Intro

Si estás leyendo este documento probablemente seas un profesor de Matcom y hayas tenido que leerte decenas de reportes de tus alumnos hablando sobre "modelo vectorial", la "famosa fórmula del coseno" y los "tf-idf". La verdad, este es un tema ampliamente difundido, desarrollado, e incluso, el lector debe de haber implementado esta idea en algún momento de su estancia como estudiante en Matcom. Lo importante de este repositorio, es hablar sobre "lo nuevo", "lo genuino", lo que es realmente aportado y concebido por el estudiante. Sinceramente no ha sido para nada fácil llevar a cabo la árdua tarea de implementar este buscador, siempre hay algo que cambiar, que optimizar, y sobre todo, el mayor incordio de todos """"INSTALAR"""". Parece como si estuviera corriendo por una loma interminable, cada vez que crees que te estás acercando al final surge un inconveniente. Pero bueno, esta es la vida que escogimos, al menos, por un período de tiempo, y es una vida llena de esfuerzo, al menos al principio, pero llena de satisfacción, y realización a través de lo creado. Por una cuestión de formalismos en la segunda parte de esta introducción se expondrá una no tan breve descripción sobre los sistemas de recuperación de información y el modelo vectorial. Invito al lector a saltarse esta parte, y comenzar la lectura por el último párrafo de esta intro, ya que en este se explican peculiaridades sobre las estructuras de datos utilizadas en la implementación del proyecto.

Un SRI(Sistema de Recuperación de Información) está constituído por los siguientes elementos:

1. Un conjunto de representaciones lógicas de los documentos de la colección
2. Representaciones lógicas de las necesidades del usuario, las cuales son denominadas consultas
3. Una función de ranking
    - Esta función le asigna a cada documento de la colección un número real, el cual simboliza su grado de similitud con la consulta.
    - Es importante aclarar que la función está determinada por la consulta, su dominio es el conjunto de los documentos, el codominio sería **R**
    - Permite establecer una relación de orden entre los documentos

Entre estos sistemas se encuentra el sistema vectorial el cual tiene las siguientes características:

- Tanto los documentos de la colección, como la consulta, se representan mediante vectores del álgebra lineal n-dimensionales, donde n es el número de palabras del vocabulario considerado(conjunto de palabras que aparecen en, al menos un documento)
- A cada componente está asociada una palabra del universo, y el valor de dicha componente es un número real que representa el peso de la palabra dentro del documento
- Cada palabra será denominada término indexado dentro del documento, y su peso vendría siendo el grado de importancia de dicha palabra o mejor dicho, su capacidad para expresar o resumir el contenido del documento
- Para calcular los pesos de los términos en los documentos y en la consulta se utilizan procedimientos distintos

Sea t_i un término indexado dentro del documento d_j y w_{ij} su peso, entonces dicho documento estará representado por el vector $(w_{1j}, w_{2j}, ..., w_{nj})$ Sea q la consulta y w_{iq} el peso del término t_i dentro de la consulta, entonces la consulta puede representarse mediante el vector $(w_{1q}, w_{2q}, ..., w_{nq})$

Para calcular el grado de similitud entre un documento dado y la consulta, se calcula el coseno del ángulo comprendido entre sus representaciones vectoriales. Esta función está dada por : $sim(d_j, q) = \frac{d_j \cdot q}{\|d_j\| \cdot \|q\|} = \frac{\sum_{i=1}^{n} w_{ij} \cdot w_{iq}}{\sqrt{\sum_{i=1}^{n} w_{ij}^2} \cdot \sqrt{\sum_{i=1}^{n} w_{iq}^2}}$

Veamos ahora el procedimiento para calcular los pesos: Sea $frec_{ij}$ la frecuencia del término t_i en el documento d_j, denominaremos frecuencia normalizada y la denotaremos por tf_{ij} al número $\frac{tf_{ij}}{\max(tf_{lj})}$ donde $l = 1, 2, 3, ..., n$

Por otro lado existe otro valor importante en el cálculo del peso de cada término, el cual depende de la frecuencia del término en cada uno de los documentos de la colección y se denomina *inverse document frecuency*. Se calcula mediante la siguiente fórmula $idf_i = \log(\frac{N}{n_i})$ donde N es el total de documentos, y n_i el número de documentos en los que aparece el término t_i .

Luego, el peso de cada término dentro del documento se calcula mediante la fórmula $w_{ij} = tf_{ij} \cdot idf_i$

Para calcular el peso de cada término dentro de la consulta se utiliza el algoritmo $w_{iq} = (a + (1 - a)(\frac{freq_{iq}}{\max(freq_{lq})})) \cdot idf_i$

Los valores más utilizados para la constante a son 0.4 y 0.5

A la hora de transformar el documento en un vector, solamente nos interesa asignar valores a los términos que aparecen en dicho documento, puesto que lo que no aparecen tendrán valor 0, y la hora de efectuar el producto con el vector de la query los términos con peso nulo no influyen en el resultado, lo mismo ocurre al calcular las normas. Además, guardar los ceros, supone un gasto de memoria mayor, y a la hora de calcular la suma del producto entre las componentes de los dos vectores se realizan operaciones innecesarias. En este proyecto se optó por utilizar diccionarios para la representación vectorial de los documentos, de esta manera, además de almacenar solo los términos que aparecen dentro del documento se hace referencia a las palabras de forma explícita, es decir mediante llaves con el mismo nombre, y no como en el caso de los arrays, donde la palabra "ciclanita" está asociada a un índice.

Desarrollo

En esta parte del repositorio se realizará una descripción sobre las clases que se crearon para la implementación del proyecto.

Clase Document

Esta clase es una abstracción del concepto documento. Cuenta con cuatro variables de instancia :

- string core= Almacena el texto del documento

- string title= Hace referencia al titulo del documento
- Dictionary<string, int> tfs= Contiene cada una de las palabras del documento con su frecuencia dentro del mismo
- Dictionary< string, double> weigth= Contiene el tf-idf de cada palabra, es decir el peso

No solo se crearán instancias de esta clase basándonos en los txt de nuestra base de datos, sino que el texto de la búsqueda será considerado como un documento, con todas sus variables de instancia. Esta clase contará con dos constructores, uno para los documentos de la base de datos, y otro para instanciar el query.

A continuación se enumerarán los métodos de esta clase y se explicará brevemente su función:

1. Constructor convencional:
   Se aplica solamente a los documentos de la base de datos. Se encarga de separar las palabras del texto y calcular su frecuencia. Toda esta información se guarda en la variable de instancia **tfs** Además inicializa las variables textbfcore y textbftitle

   Aspectos a destacar:

   - Este constructor solamente tiene en cuenta como criterio para separar dos palabras que exista un espacio entre ellas
   - A la hora de construir las palabras solo considera los caracteres que representen consonantes o vocales válidas en ciertos idiomas.
   - Remueve las tildes en las palabras donde aparezcan
   - En el momento en que se llama a este constructor no se cuenta con la suficiente información para calcular el tf-idf de las palabras, por tanto este constructor no inicializa la variable weigth, o mejor dicho le asigna un diccionario vacío.

2. Constructor del query
   A diferencia del constructor de los documentos de la base de datos, este método se aplica después de haber calculado el idf de las palabras del vocabulario, por tanto cuenta con la suficiente información para calcular los pesos de las palabras del query. Es decir, inicializa todas las variables de instancia.

   A la hora de indexar las palabras del query hay que tener en cuenta ciertos caracteres especiales, entre los que está el operador de cercanía ~ y el operador de relevancia ** . Si al query le aplicáramos el constructor de los documentos convencionales se obtendrían resultados erróneos.
   Por ejemplo:
   “computacion~**arquitectura”**

   En este caso,luego de aplicar el constructor convencional la variable tfs quedaría con el único par:

computacionarquitectura	1

   Puesto que el constructor no reconoce al carácter ~, lo ignora y como no ve un espacio considera que todo el texto de búsqueda tiene una sola palabra. Por otra parte si el texto fuera **“ computacion *****arquitectura “** entonces a la hora de calcular los pesos de las palabras no tomaría en cuenta los caracteres*****.

Otra cuestión que tiene en cuenta el constructor del query es el caso en que el usuario haya escrito una palabra incorrectamente, y como resultado exista un término de la búsqueda que no pertenezca el vocabulario, es decir al universo de palabras.
Una vez el constructor termina la indexación, recorre el diccionario que contiene a todas las palabras del query con la frecuencia. Si encuentra alguna que no pertenezca al vocabulario, entonces le aplica el método Sugestion de la clase SugestionUpdate. Este método se auxilia del diccionario global que almacena todas las palabras del universo junto con su idf, y retorna un string que hace referencia a lo que el usuario pudo haber querido decir, pero escribió mal.

Este constructor se encarga de calcular el peso real de la palabra que el usuario quiso decir. Por ejemplo:

a) **“ quer query operations qery ”**
Al aplicar el constructor se obtiene un diccionario con los pares:

query	3
operations	1

“ quer quer operations “
El diccionario tendrá los pares:

query	2
operations	1

Obsérvese que en la primera búsqueda el usuario escribió query incorrectamente de dos formas diferentes y después lo escribió bien, lo que sugiere un error de tecleo, mientras que la segunda parece un error ortográfico porque query fue escrita mal dos veces de la misma manera. Para diferenciar estos dos casos hubo que realizar cambios en la implementación.

3. GetIdf:

   - Recibe un array de instancias de la clase Document, y un diccionario de tipo <string, double>
   - Se encarga de calcular los idf de todas las palabras y guardar esta informacón en el diccionario que recibe
   - Una vez obtenidos los idf, calcula el peso de las palabras dentro de cada documento en particular, es decir se encarga de inicializar la variable weigth en cada objeto de tipo Document del array.

4. Similarity
   Compara dos documentos y retorna un double que representa el grado de similitud entre ellos

5. Sorting

   - Recibe un array de objetos Document, y un string que representará la query(el texto de búsqueda)
   - A cada documento le asigna un score con respecto al query, el conjunto de los scores es almacenado en un array double, el cual es retornado
   - Ordena el array de documentos según el score de cada uno.

Esta clase contiene métodos auxiliares como separadores de palabras especializados que se utilizan en los constructores, y un método para calcular la norma de un vector el cual es utilizado en el método Similarity

Clase Moogle:

Esta clase cuenta con dos variables estáticas. Una es un array para guardar todos los documentos indexados de nuestra base de datos, y la otra es un diccionario<string, double> para guardar todas las palabras que se encuentran en, al menos, un documento (vocabulario) junto con su idf

Esta clase contiene dos métodos fundamentales

1. GetDocuments :

   - Carga todos los archivos txt de la base de datos desde el directorio en que se encuentran y los convierte en string.
   - Crea instancias de la clase Document con estos strings y las guarda en la variable estática doc. En un principio al aplicar el constructor solo se inicializan las variables core, title, y la que hace referencia a los tfs.
   - Luego, se llama al método GetIdf, pasándole como parámetro la variable global con los idf

2. Método Query:

   - Llama a GetDocuments el cual realiza primero el preprocesamiento, en caso de que no se haya efectuado con anterioridad y las variables estáticas esten vacías.
   - Llama al método Sorting de la clase Document, el cual se encarga de ordenar el array de documentos según el grado de similitud que tengan con la búsqueda.
   - Luego, se realiza un nuevo procesamiento del texto del query, buscando palabras con los operadores `~` y `^` . El método que se encarga de dicho procesamiento es Operations de la clase del mismo nombre. Para obtener informaciíon sobre su funcionalidad revisar la clase Operations.
   - La búsqueda no arrojará más de 7 resultados.
   - Cada documento que se considere como resultado de la búsqueda deberá contener a todas las palabras que almacena la lista retornada por el método Operations.
   - Habíamos visto que el m'etodo Sorting ordenaba los documentos seg'un el score, sin embargo este score puso haber cambiado con la aplicación del métod Operations por tanto se ordenan nuevamente los documentos de mayor a menor según el score y se instancian los 7 primeros que cumplan con el requisito mencionado anteriormente.
   - Para instanciar los resultados de la b'usqueda se cuenta con dos clases SearchItem y SearchResult

textbf Clases auxiliares para instanciar los resutados de la búsqueda

1. SearchItem
   Es una representación resumida de un objeto de tipo Document, que nos aporta información sobre el título del documento, un fragmento del documento (Snippet) relacionado con la query y el score del documento con respecto a la query.

2. SearchResult
   Contiene los documentos más relevantes, y una sugerencia para la búsqueda.

Clase SugestionUpdate

Esta clase ha sufrido varias transformaciones en pos de experimentar con posibles alternativas a la aplicación del algoritmo de *Levenshtein*, de ahí el sufijo Update. Métodos:

1. Sugestion

   - Se encarga de gestionar una sugerencia en caso de que el usuario haya escrito una palabra que no se encuentre en el vocabulario.
   - Recorre todo el vocabulario, es decir, el diccionario global con los idfs de las palabras y compara cada una de ellas con el string que recibe como parámetro utilizando el algoritmo de Levenshtein.
   - Solamente se admite una distancia de edición <= 3. Por tanto únicamnete se comparan las palabras tales que el módulo de la diferencia entre su longitud y la del string que simboliza la búsqueda sea <=3. Esto es un manera de descartar palabras del vocabulario sin tener que efectuar la comparación.
   - Fue concebida otra versión de este método que antes de recorrer el diccionario global buscando coincidencias, primero intenta fraccionar la secuencia pasada como parámetro en un conjunto de palabras válidas. En caso de que dicha cadena no pudiera ser fragmentada entonces se aplicaría levenshtein. Se optó por no utilizar esta versión del método Sugestion, pero su implentación se encuentra dentro del código fuente. Y el método en que el que se auxilia para la fragmentación ha sido probado en consola con diccionarios mas pequenos.

2. Levenshtein:
   Calcula el número mínimo de operaciones con solo caracter(sustituciones, eliminaciones o agregos) que hay que realizar para igualar dos cadenas dadas.

3. Split:
   Fragmenta el string que recibe como paámetro en un conjunto de palabras válidas del vocabulario

Clase Operations

Para implementar la búsqueda de un Snippet primero es necesario definir una noción intuitiva de lo que podría ser un buen Snippet. Se llegó a la conclusión de que un buen Snippet podría ser un fragmento del documento que se asemeje a la búsqueda. Luego el mejor Snippet podría considerarse como el fragmento que más se parezca. Para implementar esta idea se le asignó a cada snippet generado un valor que refleje el grado de similitud con la búsqueda. En este valor influyen dos factores: la cantidad de palabras de la query que se encuentren en el fragmento de texto y el peso de estas palabras dentro de la query.

Dejemos esta idea flotando en el aire por un momento y hablemos un poco del operador de cercanía ~. Si este operador se encuentra entre dos palabras válidas de la query entonces el score de los documentos cambia según el grado de proximidad de esas palabras en cada uno de los documentos. El método encargado de cambiar el score de los documentos se denomina ChangeScore. Antes de hablar sobre la implementación de este método hay que tener en cuenta

cómo se modifica el valor de los scores.

Primeramente se itera por el texto de cada uno de los documentos y se calcula la distancia mínima a la que se encuentran las dos palabras. Estos resultados se guardan en un array. Tiene sentido luego obtener la mayor de estas distancias mínimas y dividir este número entre todas las distancias mínimas para determinar el valor de variación del score.

Por ejemplo, supongamos que tenemos 4 documentos y la operación computacion~permutacion

Documento	Distancia Mínima
1	30
2	20
3	no se encuentra la cadena computacion
4	5

Documento	Valor de variación
1	1
2	0.666666...
3	no sabemos como definirlo por el momento
4	6

Si multiplicamos cada score por el valor de variación anteriormente considerado se obtendrían variaciones descontroladas de cada score. Es mejor limitar el rango de estas variaciones utilizando el logaritmo en base 10. Sin embargo, cuando calculamos el logaritmo a estos números se podrían obtener valores negativos o valores positivos menores que la unidad. Para evitar esto antes de aplicar el logaritmo a cada valor se le suma 10. De esta manera solo se obtendrán valores positivos mayores que 1.
Mientras mas cercanas están las palabras mas grande será el resultado de la división considerada anteriormente. Luego el valor de variación aumentará más. Si las dos palabras se encuentran en el documento A entonces al multiplicar el score por el valor de variación siempre este aumentará. Ya tenemos nuestra respuesta para la interrogante de qué hacer cuando alguna de las palabras no se encuentra dentro del documento. Simplemente el valor de variación será 1, es decir el score no cambiará.

Luego, valores de variación serían los siguientes

Documento	Distancia Mínima	División entre la mayor de las distancias mínimas	Sumando 10 y aplicando logaritmo
1	30	`30/30==1`	log(11) == 1.04139...
2	20	`30/20==1.5...`	log(11.5) == 1.0606...
3	-1		1
4	5	`30/5==6`	log(16) == 1.204...

Un primer enfoque a la hora de calcular la distancia mínima entre dos palabras en un documento dado, implica recorrer el texto del documento. Sin embargo a la hora de determinar el mejor snippet también es necesario recorrer todo el texto nuevamente buscando el fragmento con mayor valor con respecto al query. Para esto iteramos por el texto, una vez encontremos una palabra que pertenezca al query, entonces generamos a partir de ella 10 palabras. El snippet actual será el conjunto de todas estas palabras. El valor del snippet será la suma de los pesos de las palabras de la query contenidas en él.

Hay una manera de implementar el problema del snippet y el problema de la distancia mínima sin necesidad de recorrer la cadena dos veces. Podemos crear un diccionario que tenga como llaves a las palabras de la query y a cada llave le asigne una lista de enteros ordenados, que simbolicen las ocurrencias de esa palabra dentro del documento.

Ej:
Frodo Bolson vivia en la comarca, pero despues Frodo tuvo la necesidad de irse de la comarca. En principio todo era pacifico, pero llego una era oscura, a Frodo no le quedo mas remedio que emprender el camino hacia la colina, dejando asi su querida comarca.

Supongamos que la query fuera **" Frodo~comarca"** Nuestro diccionario de ocurrencias tendría los pares

Key	Value(Lista de ocurrencias)
Frodo	0 8 28
comarca	5 16 45

Para calcular la menor distancia entre Frodo y comarca podemos iterar por la lista asociada a Frodo. Una vez fijado un índice i, es muy fácil recorrer la lista de comarca para buscar la ocurrencia de comarca mas cercana al valor del índice fijado de Frodo, para hallar la distancia entre los valores de los índices considerados se calcula el módulo de la diferencia. Nos quedamos con la menor de todas estas distancias.

Un proceso similar se podría realizar para hallar el snippet. Al fijar una posición(ocurrencia) de una palabra de la query dentro del documento podemos iterar por las listas de ocurrencias de todas las palabras de la query. De esta manera podremos saber las palabras de la query que se encuentran a un rango menor de 10 palabras con respecto a la posición fijada y si esto ocurre agregamos el peso de estas palabras al valor del snippet.

Supongamos que nuestra query es **"Frodo comarca pero"** entonces nuestro diccionario de ocurrencias tendrá los pares

Key	Value(Lista de ocurrencias)	Peso
Frodo	0 8 28	a_1
comarca	5 16 45	a_2
pero	6 22	a_3

Supongamos que estamos analizando el valor del Snippet que comienza con la palabra Frodo en la posición 0 (se considera como ocurrencia el número de la palabra y no el índice de su primer caracter). Según la lista de Frodo hay otra ocurrencia de Frodo y es la palabra no. 8 del texto, por tanto está a una distancia de 8 palabras, luego al valor del snippet actual le sumamos el duplo del peso de Frodo. Sin embargo, iterando por la lista de ocurrencia de "comarca" tenemos que hay una ocurrencia que se encuentra a un radio de 5 palabras por tanto incluimos el peso de "comarca" al valor del snippet, y luego realizamos el mismo procedimento con la lista de ocurrencias de "pero". Al final obtenemos que el valor del snippet será $2a_1 + a_2 + a_3$

Métodos más importantes de la clase Operations:

1. Positions

- Recibe un string con el texto de documento y un diccionario que representa la query en su forma indexada
- Retorna un diccionario con las ocurrencias de las palabras de la query en el documento

2. ChangeScore

   - Recibe como parámetros dos strings que son las palabras de la query entre las que se encuentra el operador ~ , un array de diccionarios con las ocurrencias de las palabras de la query en cada uno de los documentos y un array con los scores de los documentos.
   - Cambia los scores de los documentos según el grado de proximidad de las palabras.

3. Operations

   - Procesa nuevamente el texto de la query y guarda en una lista las palabras de la query precedidas por el caracter ^, y si encuentra una operación de tipo <vocablo> <vocablo> llama al método ChangeScores para actualizar los scores.
   - Retorna la lista

4. Distance

   - Recibe como parámetros dos palabras del query y un diccionario con las ocurrencias de estas palabras en el query
   - Retorna la menor distancia entre estas palabras si aparecen en el documento

5. BestStart

   - Recibe como parámentros dos diccionarios, uno con las ocurrencias de las palabras del query, en el documento, y otro con sus pesos
   - Retorna el no. de palabra a partir del cual comienza el mejor Snippet

   Esta clase cuenta con otros métodos auxiliares como generadores de palabras a partir de un índice determinado siguiendo varios criterios de generacíon y el m/'etodo auxiliar ObtainValue que calcula el valor del Snippet formado a partir de una posición determinada que recibe como parámetro

Clase SnippetOperations

Los métodos más importantes de esta clase son :

1. MakeSnippet Su función es generar el mejor Snippet dentro de un documento dado. Para realizar sus funciones se auxilia de los m'etodos de la clase Operations.

2. MakeWords

   - Recibe como parámetros un string que representa un texto de un documento dado, un int que es un índice dentro del string, y otro int n
   - Genera una frase de n palabras a partir del índice indicado dentro del texto que recibe.

Esta clase también cuenta con un generador de palabras.