Moogle!

Facultad de Matemática y Computación
Universidad de La Habana

Eduardo Brito Labrada

May 9, 2023

Una breve introducción

Moogle! es una aplicación *totalmente original* cuyo propósito es buscar inteligentemente un texto en un conjunto de documentos. Es una aplicación web, desarrollada con tecnología `.NET Core 6.0`, específicamente usando Blazor como *framework* web para la interfaz gráfica, y en el lenguaje `C#`. La aplicación está dividida en dos componentes fundamentales:

- `MoogleServer` es un servidor web que renderiza la interfaz gráfica y sirve los resultados.
- `MoogleEngine` es una biblioteca de clases donde está... ehem... casi implementada la lógica del algoritmo de búsqueda.

¿Para qué sirve?

La idea original del proyecto es buscar en un conjunto de archivos de texto (con extensión `.txt`) que estén en la carpeta `Content`.

¿Cómo usarlo?

Primeramente, se aconseja a quien use esta aplicación tener instalado Linux, ya que no se garantiza la misma eficiencia si esta en un dispositivo que use Windows. En caso de tener instalado Windows, puede optar por instalar Windows Subsystem for Linux (WSL) que añade funcionalidades de Linux en Windows.

Instrucciones

Lo primero que tendrás que hacer para poder trabajar en este proyecto es instalar `.NET Core 6.0`. Luego, te debes parar en la carpeta del proyecto y dependiendo de tu Sistema Operativo hacer lo siguiente:

- **Linux:** Debes tener instalado `make`. Si no lo tienes instalado puedes instalarlo usando `sudo apt update && sudo apt install make`. Luego podrás hacer `make dev`
- **Windows:** Deberías poder ejecutar este proyecto usando `dotnet watch run --project MoogleServer`

Después de hacer lo anterior abre en tu navegador http://localhost:5000 y podrás usar Moogle! introduciendo tu búsqueda en la "entrada" y luego presionando el botón "Buscar".

Motor de búsqueda

El motor de búsqueda usa un modelo vectorial que nos computa para una *query* dada qué tan relevante es un documento determinado. Este modelo vectorial usa *Term Frequency and Inverse Document Frequency (TF-IDF)* con *Cosine Similarity* para computar la relevancia.

Para computar el vector *TF-IDF* uso la fórmula:

$$TFIDF = (\frac{tf}{tw}) \times \ln(\frac{td}{dt})$$

En esta fórmula tenemos que:

- tf es la frecuencia del término en el documento actual.
- tw es la cantidad de palabras totales en el documento actual.
- td es la cantidad total de documentos a analizar.

- dt es la cantidad de documentos que contienen el término.

Nota importante: dado que $\frac{td}{dt}$ puede causar problemas por la división entre 0 decidí que si $dt = 0$ luego $TFIDF = 0$, tiene sentido hacer esto ya que si $dt = 0$ el término no aparece en ningún documento.

Después de hacer lo anterior necesitamos calcular la "similitud" entre el vector de *document* y el vector de la *query*, que para eso utilizo *Cosine Similarity*. Para hacer esto intentamos estimar el "ángulo" comprendido entre el vector de la *query* y el vector de *document*: mientras menor sea este ángulo, mayor "similitud" tendrán estos vectores. Esto lo hacemos usando la fórmula:

$$\cos \alpha = \frac{v_d \cdot v_q}{||v_d||\ ||v_q||}$$

Siendo:

- v_d el vector de *document*
- v_q el vector de *query*
- $||v||$ es la magnitud del vector v

Sobre la implementación

`Utils.cs`

Esta clase está principalmente para métodos que son necesarios en varias clases del proyecto.

- `int EditDistance(string a, string b)`: devuelve el menor número de operaciones de edición que se deben hacer para igualar ambas cadenas (las operaciones son insertar un caracter, eliminar un caracter, cambiar un caracter por otro). Para que esto funcione uso un algoritmo de Programación Dinámica con una optimización en memoria para que en lugar de tomar $O(n \times m)$ en memoria tome $O(2 \times min(n, m))$, a pesar de esto la complejidad temporal sigue siendo la misma $O(n \times m)$, donde n y m son las longitudes de las cadenas.
- `int LongestCommonPrefix(string a, string b)`: esta clase recibe dos cadenas y devuelve el prefijo común más largo de dichas cadenas. Complejidad temporal $O(min(n, m))$ donde n y m son las longitudes de las cadenas.
- `double Distance(string a, string b)`: devuelve la similitud de dos cadenas usando el `Edit Distance`, pero esto no siempre funciona como se espera, por ejemplo `ebelabrada` está tan cerca a `eblabrada` como a `zbelabrada`, sin embargo debería devolver a `eblabrada` que tiene un mayor prefijo común. Para arreglar esto utilizo $\frac{ed}{lcp}$ donde ed es el resultado de `EditDistance` y lcp es el resultado de `LongestCommonPrefix`.
- `bool AreSimilar(string a, string b)`: devuelve `true` si dos palabras son similares, considero dos palabras similares si su distancia de edición es a lo más 1.
- `string Capitalize(string word)`: devuelve a `word` capitalizado.
- `string Tokenizer(string word)`: devuelve a `word` eliminando todos los caracteres que no sean letras o digitos.
- `List<string> NormalizeText(string text)`: dado un texto devuelve una lista de todas las palabras dentro del texto, además las palabras en esta lista serán devueltas "tokenizadas".
- `string[] GetNeed(string[] words)`: devuelve un `Array` de las palabras que tienen a `^` delante.
- `string[] GetForbidden(string[] words)`: devuelve un `Array` de las palabras que tienen a `!` delante.

- `(string, int)[] GetMore(string[] words)`: devuelve un `Array` de tuplas, en el primer "item" de la tupla está la palabra que contiene `*` delante y en el segundo "item" de la tupla contiene la cantidad de `*` delante que contiene la palabra.
- `(string, string)[] GetNear(string[] words)`: devuelve un `Array` de tuplas que contiene las palabras que están relacionadas por `~`
- `double Norm(Dictionary<string, double> vec)`: devuelve la norma de un vector.

TFIDFAnalyzer.cs

Esta es la clase más importante. En ella calculo el TF, el IDF de todas las palabras. Además de la relevancia de una palabra en un documento determinado.

- `TFIDFAnalyzer(string path)`: dado un `path` el constructor calcula el TF de cada palabra en cada documento, el IDF de todas las palabras, la relevancia de cada palabra en cada documento. Además, aquí se guarda la información calculada en los `.json` que aparecen en la carpeta `Database`, o se toma la información guardada si ya esta calculada, de esta forma se evitan hacer los cálculos dos veces.
- `bool CanGet(string database = "../Database")`: devuelve `true` si la cantidad de archivos que hay guardados en la carpeta `Database` es la necesaria para tener **toda** la información de **todos** los documentos en `Content`. Esto significa que si se modifica/elimina/añade algún documento no va a devolver `true`, o si no están los .json necesarios para recuperar la información tampoco devolverá `true`.
- `void SaveInfo(string database = "../Database")`: se encarga de guardar toda la información en los `.json` dentro de la carpeta `Database`.
- `void GetInfo(string database = "../Database")`: se encarga de obtener toda la información guardada en los `.json` que aparecen en la carpeta `Database`.
- `void DeleteInfo(string database = "../Database")`: se encarga de eliminar los `.json` que aparecen en la carpeta `Database`.
- `void ProcessDocuments(List<string> doc, int index)`: es un método auxiliar que dado el documento como una lista de palabras devuelve calcula el TF de cada una de esas palabras dentro del documento dado. `index` es el indice de este documento.
- `string Suggestion(string query)`: dada una *query* devuelve una sugerencia de búsqueda para esta *query*. Esta sugerencia es calculada usando el `EditDistance`, por cada palabra de la *query* busca la palabra dentro del vocabulario que tenga menor distancia de edición con ella.
- `double OperatorIn(string[] words, int index)`: este método es utilizado para el operador `^` devuelve 1.0 si todas las palabras en `words` aparecen en el documento con índice `index`, de lo contrario devuelve 0.0.
- `double OperatorNotIn(string[] words, int index)`: este método es utilizado para el operador `!` devuelve 1.0 si ninguna de las palabras en `words` aparece en el documento con índice `index`, de lo contrario devuelve 0.0.
- `double OperatorMore((string, int)[] words, int index)`: dado las palabras que contienen `*` y la cantidad de veces que aparece el `*` delante, devuelve $\prod more * \ln(freq_{words})$ donde $more$ es la cantidad de veces que `*` aparece delante de la palabra y $freq_{word}$ es la cantidad de veces que $word$ aparece en el documento con índice `index`.
- `double OperatorNear((string, string)[] words, int index)`: por cada par de palabras en `words` calcula la mínima distancia entre las apariciones de estas palabras en el documento con índice `index`, esa mínima distancia (llamemosle md), la utilizo para calcular lo que le aporta a la respuesta, $\prod \frac{2}{\ln(md)+1}$.

- `double ComputeRelevance(Dictionary<string,double> queryVec, int index, string[] need, string[] forb, (string, int)[] more, (string, string)[] near)` este método recibe el vector de la *query*, el índice del documento en el que se va a analizar la query, las palabras que contienen `^`, `!`, `*` delante, y las que están asociadas por `~`. Devuelve la relevancia de la query dentro del documento, utilizando la fórmula mencionada en **Motor de búsqueda**, ese valor multiplicado por el resultado de los últimos cuatro métodos explicados anteriormente.