GESTOR DE DOCUMENTS

Algorismes i Estructura de Dades

Projectes de Programació

Quadrimestre Tardor 2022-23

Grup 13.4

Dante de Prado Rojo

Pol Salvador Nogués

Sergio Sanz Martínez

Youcef Trabsa Biskri

Algorismes i estructures de dades

1. Tf-idf i cosine similarity

Per aconseguir els documents similars a un document hem utilitzat el *cosine similarity* dels vectors *tfidf* de cada document.

Per obtenir el vector *tfidf* de cada document és necessari obtenir primer el *tf (term frecuency)* de cada document. Per fer-ho utilitzem un *HashMap<String, Integer> wordCount* on guardem quantes vegades apareixen les paraules al document (excluint les de una llista de *Stopwords*) , i amb aquestes dades podem calcular el *HashMap<String, Float> tfMap* on guardem el *term frecuency* de cada paraula del document. Calcular-lo té un cost *O(n)* on *n* és el nombre de paraules úniques del document.

$$tf(t, d) = \frac{vegades\ que\ apareix\ el\ terme\ t}{total\ de\ paraules\ del\ document\ d}$$

Cada vegada que s'afegeix/s'esborra/es canvia el contingut d'un document s'agafa el *wordCount* d'aquest document i s'actualitza el *HashMap<String, Integer> globalWordCount* que guarda cada paraula del sistema amb el nombre de documents en els que apareix.

L' *idf* només es calcula (si fa falta) quan es fa una consulta *getSimilarDocuments.* Calcular-lo té cost *O(n)* on *n* és el nombre total de paraules diferents al sistema. Es guardarà el resultat a *HashMap<String, Float> idfMap.*

$$idf(\text{t, D}) = \log \frac{total\ de\ documents\ D}{documents\ en\ els\ que\ apareix\ el\ terme\ t}$$

El vector *tfidf* de cada document es calcularà també només quan es faci aquesta consulta i sigui necessari, per fer-ho cada document rebrà per paràmetre l' *idfMap* i calcularà el valor *tfidf* de cada paraula multiplicant els valor que té aquesta al seu

tfMap amb el que té la paraula a l' *idfMap.* L'operació tindrà també cost *O(n)* on *n* és el nombre de paraules úniques del document.
La funció que calcula el *tfidf* retornarà un *HashMap<String, Float>* on es guardarà cada paraula i el seu valor *tfidf.*

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D)$$

Una vegada tinguem els vectors *tfidf* dels documents a comparar, aplicarem la fòrmula del *cosine similarity* per veure el seu nivell de semblança.

sim = $\frac{A \cdot B}{|A| \cdot |B|}$

On *A, B,* son els vectors *tfidf* dels documents a comparar. Quan més aprop estigui *sim* de 1, més semblança tenen els documents comparats.

Comparar un document D amb tots els altres té cost *O(n),* on *n* és el nombre de documents. Comparar dos documents entre si té cost *O(m)* on *m* són les paraules del document D. El cost total és *O(n·m).* El resultat es guarda a *HashMap<String, Float> d1map,* on es guarda la clau del segon document i la seva similaritat amb el primer. Amb la funció *sortByValue* es crea un ArrayList de documents a partir de *d1map* ordenat segons la similaritat i s'inserta al *HashMap<String, ArrayList<Document> similarDocs* amb la clau del document D. D'aquest manera, si fem una consulta *getSimilarDocuments* i el *HashMap<String,Document> docs* no ha estat modificat (es guarda a la variable *modified*), si ja hem calculat els documents similars podrem buscar al *HashMap<String, ArrayList<Document> similarDocs* i obtenir els documents similars amb cost *O(k),* sent *k* el número de documents similars que es vol retornar.

Estructures de dades utilitzades:
A Document:

- *HashMap <String, Integer> wordCount* — paraula, vegades que apareix
- *HashMap <String, Float> tfMap* — paraula, term frecuency
- *Float dist* — distància del vector *tfidf*

A DocumentController:

- *HashMap <String, Double> idfMap* — paraula, inverse document frecuency

- *HashMap <String,Integer> globalWordCount* — paraula, documents en els que apareix la paraula
- *HashMap<String, ArrayList<Document> similarDocs* — clau de un document, *ArrayList<Document>* amb els documents ordenats per similaritat.
- *Bool modified* — s'utiliza per guardar l'estat dels documents
- *HashMap<String, Float> tfidf* — paraula de un document, valor *tfidf*, mai es guarda aquest atribut només es calcula cada vegada que es comparen documents i *modified = true,* o quan no s'han calculat encara els documents similars d'un document.

2. Boolean Expressions

2.1. Node

La classe Node representa un node de l'arbre binari en el que es conté una expressió booleana.

Aquesta classe és abstracta i té tres filles: BinaryNode, NotNode i StringNode.
Entrem en profunditat per saber com funcionen aquestes.

2.2. BinaryNode

Aquesta classe és abstracta i és implementada per les seves dues filles: AndNode i OrNode. Bàsicament són un tipus de node que té un fill a l'esquerra i un fill a la dreta. Cada fill és la subexpressió que es troba abans (fill esquerra) o després (fill dreta) de l'operador. Un exemple simple seria el següent; suposem que tenim l'expressió {p1 | p2 & p3}, el fill esquerra del OrNode seria p1 i el dreta seria la subexpressió p2&p3 parlant conceptualment. Si volem concretar, el fill dreta seria un AndNode que tendria com a fill esquerra p2 i com a fill dret p3.

2.3. NotNode

NotNode, al igual que AndNode i OrNode és concreta i serveix per implementar el operador not, que es un tipus de node que només té un fill ja que l'operació not no

és binaria. Aquest fill pot ser tant un BinaryNode (AndNode o OrNode) com un StringNode.

2.4. StringNode

Per últim tenim la classe StringNode, que s'encarrega d'implementar les fulles de l'arbre, que sempre són una string. Aquesta string pot ser una secuencia de paraules o una sola paraula i sempre son filles d'un NotNode, OrNode o AndNode.

3. Boolean Parser

La classe Node ens permetrà crear un algoritme per transformar una llista de Strings a un arbre. Primer necessitem la llista que en aquest cas serà una ArrayList degut a que es una llista en la que es van afegint elements. Això vol dir un .add(string) per cada element rellevant de l'expressió (tot menys espais que no estiguin en frases) i fer un .add(string) en un ArrayList té cost O(1). A més, per la creació de l'arbre es farà un .get(int) per cada element de la llista i el get en un ArrayList també té cost O(1). Per tant, ens ha semblat la millor opció.

Al tenir l'ArrayList amb els element de la expressió, ja podem crear l'arbre binari que la contendrà. Aquest arbre té el següent funcionament: Les operacions amb més prioritat van el més abaix a l'esquerra possible i a l'hora de evaluar si una frase compleix l'expressió de l'arbre es començará per abaix a l'esquerra, el que significa que primer es comproven les expressions amb més prioritat, que és l'objectiu de l'operació.

Una vegada creat el binary tree, guardem a la classe BooleanExpression el node arrel de l'arbre i l'enviem a la classe ResultDocuments per a que es pugui comprovar si la expressió booleana continguda a l'arbre es compleix per a alguna de les frases de tots el documents que es troben en aquesta classe.

Aquest procés té el següent cost: En el pitjor dels casos (no es troba cap frase que compleixi l'expressió) es recorre per tots els documents de ResultDocuments i per

totes les frases de cada document. Si suposem **n** documents amb una mitjana de **m** frases cadascun en surt un cost en el pitjor dels casos de **n x m x t**, on **t** es el número de nodes de l'arbre (o el que és el mateix, el número de operadors i operands de l'expressió booleana).
En el millor dels casos (es compleix l'expressió al principi de tots els documents) el cost serà **n x m** ja que no s'han de recórrer totes les frases.

4. BinarySearch

Aquest algoritme l'utilitzem per trobar un element dins un ArrayList, dividint-lo iterativament per la meitat fins trobar l'element desitjat. Té l'avantatge de que el cost en el millor dels casos és o(1) i en el mitjà i en el pitjor dels casos log(n).

Ha estat utilitzada per certes cerques i per trobar elements per prefix. En el cas dels autors, busca el String més petit que comença pel prefix i, a partir d'aquesta posició, va una per una afegint les paraules següents a la llista de resultats de la consulta. Després és un cost lineal per cada element a l'ArrayList que comença pel prefix.
Això és possible degut a que el conjunt d'autors s'ordena alfabèticament en afegir autors.

5. Estructures de dades importants del sistema

El sistema també consta d'altres estructures de dades rellevants utilitzades en diferents classes:

- ArrayList (AL):
    - Relacions entre classes: utilitzada a Author (AL de Document), AuthorController (AL d'Author), Document (AL de Phrase).
    - Atributs de classe: a ResultDocuments (AL de Document),
    - Hem preferit utilitzar ArrayList en lloc de LinkedList en ser més eficient quan hi ha un gran nombre d'elements, per tant, suposem que hi ha un gran nombre d'elements al nostre programa.
    - A més, té cost Θ(1) les funcions get() i add(), que utilitzem molt a l'hora de retornar els resultats de les consultes. També és un punt important

el fet que la funció sort() actuï com a QuickSort amb String i MergeSort amb Objectes, els dos amb cost mitjà Θ(nlogn), ja que els utilitzem, per exemple, per ordenar autors en afegir-hi nous, o per ordenar els Document a ResultDocuments.

- HashMap (HM):
    - Relacions entre classes: utilitzada a BooleanController (HM del nom modificat pel correcte funcionament del programa i BooleanExpression, de nom de l'expressió modificada i de l'expressió sense modificar), DocumentController (HM de nom i Document),
    - Atributs de classe: utilitzada a Document (HM de nom i int, de nom i float) i a DocumentController (explicat a l'apartat del tf-idf).
    - La principal avantatge del HashMap per la que ens hem decidit a utilitzar-la és que la cerca d'un element és de cost Θ(1) i, per tant, molt eficient buscar un sol element. Això només és eficient quan no necessitem tenir ordenades les dades i no hem necessitat tenir-les per aquestes relacions i atributs.