Proyecto Moogle!

Nombre: Karell Javier Delgado Benitez
Grupo: C122

Después de leer el archivo readme.txt del repositorio donde se encuentra el proyecto, se pueden entender las características y funcionalidades del mismo. Ahora, explicaré el flujo del proyecto.

Primero, crearemos la clase ProcessedDocuments.cs para procesar los documentos. Esta clase contiene dos propiedades públicas: UniversoDePalabras y Matriz. La primera propiedad se asocia a un diccionario de string y float que contiene todas las palabras de los documentos con su respectivo IDF. La segunda propiedad se asocia a un diccionario de diccionarios, donde la clave es el nombre del documento y el valor es un diccionario de string y float que representa las palabras con su respectivo TF-IDF.

La clase consta de tres métodos: Proceso(string texto), que recibe un string como parámetro y devuelve el texto normalizado sin caracteres especiales; Obtener palabras, que recibe un string texto y un booleano para indicar si es un documento o no. Si es un documento, lo lee usando StreamReader; si no lo es, simplemente lo normaliza; y AnalisisDocumentos, que se encarga de rellenar los diccionarios anteriores con su respectivo TF-IDF y su IDF.
Proceso(string texto)

```
5 references
public static string[] Proceso(string texto)
{
    string PalabrasLinea;
    PalabrasLinea = Regex.Replace(texto.ToLower(), "á","a")
        .Replace("é","e")
            .Replace("í","i")
                .Replace("ó","o")
                    .Replace("ú","u")
                        .Replace("ñ","n");
    return Regex
        .Replace(PalabrasLinea,@"[^a-z0-9 ]+"," ")
        .Split(' ',StringSplitOptions.RemoveEmptyEntries);
}
```

Luego, estas propiedades se pasan a index.razor para que se carguen antes de iniciar el documento y se le pasen estos parámetros al método Query que se ejecuta en Moogle.cs.

Después, creamos la clase Query.cs, que recibe como argumento en el constructor un string y el diccionario Universo. En esta clase, se obtienen los operadores, el TF-IDF de la query y en el operador de importancia se le aumenta la importancia del valor en la misma.

A continuación, creamos la clase CosineSimilarity.cs, a la cual se le pasa una instancia de la clase Query y los diccionarios mencionados anteriormente para poder calcular la similitud que existe entre los vectores documento y el vector query. Luego, se ordena y se obtienen los documentos más interesantes. En esta clase, se procesan los operadores para saltar los documentos del operador !, asegurarse de que aparezcan ^ y demás. Uno de los operadores más interesantes es el de cercanía ~, que funciona leyendo el documento para ir comparando las cercanías. En caso de que las palabras tengan la cercanía máxima, se detiene el método.

```
1 reference
private static Dictionary<string, float> ActualizarPuntuacionCercania(Dictionary<string, float>
DocumentosPuntuados, List<string[]> PalabrasCercanas, float PromedioScoreFaltante,
Dictionary<string, Dictionary<string, float>> MatrizDocumentos)
{
    foreach (string Documento in DocumentosPuntuados.Keys)
    {
        float PromedioCercania = 0;

        foreach (string[] ParesCercanos in PalabrasCercanas)
        {
            string Palabra1 = ParesCercanos[0];
            string Palabra2 = ParesCercanos[1];

            if(MatrizDocumentos[Documento].ContainsKey(Palabra1) && MatrizDocumentos[Documento].
            ContainsKey(Palabra2))
            {
                PromedioCercania += CercaniaPalabras(Documento, Palabra1, Palabra2);
            }
        }

        if(PromedioCercania != 0)
            PromedioCercania = PromedioCercania / (float)(PalabrasCercanas.Count);

        DocumentosPuntuados[Documento] += PromedioCercania*PromedioScoreFaltante;
    }

    return DocumentosPuntuados;
}
```

Por último, el Snippet recibe la Query y los documentos puntuados propiedad de la similitud del coseno. Este proceso completo toma cada documento y lo divide en pequeños subdocumentos para quedarse con el más similar a la Query.