividimos por el Longest Common Prefix. El propósito de este último proviene de la idea de que es más probable equivocarse en las últimas letras que en las primeras. Así distra está más cerca de dijkstra que de citra.

```
public static float Distance(string a, string b)
{
    /* Calcula la distancia entre dos cadenas de caracteres */
    return EditDistance(a, b) / LongestCommonPrefix(a, b);
}

public static float EditDistance(string a, string b)
{
    /* Calcula el EditDistance para dos cadenas de caracteres */
    int[,] dp = new int[a.Length + 1, b.Length + 1];
    for (int i = 0; i <= a.Length; i++)
        for (int j = 0; j <= b.Length; j++)
        {
            if (i == 0 || j == 0)
            {
                dp[i, j] = Math.Max(i, j);
            }
            else
            {
                dp[i, j] = Int32.MaxValue;
                if (a[i - 1] == b[j - 1])
                   dp[i, j] = dp[i - 1, j - 1];
                else
                {
                    dp[i, j] = Math.Min(dp[i, j], dp[i - 1, j] + 1);
                    dp[i, j] = Math.Min(dp[i, j], dp[i, j - 1] + 1);
                    dp[i, j] = Math.Min(dp[i, j], dp[i - 1, j - 1] + 1);
                }
            }
        }
    return (float)dp[a.Length, b.Length];
}

```

```
34 public static float LongestCommonPrefix(string a, string b)
35 {
36     /* Calcula el LongestCommonPrefix para dos cadenas de caracteres */
37     for (int i = 0; i < Math.Min(a.Length, b.Length); i++)
38         if (a[i] != b[i])
39             return (float)(i + 1);
40 
41     return (float)Math.Min(a.Length, b.Length) + 1;
42 }
```