Reporte sobre buscador Moogle!

Darío Hernández Cubilla C-113

15 de julio de 2023

Índice

1. Document Engine **1**
- 1.1. Construcción 1
- 1.2. Utilidades 2
- 1.3. Implementación de `editDistance` 2
- 1.4. Recuperación de información 3

2. Query Engine **4**
- 2.1. Utilidades 4
- 2.2. Construcción 4
- 2.3. Diccionario de Sinónimos usado 5

3. Snippet Engine **5**

4. Matrix **5**

Resumen

En este reporte se expondrán las ideas esenciales de la implementación del buscador Moogle!, como primer proyecto de programación de primer año en la carrera de Ciencias de la Computación en la Universidad de la Habana.

Para ello se explorará el funcionamiento a grandes rasgos -y se analizará a más profundidad cuando sea necesario-, de las diferentes clases que componen el proyecto, que permite la recuperación de información de una serie de archivos .txt usando una query entrada por el usuario, a través del modelo vectorial de recuperación de información, con el objetivo de facilitar ambos el entendimiento del sistema creado y la lectura del código.

1. Document Engine

1.1. Construcción

Primero que todo, los documentos se deben guardar en la carpeta Content de moogle!. La primera vez que se llama al método `Query` de moogle.cs, se instancia un objeto de tipo `Corpus` en el Document Engine, Este objeto se crea usando el modelo vectorial como forma de simplificar los documentos en Content para la recuperación de información.

El proceso de construcción es simple, primero, se comprueba si se tienen archivos de caché de alguna ejecución anterior, estos son guardados en la carpeta MoogleEngine de moogle!, de ser así, se comprueba si la carpeta Content no ha sido modificada después de la creación de dicho caché, en este caso, simplemente se cargan los datos hacia la nueva instancia de `Corpus`. En caso contrario, el motor lee todos los archivos .txt de la carpeta Content, creando un array de Documentos (estructura que guarda el path del documento que representa, así como el vector de tf-idf que le corresponde), guardando en su campo de `Vector` un vector con el tf normalizado de las palabras del documento (la cantidad de veces que cada palabra aparece dividida sobre la cantidad de palabras totales en el documento es la coordenada de la dimensión que representa a cada palabra en este vector), de este array se extrae el IDF (logaritmo natural del cociente entre la cantidad de documentos en el Corpus y la cantidad de documentos en las que la palabra dada aparece, que indica la rareza de una palabra en el Corpus y, por tanto, su importancia en este) de cada palabra, contando primero en cuántos documentos aparece cada palabra y luego usando este número para hallar su IDF y guardarlo en el diccionario `Lexicon`, el cual usa luego para añadir el IDF a las palabras de cada vector en el array de documentos. Ya creado cada uno de estos vectores, se calcula su Norma, para que no deba ser computada en cada cálculo de coseno entre este vector y uno de una query. Ya instanciado Corpus, se crea un nuevo caché para ejecuciones posteriores, si un caché no fue usado para instanciarlo.

1.2. Utilidades

Este motor también tiene formas para preguntar desde otras clases si la palabra se encuentra en el Lexicon, cuál es su Idf, si es considerada Stop Word (esto si aparece en más de o exactamente el 95 % de los documentos), además de buscar la palabra del Lexicon con la menor distancia de Levenshtein a una palabra dada, esto se usa para resolver errores de ortografía en una query, por ejemplo. Para ello se tiene un método `editDistance` que crea una matriz en la cual cada casilla representa cuán complejo es convertir una subsecuencia de una string en una subsecuencia de la otra, la complejidad es la menor cantidad de operaciones entre reemplazar, añadir o eliminar un carácter de una subsecuencia que se deben hacer para convertir la una en la otra.

1.3. Implementación de `editDistance`

Para elaborar sobre cómo el método `editDistance` funciona, podemos imaginar la matriz creada por este como sigue, tomando por ejemplo la coordenada (indizando en 0) (2, 3) como la complejidad del problema de convertir a “am” en “o”, o al de las coordenadas (1, 1) el de convertir a “” en “”, cuya complejidad es claramente 0, pues son iguales:

	””	a	m	o	r
””	0	1	2	3	4
o	1	1	2	2	3
d	2	2	2	3	3
i	3	3	3	3	4
o	4	4	4	3	4

Para convertir una cadena vacía a otra cadena de tamaño `n` simplemente se hacen `n` operaciones de inserción, por eso la primera (segunda si tenemos en cuenta los nombres de las filas y columnas) columna de la matriz, que denota la complejidad de convertir a "" en "", "o", "od", "odi" y "odio" respectivamente es simplemente 0, 1, 2, 3, 4, porque esa es la cantidad de inserciones que se deben hacer para convertir una cadena vacía en otra, análogamente para la primera fila, pero con eliminaciones. Llenadas las primeras fila y columna, nos situamos en el problema de convertir "a" en "o", lo que hace este modelo, es ver qué operación requiere menos pasos, las operaciones se definen de la siguiente forma, dados dos strings, como "perro" y "perro", nos situamos en el último carácter, si son iguales, como en este caso, la complejidad de convertir una en la otra es simplemente la de convertir la primera sin el último carácter en la segunda sin el último carácter, o sea "perr" en "perr", es claro aquí que el caso trivial dará una distancia de 0. Si son diferentes, como en "amor" y "odio", para cada último carácter de dos subcadenas dadas, podemos hacer tres operaciones:

1. Insertar, que se basa en convertir la primera cadena en la segunda sin el último carácter, es decir "amor" en "odi" y después añadir a esta dicho carácter, o sea "o".

2. Reemplazar, que se basa en convertir la primera sin el último carácter en la segunda sin el último carácter y después añadir a esta dicho carácter.

3. Eliminar, que se basa en convertir la primera sin el último carácter en la segunda

Dada una casilla, podemos ver la casilla directamente superior como el subproblema 1, la diagonalmente superior izquierda como el 2 y la directamente a la izquierda como el 3, en caso de que los últimos caracteres sean iguales para un subproblema, simplemente se toma la complejidad del subproblema 2, convertir la primera sin el último carácter en la segunda sin el último carácter, en caso contrario, se toma el mínimo entre las operaciones adyacentes y se le adiciona 1, para indicar que se hizo alguna operación para llegar a este subproblema. Se empieza en la primera casilla vacía superior izquierda (convertir "a" en "o" en el ejemplo dado) y se va llenando la matriz con dichas reglas tomando siguiente a la derecha, columna a columna, fila a fila, se devuelve la complejidad de convertir una cadena en la otra, la complejidad en la casilla inferior derecha, en este caso, la de convertir "amor" en "odio", que es de 4 operaciones.

1.4. Recuperación de información

El lugar más importante de la clase Corpus es el método que toma una query y devuelve un array con los documentos, convertidos en `SearchItems`, más similares a esta, usando el modelo vectorial: `GetClosestDocuments`. La computación es de manual, se da una query y se calcula el coseno del ángulo que separa el vector de tf-idf que le corresponde y cada vector del array de documentos del Corpus (que conocemos que es el producto punto del vector que representa la query por el que representa el documento, dividido por la multiplicación de las longitudes (normas) de dichos vectores), estos se van añadiendo de forma ordenada a una lista, comparando con dicho coseno, en orden descendente, que después de que la iteración por todo dicho array termina, es convertida en un array de `SearchItems`, con una `Snippet` computada por `SnippetEngine`. Es importante tener en cuenta que antes de calcular el coseno del ángulo entre los vectores que representan a la

query y al documento se comprueba si el segundo contiene las palabras que el usuario exigió se incluyeran en los resultados con el operador '^' y si no contiene las palabras cuya exclusión fue exigida con '!' (operadores discutidos en Query Engine), si no cumple estas condiciones, el documento es ignorado.

En caso de que la cantidad de resultados enviados por este método sea insatisfactoria, se llama otra vez, esta vez con la misma query, pero incluyendo los sinónimos de las palabras en la búsqueda, ampliándola, por así decir. En caso de que la búsqueda sea ampliada, se indica usando el parámetro `ExtendedSearch` de `GetClosestDocuments`, que disminuye un 25 % el score de cada `SearchItem` que devuelve. Las búsquedas son después unidas ordenadamente, si hay documentos repetidos se toma el de mayor score (uno será con y el otro sin los sinónimos, posiblemente tendrán snippets diferentes, por eso se toma el más similar a la query).

2. Query Engine

2.1. Utilidades

Este es el motor que computa cada query enviada por el usuario, soporta los operadores '^' (inclusión exigida), '!' (exclusión exigida) y '*' (aumento de la importancia), además de que comprueba la correctitud ortográfica de la query basándose en el Léxico de los archivos en Content y las palabras en el diccionario de sinónimos en sinonimos.json en MoogleEngine. La clase `Query` contiene listas que denotan las palabras que deben ser excluidas e incluidas, además de un booleano `errorFound`, que indica si se encontró un error ortográfico en la query, un booleano `synonymsFound` que indica si alguna de las palabras en la query tiene sinónimos en el diccionario de sinónimos (para evitar ampliar la búsqueda si hacerlo no implicaría cambios en la query) y una string `suggestion` que se debe usar si un error fue encontrado (contendrá una copia de la query con las palabras corregidas). Además de esto contendrá un vector de tf-idf que la represente además de su norma, para sólo computarla una vez, cuando se instancie una query.

2.2. Construcción

Para computar la query se escanea el texto que la representa, se divide en palabras y se tienen en cuenta solo los operadores inmediatamente antes de ellas (en el caso del operador '*' se tienen en cuenta todas sus repeticiones, mientras no estén separadas por otros símbolos, cada una aumenta la importancia de la palabra exponencialmente, un aumento lineal era muy poco significativo), todo otro símbolo es ignorado.

El constructor de `Query` también toma un booleano `AddSynonyms` (`false` por defecto) que indica que se debe ampliar la búsqueda en esta query. Se comprueba si cada palabra se encuentra en el Lexicon del Corpus, de no estarlo pueden pasar dos cosas, o la palabra tiene un error ortográfico o simplemente no aparece en el Corpus, pero sí existe: si la palabra no se encuentra en el Lexicon, se busca en el diccionario de sinónimos, si está allí se hace `true` al booleano `synonymsFound` y si se indicó que se ampliara la búsqueda, se añade a una lista de palabras cuyos sinónimos añadir al vector de la query, si la palabra no está ni en el Lexicon ni en el diccionario de sinónimos, se asume que tiene un error ortográfico y se busca la palabra más cercana en el Lexicon (usando `editDistance`) para

tomar su lugar en la query, con un tf menor. Ya separadas todas las palabras en la query se añade su idf usando el Lexicon del Corpus.

2.3. Diccionario de Sinónimos usado

El diccionario de sinónimos que se ha referenciado se extrae de sinonimos.json en MoogleEngine. Cada palabra en sinonimos.json puede aparecer en varias listas de sinónimos, para sus diferentes acepciones, por esto cada palabra en el diccionario de sinónimos (que está implementado como un diccionario de C# en mi proyecto) tiene asignado una lista de listas de `strings` (sinónimos), en cambio de simplemente una lista de sinónimos por palabra.

3. Snippet Engine

Este motor se usa sobre los documentos más cercanos a una query dada, desde el método `GetClosestDocuments` de `DocumentEngine`, por tanto se llamará una cantidad relativamente pequeña de veces, tan pequeña como la cantidad de resultados que se le pida a `GetClosestDocuments`. Aquí se tiene una clase que representa una sliding window con una cantidad especificada de palabras n (60 por defecto) que representa un candidato de snippet para un documento, cuando se llama a `GetSnippet`, se escanea el documento especificado en su parámetro `filePath`, usando la sliding window para calcular la similitud de cada posible snippet de n palabras con la query, usando el modelo vectorial, buscando además que alguna palabra de la query esté relativamente céntrica en la snippet. Simplemente se devuelve la snippet con mayor puntaje encontrada. Si una snippet válida es encontrada en los primeros 5000 caracteres de un documento, no se sigue explorando a este, para evitar búsquedas muy largas, el trade-off con el posible contexto perdido vale la pena, para documentos largos el proceso de búsqueda se puede extender prohibitivamente si se hace hasta el final.

4. Matrix

Esta clase no es usada en el proyecto pero se entrega dentro de la carpeta MoogleEngine como fue requerido.