Informe de Proyecto

(por Gabriel Andrés Pla Lasa)

Introducción

En el presente informe se detallará la realización del "Moogle", una aplicación web de búsqueda matricial.

Esta aplicación buscará palabras o frases dentro de un contenedor que tiene una serie de documentos con extensión txt, y como resultado devolverá los documentos contengas esas palabras o frases. Para la optimización de la búsqueda se nos sugiere una serie de funcionalidades como por ejemplos: la utilización de modelos vectoriales, la introducción de operadores de consulta, etc.

Con este objetivo en mente, mi aplicación procesará los documentos antes de iniciar, es decir, que se calculará previamente una serie de valores necesarios e imprescindibles para la búsqueda. De esta tarea se encarga la clase *Terms*. Una clase que obtendrá una lista general de todas las palabras de los documentos, la cual se transformará en una matriz de TF-IDF (Term Frequency – Inverse Document Frequency).

Para la búsqueda realizada por un usuario (query) se dispone de una clase *TextProcessor*, la cual calculara un score por cada documento, donde ese score es la similitud que posee la query con un documento determinado. Y al final se devolverá como resultado de la búsqueda los documentos mas relevantes para la query.

Inicio del programa

Para que el programa procese todos los documentos .txt, que hay en la carpeta "Content", antes que la aplicación inicie, en la clase *MoogleServer.Program* se inicializara una variable estática llamada ***textProcesor*** que pertenece a la clase *MoogleEngine.TextProcessor* y está declarada como campo de la clase *MoogleEngine.Moogle*.

```
25  app.MapFallbackToPage("/_Host");
26
27  // Variable estatica de la clase TextProcessor que procesara los documentos .txt de la carpteta Content
28  MoogleEngine.Moogle.textProcessor = new MoogleEngine.TextProcessor();
29
30  app.Run();
```

Constructor de la clase *TextProcessor*

Cuando la variable ***textProcesor*** se inicialice, el constructor de la clase *MoogleEngine.TextProcessor* procesara los documentos .txt, que hay en la carpeta "Content", siguiendo los siguientes pasos:

1. Se guardará en una variable llamada ***path*** la ruta absoluta de la carpeta "Content".
2. Se obtendrán los nombres de los documentos .txt que haya en las carpetas y posibles subcarpetas, y se guardarán en un array de *strings*.
3. Se inicializará el campo de la clase *TextProcessor* de tipo *Dictionary* llamado ***documents*** y se guardarán los nombres de los documentos .txt sin extensión de archivo (key) con su correspondiente texto (value).
4. Se inicializará el campo de la clase *TextProcessor* de tipo *Terms* llamado ***terms*** y se le pasará como parámetro al constructor el diccionario ***documents.*** En esta clase se obtendrán las palabras normalizadas y la matriz de TF-IDF.
5. Se obtendrá la matriz de TF-IDF de tipo *float[,]* a través del método *MoogleEngine.Terms.GetTFIDFMatrix()* y se guardara en el campo ***tfidfMatrix*** de la clase *MoogleEngine.TextProcessor*.

```
1 referencia | 0 cambios | 0 autores, 0 cambios
public TextProcessor()
{
    // Crear una variable para guardar la ruta al fichero Content
    string path = Path.Combine(Directory.GetParent(".")!.ToString(), "Content");

    // Obtener los nombres de los documentos .txt del path
    string[] documentsNamesWithPath = documentsNamesWithPath = Directory.GetFiles(path, "*txt", SearchOption.AllDirectories);

    // Guardar los nombres (key) y su correspondiente texto (value) de los documentos .txt en un diccionario
    documents = new Dictionary<string, string>();
    for (int i = 0; i < documentsNamesWithPath.Length; i++)
        documents.Add(Path.GetFileNameWithoutExtension(documentsNamesWithPath[i]), File.ReadAllText(documentsNamesWithPath[i]));

    // Procesar los documentos y crear la matrix de TF-IDF
    terms = new Terms(documents);
    tfidfMatrix = terms.GetTFIDFMatrix();

}
```

Clase *Terms*

En esta clase se obtendrán las palabras normalizadas y la matriz de TF-IDF. El TF-IDF de una palabra se calcula a través de la siguientes formulas:

$$tf(t,d) = \frac{f(t,d)}{\max\{f(d)\}}$$

$$idf(t,D) = \log\frac{D}{D(t)}$$

$$tfidf(t,d,D) = tf(t,d) * idf(t,D)$$

donde:

- $f(t,d)$: frecuencia de la palabra t en el documento d
- $\max\{f(d)\}$: máxima frecuencia de una palabra en el documento d
- D: número de documentos de la colección
- $D(t)$: número de documentos donde aparece la palabra t

Y con el objetivo de obtener los valores que permitirán calcular la matrix de TF-IDF, se declararon una serie de campos en la clase *Terms*:

```
13      // Una matrix de tipo float que guardara el TF-IDF
14      float[,] tfidfMatrix;
15
16      // Lista que guardara las palabras normalizados de todos los documentos (sin repetir)
17      List<string> termsNormalized;
18
19      /* Diccionario que tiene como
20          - Key: los nombres de los documentos .txt
21          - Value: el texto de cada documentos .txt  */
22      Dictionary<string, string> documents;
23
24      /* Diccionario que tiene como
25         - Key: las palabras normalizados de todos los documentos (sin repetir)
26         - Value: el numero de documentos donde aparece la palabra  */
27      Dictionary<string, int> termsFrequencyPerDocument;
28
29      /* Diccionario que tiene como
30        - Key: los nombres de los documentos .txt
31        - Value: la frecuenca maxima de palabras de cada documento   */
32      Dictionary<string, int> maxTermsFrequencyOfEachDocument;
33
34      /* Diccionario que tiene como
35        - Key: los nombres de los documentos .txt
36        - Value: otro diccionario que tiene como
37              ~ SubKey: las palabras normalizadas del documento correspondiente de la key
38              ~ SubValue: una lista con las posiciones donde aparece la palabra (de la subkey) dentro del documento */
39      Dictionary<string, Dictionary<string, List<int>>> termsLineAndFrequency;
40
```

En el constructor de la clase se inicializaran todos estos campos y se llamará al método de instancia *NormalizeAndFrequency(),* en el cual iterará por cada documento, con el objetivo de obtener los *key* y *values* de los diccionarios. Por cada iteración, se añadirá a ***maxTermsFrequencyOfEachDocument*** como KeyValuePair el título del documento y un 0. También, se llamara al método *Normalizer()* el cual devolverá el *value* del diccionario ***termsLineAndFrequency*** . Además, se chequeará cuando un documento está vacío, que en caso de cumplirse se eliminará el documento de los diccionarios que lo tengan como valor, y así los valores de la matriz de TF-IDF, que se computarán más tarde, no serán afectados innecesariamente. Por último, se obtendrán los términos normalizados de la key de ***termsFrequencyPerDocument.***

```
private void NormalizeAndFrequency()
{
    // Iterar sobre los documentos
    foreach (var title in documents.Keys)
    {
        // Normalizar, obtener las frecuencias y llenar los diccionarios
        maxTermsFrequencyOfEachDocument.Add(title, 0);
        termsLineAndFrequency.Add(title, Normalizer(documents[title]));

        // Caso especifico cuando se procesa un documento vacio
        if (termsLineAndFrequency[title].Count == 0)
        {
            documents.Remove(title);
            termsLineAndFrequency.Remove(title);
            maxTermsFrequencyOfEachDocument.Remove(title);
        }
    }

    // Obtener las palabras normalizadas
    termsNormalized = termsFrequencyPerDocument.Keys.ToList();
}
```

El método *Normalizer(),* tiene como objetivo: crear una colección de palabras normalizadas y a su vez computar la diferentes frecuencias reflejadas en los diccionarios declarados como campos. Para ello se pasará como parámetro del método el texto de un documento. Y dentro de él se declarará una variable de tipo *Dictionary* llamada ***terms*** que será devuelto al final de la ejecución del método. Y también hay declarada una variable de tipo *StringBuilder* denominada ***actualWord***.

En este método *Normalizer()* se iterará por cada carácter del texto y se comprobará si es de tipo letra o número. En caso afirmativo se normalizará, es decir, en caso que lo necesite se convertirá en minúscula y se eliminará cualquier tipo de acento; y por último se añadirá a la variable ***actualWord*** a través del método *Append()*. En caso contrario se llamará a una función local llamada *Check(),* al cual se le pasaran como parámetros la variable ***actualWord*** y la posición en el texto (que será el número de iteración actual).

En el *Check()*, ***actualWord*** se convertirá a *string* y se efectuaran las siguientes operaciones condicionales:

1º. Se comprobará que ***word*** no sea un *string* nulo o vacío.
2º. Luego se chequeará si el diccionario ***terms*** tiene un key ***word*** y se añadirá a la lista, que posee como value, la posición en el texto.
3º. En caso que ***terms*** no tenga ningún key ***word***, se verificará que ***termsFrequencyPerDocument*** lo tenga como key también, y se le adicionará 1 a su valor. Además, es necesario resaltar que ***termsFrequencyPerDocument*** tendrá como llave (en general) las palabras normalizadas de los documentos, es decir, será las columnas de la matriz de TF-IDF.
4º. Finalmente se comprobará el diccionario ***maxTermsFrequencyOfEachDocument***, es decir si es menor el value actual de ***maxTermsFrequencyOfEachDocument*** comparado con la frecuencia en que aparece una palabra en un documento.

```
// Metodo auxiliar que llenarara los diferentes diccionario cuando un caracter no sea una letra o un numero
void Check(StringBuilder actualWord, int i)
{
    string word = actualWord.ToString();
    if (!string.IsNullOrWhiteSpace(word))
    {
        // Para termsLineAndFrequency
        if (!terms.ContainsKey(word))
        {
            terms.Add(word, new List<int>());

            // Para termsFrequencyPerDocument
            if (termsFrequencyPerDocument.ContainsKey(word))
            {
                termsFrequencyPerDocument[word]++;
            }
            else
            {
                termsFrequencyPerDocument.Add(word, 1);
            }

        }

        terms[word].Add(i);

        // Para maxTermsFrequencyOfEachDocument
        if (maxTermsFrequencyOfEachDocument.Values.Last() < terms[word].Count)
            maxTermsFrequencyOfEachDocument[maxTermsFrequencyOfEachDocument.Keys.Last()] = terms[word].Count;

        actualWord.Clear();
    }
}
```

Luego de finalizar el método *NormalizeAndFrequency()* y haberse obtenido los respectivos key y values de todos los diccionarios, en el constructor de la clase *Terms* se prosigue, finalmente, a calcular la matriz de TF-IDF a través del método *CalculeteTFIDFMatrix()*. En este método se iterará por los documentos (filas) y las palabras normalizadas (columnas), para obtener el TF-IDF del elemento correspondiente de la matriz ***tfidfMatrix***, utilizando las fórmulas expuestas anteriormente con los valores de los diccionarios. Y así, finaliza la ejecución del constructor.

Además, en esta clase *Terms*, se tienen diferentes métodos que devuelven los valores de los campos y un método llamado *GetQueryVector()*, el cual será explicado junto con la query.

```
// Devuelve el indice de una palabra que se encuentra en la lista termsNormalized
1 referencia | 0 cambios | 0 autores, 0 cambios
public int GetTermIndex(string term)...

// Devuelve la frecuencia de una palabra de un documento
1 referencia | 0 cambios | 0 autores, 0 cambios
public int GetFrequency(string title, string term)...

// Devuelve como un array la lista de las posiones de una palabra en un documento
1 referencia | 0 cambios | 0 autores, 0 cambios
public int[] GetPositions(string title, string term)...

// Devuelve la matrix de TFIDF
1 referencia | 0 cambios | 0 autores, 0 cambios
public float[,] GetTFIDFMatrix()...
```

Y de esta forma, se termina de procesar los documentos .txt, la aplicación inicia y está a disposición de las búsquedas que haga el usuario.

La Query y la clase *TextProcessor*

Cuando el usuario hace la búsqueda se lanza el método *Query()* de la clase *MoogleEngine.Moogle*, en la cual, esta declarada una variable de tipo *SearchItem[]* llamado ***ítems***. Esta variable obtiene su valor (el resultado de la búsqueda) a través de un método de instancia denominado *VectorialModel()* perteneciente al objeto ***textProcesor***, el cual fue inicializado al empezar la aplicación.

Al método *VectorialModel()* se le pasa como parámetro un *string* ***query*** con la búsqueda realizada por el usuario. En este método, primeramente, se obtienen los valores de los campos ***queryVector*** y ***queryTerms*** de la clase *TextProcessor,* a traves al método *GetQueryVector()* de la clase *Terms*. Es decir, en este método se devuelve el vector, que contiene los TF de la query, y las palabras normalizadas de la query que aparezcan en al menos un documento, utilizando de por medio el método *Normalizer()* ante descrito.

Luego en el método *VectorialModel()* se llama al método *CosineSimilarity()*, al cual se le pasan dos parámetros: los campos ***tfidfMatrix*** y ***queryVector***. Para luego efectuarse una serie de cálculos:

1º. Se multiplicaran la matriz ***tfidfMatrix*** con el vector ***queryVector*** (considerado como una matriz columna) utilizando el método estático *Multiplication()* de la clase *Matrix*. En este método primero se comprueba que haya tantas columnas en la matriz como filas en el vector (matriz columna). Y por último se multiplican dando como resultado un vector (matriz columna). La multiplicación está dada por la siguiente formula: $\sum_{k=0}^{p} M_{ik} V_{kp}$, donde M_{ik}, V_{kp}, p, i representan respectivamente a un elemento de la matriz, un elemento del vector, las filas y las columnas. Estos valores se guardan en un array llamado ***vectorResultante.***

2º. Se obtiene la norma tanto de la matriz ***tfidfMatrix*** como la del vector ***queryVector***, que en este caso se considerara como una matriz fila. Estas dos normas se calculan por un método estático llamado *Norma()*, uno declarado en la clase *Matrix* y otro en la clase *Vector*. La norma de una matriz fila puede calcularse como: $\sqrt{\sum_{k=0}^{p} a_k}$, donde p son las columnas y a_k un elemento de la matriz fila. Por tanto, al final se obtendrán un vector (matriz columna) por parte de ***tfidfMatrix*** llamado ***normaMatrix*** y un número por parte ***queryVector*** denominado ***normaQuery.***

3º. Por último el método *VectorialModel()* devuelve un array (vector) con la similitud de coseno, la cual se calcula dividiendo cada elemento del ***vectorResultante*** por la multiplicación de la ***normaQuery*** con el correspondiente elemente en ***normaMatrix.***

```
1 referencia | 0 cambios | 0 autores, 0 cambios
private float[] CosineSimilarity(float[,] tfidfMatrix, float[] queryVector)
{
    // Calcular el vector resultante de la multiplicacion de la matriz de TF-IDF con
    float[] vectorResultante = Matrix.Multiplication(tfidfMatrix, queryVector);

    // Calcular la norma
    float[] normaMatrix = Matrix.Norma(tfidfMatrix);
    float normaQuery = Vector.Norma(queryVector);

    // Calcular la similutd de coseno
    float[] vectorSimilitud = new float[documents.Count];
    for (int i = 0; i < documents.Count; i++)
    {
        vectorSimilitud[i] = vectorResultante[i] / (normaMatrix[i] * normaQuery);
    }

    return vectorSimilitud;
}
```

Después de finalizar el método *CosineSimilarity()* se obtiene un número (score) por cada documento entre 0 y 1, el cual representa a medida que se acerque a 1 cuan relevante es ese documento con respecto a la búsqueda realizada por el usuario.

Entonces, para poder mostrarle al usuario los resultados de su búsqueda, esta definida una clase llamada *SearchItem*, en donde se relacionan los documentos con su respectivo score y una parte del documento donde aparece la query (snippet). Por tanto, en el método *VectorialModel()* hay declarada un array de tipo *SearchItem*, llamada ***queryDocumentsResult*** el cual será devuelto al finalizar la ejecución del método.

La variable ***queryDocumentsResult*** obtiene su valor, es decir, los resultados de la búsqueda, a través de la función local *QueryDocumentsResult()*. En este función se crea una variable temporal ***tempResult*** del tipo *SearchItem[]* para guardar los resultados de la búsqueda.

Luego se obtienen los snippets de los documentos a través del método *GetSnippets()*. En este método se iterara por los documentos para obtenerlos a través del método *SnippetScore()* y guardarlos en una variable de tipo *string[]*, la cual se devolverá al finalizar el método *GetSnippets()*.

Para obtener un snippet de un documento, se calculará primero el score de los posibles snippets que haya dentro del documento y se devolverá el mayor. Para calcular el score de un posible snippet se multiplicará el número de palabras diferentes de la query que aparecen en el snippet por la sumatoria de los valores de TF-IDF de todas las palabras de la query que están en el snippet.

Por lo tanto, dentro del método *SnippetScore()* se disponen de dos diccionarios: ***queryPositions,*** donde se relaciona una lista de posiciones de una palabra de la query con su TF-IDF; y ***queryTFIDFValues*** en el cual el TF-IDF de las palabras de la query se vincula con el número de veces que aparece una palabra en un posible snippet. Las palabras de la query con su TF-IDF se obtienen a través de los campos ***queryTerms*** y ***tfidfMatrix*** de la clase *TextProcessor* y su lista de posiciones dentro de un documento por el método *GetPositions()* de la clase *Terms*. Luego, para calcular los scores de los posibles snippets, se iterará por un array ordenado, que tiene todas las posiciones de las palabras de la query de un documento, donde la distancia que haya entre dos posiciones diferentes deberá ser menor de 200 caracteres.

El *SnippetScore()* para finalizar llamará a *GetSnippetFromDocument()*, un método que dado el texto de un documento y la posición inicial y final del snippet con mayor score, iterara hasta encontrar el principio y el final de la línea donde esta la palabra de la query en la posición inicial y final respectivamente, y devolverá el texto comprendido entre ellos.

Después de haber obtenido los snippets de los documentos, se iterará por los documentos relacionándolos con su respectivo score y snippet en la variable ***tempResult***. Los documentos que tengan score 0 y su snippet sea un *string* vacío no se añadirán al ***tempResult***, puesto que no aportan nada a los resultados de los query. Y al final quedaran en ***tempResult*** solo los documentos relevantes, los cuales serán devueltos al finalizar *QueryDocumentsResult()*.

Por último, antes de finalizar el método *VectorialModel()* y delvolver el valor ***queryDocumentsResult***, los resultados de la búsqueda se ordenaran de manera descendente, utilizando el algoritmo de Selection Sort.

Y así termina la ejecución del método *VectorialModel()* que obtiene los resultados de la búsqueda de la query.

```
public SearchItem[] VectorialModel(string query)
{
    // Se obtienen el vector de la quers, sus palabras normalizadas
    float[] queryVector = terms.GetQueryVector(query, out queryTerms);

    // Se calcula el score a traves de la similitud de coseno y se obtiene el resultado de la busqueda
    float[] cosineSimilarity = CosineSimilarity(tfidfMatrix, queryVector);
    SearchItem[] queryDocumentsResult = QueryDocumentsResult();

    // Se ordena en orden descente el resultado de la busqueda y se devuelve
    Sort(queryDocumentsResult);
    return queryDocumentsResult;

    // Metodo auxiliar que relaciona los documentos con el score y busca el snippet dentro del documento
    SearchItem[] QueryDocumentsResult()
    {
        // Crear un array temporal para los resultados de la busqueda
        SearchItem[] tempResult = new SearchItem[documents.Count];

        // Obtener los snpitted de todos los documentos con respecto al query
        string[] snippets = GetSnippets();

        // Iterar por los documentos
        int indexResult = 0, indexDocument = 0;
        foreach (var title in documents.Keys)
        {
            // Verificar el snippet y el score: no se devolveran los documentos con score 0 y que su spippet sea null
            if (!string.IsNullOrEmpty(snippets[indexDocument]) && cosineSimilarity[indexDocument] != 0)
                tempResult[indexResult++] = new SearchItem(title, snippets[indexDocument], cosineSimilarity[indexDocument]);

            indexDocument++;
        }

        return tempResult[0..indexResult];
    }
}
```