Reporte de Proyecto de Programación 1er Año (Moogle!)

Leonardo Amaro Rodríguez C111

# Objetivo del Proyecto

El objetivo final del proyecto fue crear un programa, que dado un texto de query, buscara en un conjunto de documentos (ubicados en la carpeta Content) y los ordenara por relevancia, dejando de lado aquellos que no se consideraran relevantes para la búsqueda en cuestión.

# Desarrollo del Reporte

Este reporte se organiza en las siguientes secciones:

1. [Indexado de los Documentos.](#_toc46)
   1. [Lectura de los documentos y filtrado de términos.](#_toc48)
   2. [Modelo de Espacio Vectorial.](#_toc51)
   3. [Ponderación de términos.](#_toc55)
      1. TF-IDF.
2. [Búsqueda.](#_toc67)
   1. [Filtrado de términos del Query.](#_toc70)
   2. [Búsqueda estricta.](#_toc72)
      1. Ponderación del Query.
      2. Similitud de Coseno.
      3. Operadores de Búsqueda.
         1. Implementación de los operadores.
   3. [Búsqueda de variaciones.](#_toc97)
      1. Distancia de Damerau-Levenshtein.
   4. [Snippet.](#_toc103)
   5. [Sugerencia de Búsqueda.](#_toc105)
3. [Cambios Extra al Proyecto.](#_toc107)
   1. [Cambios de HTML y CSS.](#_toc109)
   2. [Uso de métodos asíncronos.](#_toc114)

## Indexado de los Documentos

Antes de poder realizar cualquier búsqueda es necesario generar y almacenar todos los datos que el motor necesitara para realizar búsquedas de manera rápida y eficiente

### Lectura de Documentos y Filtrado de Términos

El objetivo principal en un inicio es conformar el vocabulario, que contendrá todos los términos distintos del corpus, ademas de que se necesita saber cuantas veces ocurre cada termino en cada documento ([más sobre esto en la sección de Modelo de Espacio Vectorial](#_toc51)).

Para lograr esto es necesario un método recorra todas las palabras en un documento, y nos permita ejecutar alguna acción sobre cada una de ellas, para este mismo propósito fue creado el método estático de la clase estática Utils, Utils.ForEachWordInFile, que nos permite ejecutar una acción sobre cada palabra a través de un delegado Action<string>. Dicho método no solo nos permite ejecutar una acción sobre cada palabra, sino que ademas filtra los caracteres innecesarios, como signos de puntuación y caracteres especiales, al igual que convierte determinados caracteres a una forma más general (por ejemplo eliminando las tildes y otros caracteres similares). La acción que se ejecuta añade cada palabra que se encuentra al vocabulario, y registra la palabra (o más bien su indice en el vocabulario) con la cantidad de veces que aparece en el documento que se esta leyendo a través de un diccionario de tipo <int, int>, si la palabra ya esta registrada en el diccionario (hay un diccionario separado para cada documento), simplemente se incrementa la cuenta de veces que aparece.

### Modelo de Espacio Vectorial

El Modelo de Espacio Vectorial es un modelo algebraico utilizado para representar documentos (o otro cualquier objeto) como vectores de valores numéricos. El objetivo de almacenar documentos de esta forma es que permite realizar búsquedas en docenas de documentos en fracciones de segundos, ya que en vez de recorrer todos los documentos (que pueden llegar a ser bastante extensos), realiza operaciones algebraicas que, ademas de dar resultados mucho mas precisos que búsquedas crudas, provee un nivel de rendimiento muchísimo mayor.

En nuestra implementación organizamos todos los vectores de documentos en una matriz de dimensión mxn, donde m es el numero de documentos, y n el numero de términos en el vocabulario, de forma tal que todos los vectores documentos tienen la misma longitud (esto sera particularmente útil mas adelante), donde si un termino j no aparece en el documento i, el valor Ai,j en la matriz seria 0.

(Más sobre esto en la sección de TF-IDF)

### **Ponderación de Términos**

Pero para poder colocar valores en la matriz, se debe definir un sistema de ponderación, pues no todos los términos pueden tener el mismo valor, ya que esto haría la búsqueda completamente irrelevante, debe de definirse una forma en la que a los términos se les pueda asignar un valor de *relevancia* con respecto al documento al que pertenecen, e incluso con respecto al resto de los documentos.

# TF-IDF

El esquema de ponderado TF-IDF (Term Frequency – Inverse Document Frequency) es ideal para el propósito del buscador, pues le asigna a cada termino un valor que indica cuan relevante es, no solo en su propio documento con respecto al resto de los términos, sino que también con respecto al corpus en general.

El TF-IDF se divide en dos partes: TF, que indica cuan importante es el termino en un documento, y el IDF, que indica cuan poco común es el termino en el corpus, lo cual le da mayor relevancia. Ambas partes son calculadas y multiplicadas para dar lugar al peso del termino en dicho documento.

El TF se puede calcular de diversas maneras, al igual que se pueden tomar muchas interpretaciones en dependencia del problema en cuestión. La implementación de este proyecto plantea el calculo del TF de la siguiente manera:

Esta formula divide la frecuencia cruda (cuantas veces se repite el termino en el documento) por la máxima frecuencia cruda en el documento. La ventaja principal de escoger esta implementación es que evita una inclinación hacia documentos mas extensos, pues con otras implementaciones estos tendrían un TF mayor en sus términos, mientras que documentos mas cortos podrían tener mayor relevancia para la búsqueda.

El IDF es calculado de la siguiente manera en este proyecto:

Esta formula halla el logaritmo de la división de la cantidad de documentos por la cantidad de documentos en los que aparece el termino, de forma tal que si un termino aparece en todos los documentos, su IDF es 0, ya que es irrelevante para la búsqueda.

Una vez definido el TF-IDF el algoritmo que permite traducir los diccionarios que contienen las frecuencias crudas de cada termino en cada documento, a una matriz de frecuencias crudas, y de ahí a una matriz ponderada con TF-IDF es trivial.

## Búsqueda

La búsqueda constituye el órgano fundamental del proyecto, y funciona tal que dado un texto (query) encuentra y ordena documentos en orden de relevancia para dicho query en especifico.

Como ya se menciono la búsqueda se realiza con operaciones algebraicas entre vectores (mas sobre esto en la sección de Similitud de Coseno), pero antes de eso se debe preparar el query y transformarlo en un vector, así como hallar el TF-IDF de cada uno de los términos en dicho vector.

### Filtrado de términos del Query

Al igual que los términos de los documentos, los términos del query deben ser filtrados de una manera similar, pues deben coincidir con aquellos en los documentos. Esto se realiza utilizando el método estático Utils.SeparateTextInString que al igual que Utils.ForEachWordInFile, nos permite (con una implementación mas general) separar y filtrar las palabras que de un texto cualquiera. En el caso del query se utilizo un filtro especial ademas del mismo filtro que se utilizo al filtrar los términos en los documentos; este nuevo filtro permite que determinados caracteres no sean eliminados (mas sobre esto en la sección de Operadores de Búsqueda). Cambien deben saltarse (pero no eliminar por completo, ya que serán usados mas tarde) términos que no se encuentren en el vocabulario)

### Búsqueda estricta

Se creo el método StrictSearch en la clase Moogle, cuyo objetivo es realizar una búsqueda relativamente simple de un termino sobre todos los documentos, se le llama “simple” debido al hecho de que asume que todos los términos pertenecen al vocabulario.

# Ponderación del Query

Para calcular cuan relevante es un documento para un query en especifico, se realizan operaciones de álgebra vectorial que requieren que los vectores tengan la misma dimensión, así como significados equivalentes, y ya que nuestros vectores documento contienen el TF-IDF de los términos del vocabulario, nuestro query debe ser convertido en un vector que contiene el TF-IDF de todos los términos de mismo vocabulario, solo que con respecto al query en vez de un documento, es decir, se toma el Query como un documento especial. El procedimiento para ponderar dicho vector es idéntico al del resto de los documentos.

# Similitud de coseno

La similitud de coseno es una medida de similitud entre dos secuencias de números. Para definirlo, las secuencias pueden ser vistas como vectores de un mismo espacio, y la similitud de coseno se define como el coseno del angulo entre dichos vectores. Por ejemplo, dos vectores proporcionales tienen una similitud de coseno de 1, mientras que dos vectores ortogonales tienen una similitud de coseno de 0. Mientras mayor sea la similitud de coseno, mas similares son dos vectores, y, en esta implementación, mayor es la relevancia del documento con respecto al Query. La similitud de coseno se calcula de la siguiente manera:

Que es la división del producto escalar entre los dos vectores sobre la multiplicación de las normas de dichos vectores. Entonces al tener los vectores de los documentos y el vector del Query, se puede calcular la similitud de coseno entre ellos y así determinar que documentos son mas relevantes para la búsqueda.

# Operadores de Búsqueda

El motor de búsqueda también cuenta con la facilidad de definir los siguientes operadores:

1. ^ : este operador al inicio de un termino en el Query indica que dicho termino **debe** aparecer en todo documento que sea devuelto por la búsqueda.
2. ! : este es completamente lo contrario al anterior, se coloca al inicio de un termino en el Query e indica que dicho termino **NO debe** aparecer en ningún documento que sea devuelto por la búsqueda.
3. \* : este operador se coloca al inicio de un termino en el Query, e indica que dicho termino debe tener mayor relevancia en la búsqueda que el resto de los términos.
4. ~ : este operador se coloca entre dos o mas términos en el query, e indica que mientras mas cerca se encuentren en el documento los términos unidos por dicho operador, mayor debe ser la puntuación del documento.

# *Implementación de los operadores*

Operador ^:

Este operador se implemento en la búsqueda de manera simple, en el momento de ordenar y filtrar resultados irrelevantes de la búsqueda, si un documento no contenía el termino seria marcado de irrelevante, y no seria devuelto.

Operador !:

Implementado de forma similar al anterior, si un documento contenía el termino al que se le aplico el operador, este documento seria marcado de irrelevante para la búsqueda.

Operador \*:

Antes de realizar la búsqueda estricta, mientras se filtraban términos que no pertenecían al vocabulario, se aumenta artificialmente la frecuencia cruda de los términos marcados por uno o mas de este operador, haciéndolos así mas relevantes en el query.

Operador ~:

Antes de realizar la búsqueda estricta se guardaron los términos a los que se les aplicaba este operador, luego, mientras se ordenan los resultados de la búsqueda, la puntuación de cada documento se aumentaba en razón de cuan cerca estaban los términos seleccionados y para obtener la distancia mínima entre estos términos dentro de cualquier documento se realiza el siguiente procedimiento:

1. Se obtienen todas las posiciones en las que ocurren estos términos dentro del documento.
2. Entonces un método recursivo buscaría el camino mas corto entre todas las posiciones, la longitud del cual seria la mínima distancia entre los términos

### Búsqueda de variaciones

En el caso de que la búsqueda inicial no diera suficientes resultados (el numero de resultados considerado mínimo esta definido como una constante) el buscador debería buscar mas aya, buscando variaciones de las palabras del Query. Para esto seria necesario un algoritmo que compute cuan similar es una palabra de otra, y así se puede calcular que palabras del vocabulario se pueden considerar como variaciones de otra.

# Distancia de Damerau-Levenshtein

Precisamente el algoritmo que necesitaríamos para hallar la diferencia entre dos palabras. En su definición recursiva el algoritmo de Damerau-Levenshtein encuentra las inserciones, eliminaciones, sustituciones y transposiciones que se le necesitarían realizar a una palabra para convertirla en otra, efectivamente hallando la diferencia entre ellas.

Utilizando una distancia limitada se pueden hallar todas las variaciones de cada termino de manera tal que no sean demasiado distintos del original.

Luego con un simple método recursivo se pueden determinar todas las combinaciones de todas las variaciones de los términos del query. Una vez se tienen todas las posibles variaciones de query una opción seria realizar búsquedas con todas y luego ordenar los resultados por relevancia, pero en casos extremos el numero de variaciones total del query puede llegar a las decenas de miles; por lo tanto una mejor opción es ordenar las variaciones por orden de potencial relevancia, y realizar búsquedas hasta que se llegue al mínimo de búsquedas necesarias.

### Snippet

El snippet es el texto que va debajo de los títulos de los documentos encontrados al realizar una búsqueda, debe dar una previsualización del documento y del contexto en el que se encuentran los términos buscados en el query. Para esto hay múltiples implementaciones, pero la que se escogió para este proyecto fue: dividir el documento en párrafos, tratar los párrafos como documentos especiales y hallar el vector ponderado de TF-IDF de cada párrafo, luego hallar la similitud de coseno entre el query y cada vector, y ordenarlos por relevancia, luego se añaden estos párrafos al snippet en orden, hasta que un limite de caracteres es alcanzado en el snippet. (definido en una constante).

### Sugerencia de Búsqueda

Luego de realizar cualquier búsqueda exitosa (ya sea la original o las variaciones), se guarda en una lista una tupla que contiene el query de dicha búsqueda, y la similitud de coseno del elemento de mayor relevancia. Una vez se han realizado todas las búsquedas, si alguna de ellas tuvo mayor similitud de coseno máxima que la búsqueda original, se enviá como sugerencia de búsqueda al usuario.

## Cambios Extra al Proyecto

El proyecto inicial, a pesar de que funciona perfectamente, carece de determinados rasgos de calidad de vida que se dejo que los estudiantes implementaran opcionalmente.

### Cambios de HTML y CSS

1. Se cambio el botón de búsqueda a un elemento <input> de tipo submit y tanto el botón como el campo de texto de búsqueda se colocaron dentro de un elemento <form> cuya acción submit era el método de búsqueda. Esto se hizo con el objetivo de que el usuario pudiese buscar presionando Enter sobre el campo de búsqueda, sin necesidad de presionar el botón.
2. Se anadino un método que al hacer clic en el texto de sugerencia, realizaría la búsqueda sugerida.
3. Se introdujeron textos de información, uno para el indexado y otro para la búsqueda, así como animaciones para estos, lo que hace que sea menos *desesperante* la tensión mientras el método busca, así como que es una señal de que todo esta funcionando como deberia.
4. Cambien se anadino un link a los nombres de los archivos cuando se realiza la búsqueda, de forma tal que si se hace clic en el nombre del archivo, se abre en la aplicación de preferencia del usuario (Nota: esto no funciona en Linux y macOS por limitaciones la API).

### Uso de métodos asíncronos

Al comenzar a trabajar con elementos de la interfaz de la pagina, se hizo notar que esta se bloquea al realizar una búsqueda, pues es una operación bastante pesada, pero esto no debería ser así en una aplicación lista para producción, así que tanto el método de indexado como el de búsqueda fueron marcados como async, y varias de sus tareas mas costosas se realizan de forma asíncrona, por lo cual ya no bloquean la interfaz al ejecutarse.

# Referencias

* [Modelo Espacio Vectorial, Wikipedia](https://en.wikipedia.org/wiki/Vector_space_model)
* [TF-IDF, Wikipedia](https://en.wikipedia.org/wiki/Tf–idf)
* [Producto Escalar, Wikipedia](https://en.wikipedia.org/wiki/Dot_product)
* [Norma de un vector, Wikipedia](https://en.wikipedia.org/wiki/Norm_(mathematics))