Jocdan Lismar López Mantecón*

*Primer año de Lic. en Ciencias de la Computación

Facultad de Matemáticas y Computación, Universidad de la Habana

Resumen:

El objetivo de este proyecto es crear un motor de búsqueda de documentos de archivos txt basándose en el Modelo del Espacio Vectorial para la Recuperación de Información. Se realizó una revisión bibliográfica previa de la cual se decidieron los siguientes procedimientos: Cada documento se representa como un vector de sus términos en el Corpus de términos, constituido por todas las palabras de la colección de documentos. Como medida de la importancia de las palabras, se recurrió al factor TF-IDF. Así mismo, se trata la query como un pseudo-documento representado por su propio vector de términos del Corpus. Para la determinación de la similitud entre la query introducida por el usuario y los documentos de la colección se utilizó el indicador Similitud de Cosenos que da una medida de la distancia entre el vector query y los vectores-documento a partir de la amplitud del ángulo que forman estos dos vectores en el espacio vectorial de vectores-documento. Además, se generó un snippet para cada resultado de la búsqueda y se implementaron cuatro operadores de búsqueda que facilitan el control sobre la misma.

EL proyecto se desarrolló en el Sistema Operativo Windows 10 Pro y fue codificado en el editor de texto Visual Studio Code empleando C# como lenguaje de programación y el **framework .NET SDK 7***. Se analizaron estructuras de datos factibles para la ejecución del mismo donde resaltan, además de los conocidos array<T>, las colecciones List<T> y HashSet<T> destacándose este último por su elevado rendimiento temporal. También se estudió el tipo System.String y sus métodos para la generación de Snippets y las expresiones lambda.*

Objetivos:

- Crear un motor de búsqueda de documentos en formato .txt empleando el Modelo de Espacios Vectoriales para la Recuperación de Información.
- Generar un snippet para cada resultado de la búsqueda.
- Implementar funcionalidades adicionales que mejoren la calidad de la búsqueda y el control sobre la misma.

Introducción:

Modelo de Espacio Vectorial para la Recuperación de Información (MEV):

El modelo de espacio vectorial es una técnica estándar en la los Sistemas de Recuperación de Información, consiste en representar los documentos como vectores de las palabras que los constituyen. Se construye un vocabulario con todas las palabras de la colección de documentos de forma que no se repitan los términos y cada uno de estos términos representa una dimensión del espacio. Los vectores-documento representaran la importancia de cada término del vocabulario en el propio documento mediante su valor de peso, así mismo la query se trata como un pseudo-documento y se representa mediante un vector en el espacio. Luego, se emplean indicadores de similitud para determinar que documentos son similares a otros o a una query dada.

Term Frequency – Inverse Document Frequency (TF-IDF):

Esta medida de peso de los términos se basa en dos medidas estadísticas:

- Frecuencia de términos (TF): mide la frecuencia relativa con la que un término aparece en un documento, es decir, cuantas veces aparece el término en el documento (frecuencia absoluta N_w) dividido por la cantidad total (T_d) de palabras del documento. La fórmula utilizada es:

  $$\boldsymbol{TF_w = \frac{N_w}{T_d}}$$

  La intuición con esta medida es que los términos que aparecen más veces probablemente sean más importantes para el contenido del documento.
- Frecuencia inversa de documento (IDF): Mide la importancia del término en la colección de documentos, es decir, cuantos documentos de la colección contienen el término.
  Se empleó la fórmula:

  $$\boldsymbol{IDF_w = log_2(\frac{|D|}{|d_w|})}$$

  Dónde |D| es la cardinalidad de la colección de documentos, es decir, la cantidad total de documentos en la colección y $|d_w|$ es la cantidad de documentos de la colección que contienen el término. El logaritmo se utiliza para evitar asignar demasiado peso a los

documentos que aparecen en muchos documentos y para que los términos que aparecen en pocos documentos tengan un mayor peso.

La idea que subyace esta medida es que los términos que aparecen en muchos documentos probablemente sean menos importantes y los que aparecen en menos documentos son más específicos y más relevantes para la búsqueda.

La medida TF-IDF asigna un peso a cada término en cada documento de la colección. El peso de un término se calcula multiplicando la frecuencia del término TF por la frecuencia inversa del documento IDF:

$$\boldsymbol{weight_w = TF_w \times IDF_w}$$

Una vez que se han calculado los pesos para cada término en cada documento de la colección se pueden utilizar para calcular la similitud entre una query y los documentos de la colección. En este caso se representan los documentos como un vector de pesos y se calcula la similitud de cosenos.

Similitud de Cosenos:

Es una técnica utilizada en la Recuperación de Información para comparar la similitud entre dos vectores, se basa en el ángulo entre los mismos. Cuanto menor sea el ángulo, mayor será la similitud entre los vectores. La similitud de cosenos se calcula como el coseno del ángulo entre los vectores mediante la fórmula:

$$\boldsymbol{sim = \frac{A * B}{||A|| * ||B||}}$$

Donde A y B son los vectores que se están comparando, * representa el producto punto de los vectores, y ||A||, ||B|| representan las magnitudes de los mismos.

El producto punto entre dos vectores es la suma de los productos de los componentes correspondientes de los vectores. La magnitud de un vector es la raíz cuadrada de la suma de

los cuadrados de sus componentes. La similitud de cosenos se puede calcular directamente a partir de los pesos de los términos en los vectores.

$$\boldsymbol{A} * \boldsymbol{B} = \sum(\boldsymbol{a_i} * \boldsymbol{b_i}) \qquad |\boldsymbol{A}| = \sqrt{\sum \boldsymbol{a_i^2}}$$

Generando el snippet:

Para generar del snippet se decidió mostrar un subtexto del documento que contenga la palabra de la query de mayor importancia (mayor TF-IDF). b Se extrae inicialmente un snippet con un tamaño de 80 caracteres alrededor de la palabra mas importante de la query con respecto al documento y al corpus; a partir de este primer resultado seguimos tomando caracteres hasta llegar al comienzo y al final de la oración completa respectivamente buscando la primera aparición de los signos de puntuación más comunes que pueden delimitar una oración: " **¿? . ; ¡!** "

Operadores de Búsqueda:

Los operadores de búsqueda añaden opciones y control sobre los resultados de la búsqueda. Se escriben inmediatamente antes de la palabra sobre la cual se quiere aplicar el operador.

Para su implementación se calcula un factor al que se nombró scoreModifier (sM) el cual es un modificador porcentual que multiplicará al valor de Similitud de Cosenos entre el vector documento y el vector query obteniéndose el resultado de Score final. Cada operador contribuye (si aparece en la query) al valor final del scoreModifier.

$$\boldsymbol{sM} = \mathbf{1.00} + \ \boldsymbol{\hat{}mod} + !\boldsymbol{mod} + *\boldsymbol{mod} + \boldsymbol{\sim mod}$$

Se implementaron cuatro operadores de búsqueda:

1. Existe **^** : Las palabras a la que se aplica a este operador debe aparecer en los documentos que se muestran como resultados de la búsqueda. Si el documento a evaluar no contine la palabra-operando (a las cuales se denominó ***markers***) se disminuye su score en un 100%, en caso contrario permanece inalterado:

$$\boldsymbol{\sim mod} = \mathbf{-1.00}$$

2. NoExiste **!** : Los marcadores de este operador no deben aparecer en los resultados de búsqueda. Si el documento contiene dicho marcador disminuye su score en 100% en caso contrario permanece inalterado:

$$!\boldsymbol{mod} = \mathbf{-1.00}$$

3. Importancia *****: Este operador se puede emplear N veces consecutivas al inicio de la palabra. Por cada vez que se utilice se incrementa el score del documento que contenga los marcadores de este operador en un 35%.

$$*\boldsymbol{mod} = \mathbf{+N * 0.35}$$

4. Distancia **~**: Este es el único operador binario, se emplea para indicar dos palabras, la distancia entre estas incrementa el score del documento de forma tal que a menor distancia mayor es el incremento. Para utilizarlo escriba primero la primera palabra y luego la segunda comenzando con el operador (***Ej.: Harry ~vacaciones***).

   Se tuvo en cuenta que el tamaño de los distintos documentos difiere y esto podría sesgar dicha medida de la distancia, por ello luego de calcular la distancia entre los términos en caracteres, se normaliza la misma dividiéndola entre la cantidad total de caracteres del documento. La función que caracteriza este operador es:

$$\sim\boldsymbol{mod} = \mathbf{1.00} \; - \frac{\boldsymbol{dist}(\boldsymbol{w1}, \boldsymbol{w2})}{\boldsymbol{doc.Length}}$$

   Donde $\boldsymbol{dist}(\boldsymbol{w1}, \boldsymbol{w2})$ es la menor distancia entre las dos palabras en el documento y $\boldsymbol{doc.Length}$ es la cantidad total de caracteres del documento.

   De esta forma, si la distancia entre las palabras es pequeña entonces el segundo término de la expresión tiende a cero y el factor tiende a 1.00 (100%); de lo contrario, si la distancia es grande entonces el segundo término tiende a 1 y el factor tiende a 0%.

Sugerencias:

Se implementó la funcionalidad **Sugerencia** basado en la Distancia de Edición o Distancia de Levenshtein. Si alguna palabra de la query no aparece en el vocabulario del corpus y no es una stopword entonces se calcula la distancia de edición entre dicha palabra y todos los términos del vocabulario y sustituimos esta con la palabra del vocabulario con menor distancia de edición.

Luego se sustituye esta palabra en la query.

Distancia de Levenshtein o distancia de Edición

La Distancia de Levenshtein o distancia de Edición es una medida de que tan similares son dos strings. Se define como el mínimo número de transformaciones (insertar, eliminar o remplazar) que se necesitan para convertir uno de los strings en el otro.

Clases Empleadas:

I.) **Corpus**: Se creó la clase **Corpus**, la misma representa la colección de documentos a través del conjunto de términos(palabras) que la componen y contiene al sistema de vectores-documento.

Elementos de la clase:

```
1. string[] documents;
2. Dictionary<string, double> idfs;
3. public Dictionary<string, double> IDFs { get{return vocabulary;}}
4. DocumentVector[] vectorList;
5. HashSet<string> vocabulary;
6. public HashSet<string> Vocabulary { get{ return words;}}
7. public Corpus(string ContentPath){…}
8. public void RankDocumentsBySimilarity(DocumentVector query){…}
9. public void RankDocumentsWithOperators(DocumentVector query){…}
10. public DocumentVector[] Ranking { get;}
11. public HashSet<string> stopWords { get;}
```

1. Contiene un listado de todos los archivos txt que se encuentran en la carpeta Content.
2. Este diccionario le asigna a cada palabra del corpus su valor de IDF.
3. Propiedad que regula el acceso al campo idfs.
4. Arreglo que contiene todos los documentos de la colección representado como vectores.
5. Conjunto que contiene todas las palabras de la colección. Se emplea un HashSet para evitar repeticiones de términos y por su elevado rendimiento temporal en comparación con otras estructuras de datos.
6. Propiedad que regula el acceso a la variable al campo vocabulary.
7. Constructor de la clase Corpus, recibe como argumento la dirección de la carpeta donde se almacenan los archivos txt. Procesa todos los documentos convirtiéndolos en vectores a la vez que conforma el vocabulario, los IDFs y asigna los pesos a cada termino en cada documento.
8. Método que recibe una query y le asigna a cada vector-documento un valor de Score determinado por la Similitud de Cosenos. Este método se emplea cuando la query no tiene operadores de búsqueda.
9. Este método se emplea para asignar el valor de Score a cada documento cuando la query emplea operadores de búsqueda. Para cada documento aplica la función de los operadores de búsqueda correspondientes lo que devuelve un factor que modifica el score (dado por similitud de cosenos) entre el documento y la query.
10. Propiedad que devuelve un arreglo de vectores-documentos organizados en orden decreciente, según su valor de Score, constituido por todos los documentos cuyo score sea mayor que 0.
11. Conjunto de las stopWords. Se decidió como stopword aquellas palabras aparezcan en más del 85% de los documentos.

II.) DocumentVector: Esta clase representa a cada documento como un vector de sus términos a la vez que contiene información útil del documento y métodos necesarios en la concepción del modelo.

Elementos de la clase:

```
1. public string FileName {get; set; }
2. public string FileText {get; private set; }
3. public string[] Terms{ get;}
4. HashSet<string> words;
5. public HashSet<string> Words { get { return this.words; } }
6. Dictionary<string, int> docTermFrequency;
7. double[] weights;
8. public double Score { get; set; }
9. double magnitude;
10. HashSet<int> nonZeroIndexes;
11. public DocumentVector(string docText) {}
12. public void SetWeightsInCorpus(HashSet<string> corpusWords,
    Dictionary<string, double> idfs) {}
13. public void Normalize() {}
14. public static double Similarity(DocumentVector v1, DocumentVector v2){}
15. public static string[] Tokenize(string text) {}
```

1. Contiene el título del documento.
2. Contiene el texto del documento.
3. Vector de todas las palabras del documento (con repeticiones). Se emplea para conocer el tamaño del documento en palabras, la cantidad de repeticiones de cada palabra y en otros puntos del proyecto como el snippet y los operadores.
4. Conjunto de todas las palabras del documento (sin repeticiones)
5. Propiedad que regula el acceso al campo words.
6. Diccionario que a cada palabra del documento le asigna su valor de frecuencia relativa.
7. Representa el vector de coordenadas del documento en el espacio de los términos del corpus donde el peso esta dado por el factor TF-IDF y cada término es una dimensión del espacio.
8. Propiedad que contiene el valor de Score del documento con respecto a la query dada por la Similitud de Cosenos.
9. Norma del vector de pesos.
10. Almacena los índices del vector con valor distinto de 0 para optimizar los cálculos.

11. Constructor de la clase DocumentVector, el texto del documento como parámetro. Inicializa y da valores a otros campos de la clase.
12. Recibe el vocabulario del corpus con los idf de todos los términos, inicializa el array weights que expresa al documento como un vector donde cada coordenada es un término del corpus y cada valor es el peso dado por **tf-idf**.
13. Normaliza el vector dividiendo los valores por la norma.
14. Aplica la fórmula de Similitud de Cosenos entre dos vectores.
15. Método para extraer los tokens del texto. Divide el texto en palabras depurando todos los caracteres que no sean letras o dígitos y devuelve el array de palabras.

III.) Snippet: Clase estática que implementa un grupo de funciones para generar el snippet.

Elementos de la clase:

```
1. static string documentText;
2. static string lowerText;
3. public static string GetSnippet(DocumentVector queryVector,
   DocumentVector docVector){}
4. static string MostRelevantWord(DocumentVector queryVector,
   DocumentVector docVector)
5. static string GetTextPieceAround(int index)
6. public static int GetIndexOf(string word)
7. static int SentenceBeginning(int start)
8. static int SentenceEnding(int start)
9. static string MyTrim(string text)
```

1. Contiene el texto del documento.
2. Aplica el método ToLower() al texto.
3. Método principal de la clase. Recibe el vector de la query y el del documento y genera un snippet que muestra un subtexto del documento que contiene la palabra de mayor relevancia de la query para el texto.
4. Método de que determina la palabra de mayor relevancia de la query. Empleando operaciones entre conjuntos, específicamente la intersección, interseca las

palabras de la query, el documento y el vocabulario (este último para eliminar stopwords) y de las palabras resultantes determina la de mayor peso (TF*IDF).

5. Este método extrae un subtexto alrededor de la primera aparición en el texto de la palabra determinada en el paso anterior, de un tamaño mayor o igual a 80 caracteres.
6. Determina el primer índice donde aparece la palabra más relevante.
7. Determina el inicio de una oración a partir de los signos de puntuación que pueden delimitar la misma.
8. Determina el fin de una oración a partir de los signos de puntuación que pueden delimitar la misma.
9. Método que elimina los espacios en blanco innecesarios en el snippet.

IV.) SearchOperators: Clase estática que define e implementa los operadores de búsqueda.

Elementos de la clase:

```
1.  static HashSet<char> operators = new HashSet<char>("~!^*");
2.  static HashSet<(string word, int count)> importanceMarkers;
3.  static HashSet<string> existenceMarkers;
4.  static HashSet<string> nonExistenceMarkers;
5.  static HashSet<(string, string)> distanceMarkers;
6.  public static bool[] operationsSwitch;
7.  public static bool QueryContainsOperators(string query);
8.  public static void SetMarkers(string query);
9.  private static string TakeWord(string input);
10. public static float ApplySearchOperators(DocumentVector doc);
```

1. Define el conjunto de símbolos que representan los operadores.
2. Contiene las palabras que constituyen operandos del operador Importancia (*).
3. Contiene las palabras que constituyen operandos del operador Existe (^).
4. Contiene las palabras que constituyen operandos del operador No Existe (!).
5. Contiene las palabras que constituyen operandos del operador Distancia (~).
6. Vector booleano que regula cuales operaciones se deben realizar según la query.
7. Método para determinar si la query emplea operadores de búsqueda.

8. Métodos que determina los marcadores (palabras operandos) en la query.
9. Este método separa las palabras de los operadores.
10. Método principal, se encarga de implementar los operadores de búsqueda

V.) Moogle: La clase estática Moogle contiene los métodos que se ejecutan al iniciar el proyecto y desarrollar, es el código que guía la ejecución del programa coordinando al resto de las clases y métodos.

Elementos de la clase:

```
1. public static Corpus corpus;
2. public static void SetCorpus()
3. public static SearchResult Query(string query)
```

1. La variable estática corpus permitirá a otras clases acceder a los elementos del corpus como el vocabulario, los idfs de los términos y el sistema de vectores.
2. Este método se invoca al ejecutar el proyecto. Establece el corpus y lo guarda en la variable estática 1.
3. Se invoca al presionar el botón búsqueda en la aplicación. A través de este método se recibe la query del usuario, esta se transforma en un vector y se establece su similitud con cada documento, se genera el snippet y se forma el resultado final de la búsqueda.

VI.) SearchItem: Es el objeto mediante la cual se representa cada resultado de búsqueda, conformada por el título del documento, el valor de Score y el snippet generado.

Elementos de la clase:

```
1. public string Title { get; private set; }
2. public string Snippet { get; private set; }
3. public double Score { get; private set; }
4. public SearchItem(string title, string snippet, double score)
```

VII.) SearchResult: Este objeto representa el resultado general de la búsqueda, es la colección de resultados individuales (search items). Además dentro de esta clase se implementa la funcionalidad Sugerencia a partir de métodos estáticos.

Elementos de la clase:

```
1. private SearchItem[] items;
```

```
2. public SearchResult(SearchItem[] items, string suggestion="")
3. public string Suggestion { get; private set; }
4. public SearchResult() : this(new SearchItem[0]) {}
5. public IEnumerable<SearchItem> Items() => this.items;
6. public int Count { get { return this.items.Length; } }
7. public static string Suggestion(DocumentVector query)
8. private static string MostSimilarWordInCorpus(string term)
9. private static int EditDistance(string s1, string s2, int m ,int n)
```

Ejecutando el proyecto:

1) Asegúrese de tener instalado .NET Core 7.
2) Ir al directorio del proyecto “Moogle Project”.
3) Copiar la base de datos (colección de documentos en formato .txt) a la carpeta Content
4) Ejecutar en la terminal de Linux:

   ```bash

   make dev

   ```

   Si estás en Windows, desde la carpeta raíz del proyecto:

   dotnet watch run --project MoogleServer

Desarrollo:

Al ejecutarse el proyecto, mediante el paso anterior, automáticamente se llama al método public static void SetCorpus() de la clase Moogle que se encarga de establecer el corpus.

El corpus es una instancia de la clase Corpus, el mismo recibe en su creación la ubicación de la colección de documentos ContentPath, carga todos los archivos de este directorio que contengan la el patrón ".txt"; inicializa el vocabulario de términos, el diccionario idfs, y el array vectorList.

Mediante un ciclo for, para cada documento de la colección:

1. Se lee el texto del mismo mediante el método File.ReadAllText() y a partir de este crea un DocumentVector.
    a. Las instancias de la clase documentVector reciben un texto en su constructor.
    b. Procesa el texto, dividiéndolo en sus tokens(palabras) mediante el método Tokenize()
    c. Inicializa el set de palabras del docuemento words y el diccionario docTermFrequencies.
    d. Calcula la frecuencia de todos los términos del documento.
2. Asigna el nombre del archivo a la propiedad FileName del vector utilizando el método GetFileName().
3. Llena el vocabulario de términos de la colección añadiendo los términos de cada documento. Al emplear un HashSet como estructura de datos para esta tarea se evita la repetición de palabras.

Se calcula el idf de cada palabra eliminando las stopwords. Se consideró stopword aquellas palabras que aparecen en más del 85% de los documentos.

```
4.   // Calculate IDFs and remove stop words
5.           var stopWordThreshold = Math.Log((double)100 / 85);
6.           foreach (var term in this.vocabulary)
7.           {
8.               var idf = Math.Log(documents.Length / idfs[term]);
9.               if (idf > stopWordThreshold)
10.                  idfs[term] = idf;
11.              else
```

```
            {
                idfs.Remove(term);
                vocabulary.Remove(term);
            }
        }
```

Se inicializan y calculan los vectores de pesos de los documentos con el método SetWeightInCorpus(), luego se normalizan los vectores de pesos, de esta forma no tenemos que realizar este paso durante la búsqueda y disminuimos el tiempo de la misma, pues para hallar la similitud solamente restaría realizar el producto escalar.

$$sim = \frac{A*B}{||A||*||B||} \rightarrow sim = A' * B'$$

Donde A' y B' son los vectores normalizados.

Luego se ejecuta la aplicación web donde el usuario introduce la query y presiona el botón ***búsqueda*** que invoca al método Query() de la clase Moogle. Esta toma como argumento la query introducida por el usuario y la transforma en un DocumentVector, calcula el array de pesos y lo normaliza. Luego calcula los scores de cada documento a partir de la similitud de cosenos.

Operadores:

La estructura de control if nos ayuda a determinar si la query contiene o no operadores de búsquedas.

```
    If (SearchOperators.QueryContainsOperators(query))
    {
        SearchOperators.SetMarkers(query);
        corpus.RankDocumentsWithOperators(queryVector);
```

El método booleano QueryContainsOperators(string query) crea un conjunto (HashSet) de los caracteres de la query y lo interseca con la variable estática operators que contiene los caracteres designados para los operadores. Si la interacción es vacía la query no contiene operadores y devuelve false, en caso contrario la query contiene operadores y devuelve true.

Si la query contiene operadores se invoca el método SetMarkers (string query), este divide a la query en palabras, creando un array y analiza mediante el ciclo for el primer carácter da cada una de ellas en busca de operadores.

Se ejecuta la estructura de control Switch, con el default continue, es decir, si el primer carácter no es ninguno de los operadores de búsqueda se pasa a la siguiente palabra de la query:

```
switch (firstChar)
            {
                case '!':
                    nonExistenceMarkers = new HashSet<string>();
                    nonExistenceMarkers.Add(TakeWord(term));
                    operationsSwitch[0] = true;
                    break;
                case '^':
                    existenceMarkers = new HashSet<string>();
                    existenceMarkers.Add(TakeWord(term));
                    operationsSwitch[1] = true;
                    break;
                case '*':
                    importanceMarkers = new HashSet<(string word, int count)>();
                    var word = TakeWord(term);
                    var count = 1;
                    for (var j = 1; j < term.Length; j++)
                    {
                        if (term[j] != '*') break;
                        count++;
                    }
                    (string, int) pair = (word, count);
                    importanceMarkers.Add(pair);
                    operationsSwitch[2] = true;

                    break;
                case '~':
                    distanceMarkers = new HashSet<(string, string)>();
                    var marker1 = TakeWord(queryTerms[i - 1]);
                    var marker2 = TakeWord(term);
                    (string, string) marker = (marker1, marker2);
                    distanceMarkers.Add(marker);
                    operationsSwitch[3] = true;
                    break;

                default:
                    continue;
```

En caso de que primer carácter sea un operador: se inicializa la variable estática correspondiente al operador (ver **Clases Empleadas IV.**) y se la añade la palabra depurada con el método TakeWord(), se "activa" el operationSwitch en la posición correspondiente.

El operationSwitch es un array de bools de cuatro elementos, uno para cada operador, cuando el método SetMarkers encuentra un operador asigna true a la posición correspondiente al operador en el array, esto nos ayudará a ¨recordar¨ que operadores debemos calcular más adelante.

El operador ***** se pude emplear cualquier cantidad de veces consecutivas, por eso en su case empleamos un ciclo for y una variable auxiliar count para contar la cantidad de veces que se empleó.

Al ser binario, el operador de distancia requiere dos operandos. En este case se toma la palabra que lo contine y la palabra anterior, se unen en una tupla y se guardan los pares de palabras en tuplas en la variable estática distanceMarkers.

Luego se procede aplicar el método RankDocumentsWithOperators(queryVector):

Empleando un ciclo for, para cada vector – documento, cuyos scores han sido calculado previamente según su similitud de cosenos con la query, se le aplica un modificador que consiste en multiplicar el score por un factor scoreModifier que se calcula con el método ApplySearchOperators (DocumentVector doc):

Este método "decide" (if) que operadores se deben calcular "preguntándole" al operationSwitch; calcula y devuelve el scoreModifier que es la suma del aporte de cada uno de los operadores aplicados (ver **Introducción**). Para ello invoca los métodos auxilares ApplyXOp (DocumentVector doc) donde ´´X´´ es el operador correspondiente.

```
public static float ApplySearchOperators(DocumentVector doc)
   {
       var scoreModifier = 0.0f;
       if (operationsSwitch[0])
           scoreModifier += ApplyNonExistenceOp(doc);
       if (operationsSwitch[1])
           scoreModifier += ApplyExistenceOp(doc);
```

```
        if (operationsSwitch[2])
            scoreModifier += ApplyImportanceOp(doc);
        if (operationsSwitch[3])
            scoreModifier += ApplyDistanceOp(doc);
        return 1.0f + scoreModifier;
    }
```

Resultados:

La propiedad Ranking del corpus nos devuelve un arreglo, ordenado de forma decreciente, con todos los vectores-documento con un score mayor que 0, todos ellos se mostraran como resultados de la búsqueda, en el orden mencionado, transformándolos antes a SearchItems.

Para la generación del Snippet se llama al método Snippet.GetSnippet(queryVector, docVector), este determina la palabra de mayor importancia(peso) de la query que aparece en el documento,

```
static string MostRelevantWord(DocumentVector queryVector, DocumentVector docVector)
```

La ubica dentro del texto

```
 public static int GetIndexOf(string word)
```

y extrae una ventana del texto alrededor de dicha palabra.

```
static string GetTextPieceAround(int index)
```