Universidad de La Habana

"Facultad de Matemática y Computación"

Moogle

Proyecto de Programación

Autor:

Rubén Martínez Rojas

Abstract

Moogle es una página web creada con el fin de buscar un texto en un grupo de archivos. Para su funcionamiento utiliza un Sistema de Recuperación de Información desarrollado en lenguaje *C#*

1 Introducción

Para la creación del Proyecto se implementaron diferentes clases en MoogleEngine:

- **Inicio:** La clase donde se inicializan los datos, y se completan con la información sobre los documentos que se van a utilizar para cada búsqueda. Datos que se pueden crear antes de que el usuario haga la búsqueda. Ej. Diccionario IDF de <string,double> con <palabras , IDF de la palabra>
- **ModVec:** La clase donde se realiza todo el procesamiento de los textos de los documentos para realizar los cálculos del Modelo Vectorial utilizando las fórmulas del TF-IDF, los cálculos para obtener el peso de cada palabra de la búsqueda del usuario, y la similitud de coseno para calcular el score de cada documento según su relevancia.
- **Búsqueda:** La clase donde se trabaja con la query para crear una sugerencia, y para después con el método ModVec calcular el peso de cada una de sus palabras
- **Ops:** La clase con la que después de calcular el score de cada documento se le aplica el efecto de los distintos operadores ! , $\wedge$, $\sim$, * a los resultados
- **Matrices:** Una clase que no se ejecuta en el proyecto, pero que permite sumar, restar y multiplicar matrices.

Además, se hicieron cambios en la interfaz gráfica del Moogle!, agregando un fondo de pantalla, modificando la barra de búsqueda, y agregando la funcionalidad de que al tocar la tecla Enter se ejecute la búsqueda.

2 Funcionamiento

Cómo funciona Moogle! ??

1. Cuando se ejecuta el proyecto, se ponen en funcionamiento una serie de métodos que procesan el texto de los documentos, son los métodos de la clase Inicio, que llaman a algunos métodos de la clase ModVec, y para cuando aparece la página web del Moogle está creado un Diccionario de <documentos, Diccionario<palabras del documento, TF-IDF de cada palabra>>

2. Cuando introducimos una búsqueda en nuestra página, se comienza a ejecutar el método Query de la clase Moogle, que:

   (a) Primero, utilizando los métodos de la clase Búsqueda, trabaja con la query hasta crear la sugerencia, y después, utilizando los métodos de la clase ModVec crea un Diccionario de <palabras de la query, peso de cada palabra>

   (b) Posteriormente utilizando la clase ModVec, y los archivos creados al iniciar el Proyecto, se calcula la similitud de coseno de cada documento con la query, asignándole un score a cada documento y creando un Diccionario de <Documentos, Score del Documento>

   (c) Después utilizando los métodos de la clase Ops, se le aplica el efecto de los distintos operadores a ese diccionario, quedándonos con la parte de los documentos que nos importan para dar como resultados

   (d) Y a ese Diccionario en dependencia de si todas las palabras de la búsqueda tienen el operador ! o no, se le eliminan los documentos con score 0.

   (e) Después se ordenan los documentos y se dan como resultado los 5 primeros documentos, y además la sugerencia, que es lo que devuelve el método Query. Y esos documentos y la sugerencia que son devueltas son mostradas en la página donde el usuario hizo la búsqueda

3 Ejecución del proyecto

Cuando el proyecto comienza a ejecutarse, antes de abrirse la página web en la que el usuario introduce la búsqueda, se ejecuta el método Main de la clase Inicio de MoogleEngine, y ahí se comienza a hacer lo siguiente:

1. Primeramente, hay que acceder a la carpeta donde están todos los documentos para comenzar a procesar los textos de cada uno; para acceder a ella utilizamos el método Path.Join al que le indicamos que

debe salir al directorio padre y entrar en la carpeta Content, y devolver el path de acceso a esa carpeta que es dónde están todos los documentos.

2. Con el path de acceso a la carpeta donde están todos los documentos se utiliza de la clase Directory el método GetFiles que devuelve un array de string con los path a cada archivo txt que este guardado en la carpeta Content. Después a partir de esos path creamos un Diccionario de tipo <string,string> donde guardamos
<nombre de documento, texto del documento>, aquí el texto del documento se guarda con las letras en minúsculas pues de esta manera es más fácil contar las palabras, pues sino Papa y papa serian palabras distintas, todo esto lo hacemos utilizando el método DocText de la clase ModVec. Después recorriendo ese Diccionario, creamos otro nuevo Diccionario de <string, string[ ] > donde guardamos <nombre del documento, palabras separadas del documento>, estas palabras ya están sin signos de puntuación ni espacios en blanco, y esto lo hacemos utilizando el método ToTextDivide de la clase ModVec.

3. Con el método ToTF de la clase ModVec, creamos un Diccionario de <string,Diccionario<string, double>> donde guardamos
<nombre del documento,<palabras del documento, TF de cada palabra en el documento>.

   La fórmula que usamos para calcular el TF de una palabra en un documento es $TF_{palabra} = \frac{Freq}{MaxFreq}$;
   Freq = las veces que aparece la palabra en el documento;
   MaxFreq = la frecuencia de la palabra que más veces aparece en el documento;
   Con el método ToTF vamos recorriendo cada documento del diccionario anterior, por cada documento creamos un diccionario, e iteramos por el array de palabras del documento anterior, si la palabra en ese diccionario no está la agregamos con double 1 que sería la frecuencia, aparece una vez, si ya está le sumamos uno al double, la frecuencia, y guardamos el valor máximo de frecuencia, después que tengamos la frecuencia de cada palabra, volvemos a iterar por ese diccionario y dividimos cada frecuencia por el valor máximo de frecuencia en ese documento, y así tenemos el TF de cada palabra. Cuando una palabra aparezca mucho va a ser relevante para ese documento por tanto su valor de TF va a ser mayor.

4. Para calcular el IDF usamos la fórmula $\log(\frac{Freq}{Length})$
   Freq = Cantidad de documentos en los que aparece esa palabra

Length = Total de documentos
Cuando una palabra aparezca en muchos documentos, su valor de IDF va a estar más cerca de 1 y log(1) va a estar más cerca de cero, por tanto, su valor de IDF va a ser menor mientras aparezca en más documentos. El IDF lo calculamos con el método ToIDF de la clase ModVec, que recibe el Diccionario <nombre del documento,<palabras del documento, TF de cada palabra en el documento>> donde aparecen repetidas por documento la palabra una sola vez, y a partir de él crea un nuevo Diccionario de <string,double> compuesto por <palabras de todos los documentos, IDF de cada palabra>, primeramente va guardando la cantidad de documentos en que aparece la palabra, recorriendo cada documento del diccionario TF y y verificando si tiene ya agregada la palabra al Diccionario IDF, en caso de tenerla le suma 1 a su frecuencia de aparición; y en caso de no tenerla agrega la palabra, y le pone como frecuencia 1. Después vuelve a iterar por cada palabra, dividiendo cada frecuencia por el total de documentos y calculando el logaritmo de esa división, dejando creado un diccionario con los valores TF de cada palabra.
Ya teniendo ambos diccionarios TF e IDF, con el método ToTF_IDF de la clase ModVec calculamos el TF_IDF, que no es más que el producto del valor TF de cada palabra en el documento, por el valor IDF de la palabra. Este es un método void que recibe ambos diccionarios el de TF y el de IDF, y modifica el valor del TF de cada palabra en el diccionario
<nombre del documento,<palabras del documento, TF de cada palabra en el documento>> por el valor de TF_IDF, y lo que hace es ir iterando por cada documento del diccionario TF, y por cada documento va iterando por todas las palabras del IDF, verificando si están en el documento, en caso de estarlo multiplican el valor TF*IDF de esa palabra y ya guardan en ese valor como TF_IDF, en caso de no estarlo es porque la palabra no aparece en el documento, por tanto su TF es 0, y al multiplicarlo anularía todo el resultado, por tanto se pone directamente que el valor TF_IDF de esa palabra en ese documento es 0.
Aquí se termina de ejecutar el método Main, y posteriormente aparece la página del Moogle donde el usuario puede hacer sus búsquedas, al hacer una búsqueda se comienza a ejecutar el método Query de la clase Moogle, que da como resultado los documentos a mostrar y las sugerencias.

5. Con el diccionario de TF_IDF solo nos quedaría trabajar con la query

para hacer la sugerencia, crearle un peso a cada palabra de la query, y con la similitud de coseno hacer un sistema de scores que represente la relevancia de cada documento con respecto a la query.

- Primeramente para hacer la sugerencia, trabajamos con la query que se recibe en un string.
- Con el método **DivideQuery** de la clase búsqueda, convertimos todas las palabras de la query a minúsculas, y guardamos en un array la query con todas las palabras divididas.
- Después con el método **Clean** de la clase Búsqueda, creamos un array de la query nuevo, con las palabras divididas y sin los signos de puntuación, pero con los operadores.
- Después con el método **LowerString** de la clase Búsqueda, creamos un array nuevo con las palabras del array anterior pero con todas las palabras sin los operadores.
- Después con el método **SearchTheOne** de la clase Búsqueda donde creamos un nuevo array con las palabras de la query como el creado anteriormente, vamos verificando
  - si las palabras del array de palabras existen en el diccionario IDF que tiene todas las palabras, en caso de hacerlo las mantenemos igual, y en caso de no existir
  - verificamos también que esa palabra no sea "" porque en ese caso quiere decir que era o un signo de puntuación o un operador, y en ese caso no seria una palabra, por tanto no habría que buscarle ninguna palabra parecida,
  - En caso de no existir, y que no sea "", con la distancia levenshtein calculamos la palabra más parecida a ella entre todas las del IDF, guardando siempre la que menor distancia tenga y con esa nos quedamos, sustituyéndola entonces en la query.
- Después con el método **Change** de la clase Búsqueda, vamos iterando por cada palabra del array creado por SearchTheOne si es diferente a la palabra creada por el método LowerSTring, significa que la palabra cambio al aplicar la Levenshtein, en ese caso con el método Replace de la clase string, vamos a la posición del array con las palabras en minúscula y con signos de puntuación, y cambiamos la palabra vieja por la nueva, y al final con Split volvemos a unir todo el array de palabras ya modificado y esa es la sugerencia

Después con el método **ToTFQuery** de la clase ModVec calculamos el tf de la palabra, con la misma fórmula vista anteriormente $\frac{Freq}{MaxFreq}$, y lo hacemos a partir del array de palabras de la query que no ha sido modificado por la levenshtein, contamos las veces que aparece cada palabra, buscamos la frecuencia mayor y las dividimos, y guardamos un diccionario de <string, double>, con <palabras, tf de cada palabra>
Posteriormente con el método **ToQueryTF_IDF** de la clase ModVec creamos un diccionario de <string, double> que contiene <palabras de la query, peso de la palabra> y se calcula su el peso de cada palabra con la fórmula (a+(1-a)*TF de la palabra)*IDF de la palabra , donde a = 0,5;
Entonces lo que hace este método es ir iterando por el diccionario de <palabras, IDF de la palabra> creado anteriormente para los documentos , si la palabra aparece en el diccionario de
<palabras de la query, tf de la palabra>, aplicamos la fórmula anterior y la agregamos con ese valor de peso, en caso de no aparecer en el diccionario de palabras de la query le ponemos como valor 0, de esta manera si hay alguna palabra en la query que no aparezca en ningún documento es obviada.
Después se ejecuta el método **SearchScore** de la clase ModVec, que crea un diccionario de <Documentos, peso del documento>
Para el calcular el score utilizamos la fórmula de similitud de coseno, que busca la similitud entre el vector de palabras de cada documento, y el vector de palabras de la query, con la fórmula
$W_{iq} \rightarrow$ Peso de la palabra i en la query
$W_{iq} \rightarrow$ Peso de la misma palabra i, en el documento j

$$\frac{\sum_{i=1}^{n} W_{i,j} \times W_{iq}}{\sqrt{\sum_{i=1}^{n} W_{i,j}^2} \times \sqrt{\sum_{i=1}^{n} W_{iq}^2}}$$

Numerador = Sumatoria de: EL producto del peso de cada palabra en el documento por el peso de la misma palabra en la query
Denomindor = Sumatoria B = Sumatoria de los cuadrados de los pesos de cada palabra del documento.
El método recibe el diccionario de <documentos, <palabra del documento, tf_idf de la palabra>>

Y el diccionario de <palabras de la query, peso de la palabra>
Por cada palabra de la query en el diccionario de palabras en la query, va calculando la sumatoria de los pesos de la palabra en la query, y va calculando la sumatoria de los pesos de la palabra en el documento, y va calculando la sumatoria del producto del peso de la palabra en la query por el peso de la palabra en el documento, al final le halla la raíz cuadrada a la sumatoria A y a la sumatoria B, si el producto de las dos raíces es 0, el valor del score es 0 automáticamente, sino se hace la división y ese es el valor del score. $\wedge$, $\sim$, *

- Posteriormente con el método CleanDocs de la clase Ops, se aplica el efecto de cada operador a los resultados. Los operadores son:
  ! si aparece delante de alguna palabra significa que la palabra no puede existir en el documento
  $\wedge$ si aparece delante de alguna palabra significa que la palabra tiene que existir obligatoriamente en el documento
  * si aparece delante de una palabra, quiere decir que si la palabra está en alguno de los resultados, ese resultado tiene mayor relevancia, mientras más veces se repite el operador más relevancia tiene
  $\sim$ Aparece entre 2 palabras, quiere decir que mientras más cerca estén esas dos palabras más relevancia tiene el documento
  En caso de aparecer más de un operador delante de una palabra, solo se tendría en cuenta el primero de ellos.
  Entonces lo que va haciendo este método es, por cada operador va reccorriendo el array de palabras con operadores que habíamos creado anteriormente,

  si la palabra tiene el operador !
  - Verifica en el diccionario de <documentos, diccionario <palabras del documento, tf de la palabra>>, si el diccionario en ese documento contiene la palabra, se elimina el documento del diccionario de score.

  Si la palabra empieza con el operador $\wedge$
  - Verifica en el diccionario de <documentos, diccionario <palabras del documento, tf de la palabra>>, si el diccionario en ese documento no contiene la palabra, se elimina el documento del diccionario de score

Si la palabra empieza con el operador *
- Se cuenta cuantas veces aparece el operador
- Verifica en el diccionario de <documentos, diccionario <palabras del documento, tf de la palabra>>, si el diccionario en ese documento contiene la palabra, se multiplica el score de ese documento por la cantidad de veces que se repite el operador + 1.

Si la palabra comienza con el operador ∼
- Primero se verifica que no aparezca en la primera palabra, si aparece en esa palabra no se aplica
- SI aparece en cualquier otra posición, se verifica en el diccionario de <documentos, diccionario <palabras del documento, tf de la palabra>>, si el diccionario en ese documento contiene la palabra y la palabra anterior, en ese caso se cuenta cuantas palabras hay de diferencia entre las dos,
o si la distancia es 0, quiere decir que es la misma palabra, y se le suma 1 al score de ese documento.
o si la distancia es distinta de 0, se divide 1/distancia y el resultado se le suma al score de ese documento

Después con el método SortScores de la clase ModVec se ordena el diccionario resultante, guardándolo en una lista ordenada.
Y Después se dan los resultados:
- En el caso de que todas las palabras de la query empiecen con el operador ! significa que ninguna de esas palabras pueden aparecer, por tanto se quedarían solamente los documentos con score 0, si la lista está vacía, se devuelve SearchItem con (“”,“No hay resultados para su búsqueda”,0) y la sugerencia elaborada anteriormente, y eso es lo que se muestra como resultado.
Si la lista tiene menos de 5 elementos, se devuelve un SearchItem con <nombre del documento, snippet del documento, score del documento> de cada documento, y la sugerencia.
Y si tiene más de 5 elementos, se muestran los 5 primeros documentos
En el caso de que todas las palabras de la query no empiecen con el operador ! se eliminan los documentos con score 0, y se repiten las mismas condiciones para los resultados dichas anteriormente, en dependencia de la cantidad de documentos que queden en la lista.

También hay una clase Matrices en la que:

- El método **Suma**, recibe dos matrices A y B, donde A tiene que tener la misma cantidad de filas y de columnas que tiene B, A y B son matrices de MxN y el método devuelve la suma de A y B
- El método **Resta**, recibe dos matrices A y B, donde A tiene que tener la misma cantidad de filas y de columnas que tiene B, A y B son matrices de MxN y el método devuelve la resta de A y B
- El método **Producto**, recibe dos matrices A y B, donde A tiene que tener la misma cantidad de columnas que filas tiene B, A es una matriz de MxN y B es una matriz de NxK y el método devuelve el producto de A y B.
  Para el producto de una matriz por un vector, se puede utilizar este mismo método, donde A sería una matriz de MxN y el vector tendría que cumplir ser una matriz de NX1
  En caso de no cumplirse las condiciones necesarias para realizar la suma, resta y la multiplicación, cada método devolverá una matriz nula de 1x1
  También está el método ProductoWithScalar, que recibe un escalar y una matriz, y realiza el producto de un escalar por una matriz.