Peter El Jiz
62506661
Project 2 Report

Data structure and architecture description:

The Data Structure and architecture used is practically similar to Project1's, with some minor modifications. We used to parse a file, and store docID-term in a Map<docID, term>. We changed that to include the term frequency ==> Map<docID, Map<term, frequency>>. When a same word is found more than once in a document, the frequency is updated to the new count. We also create a new Map<Integer, Integer> docs_TokensAm, which will store the amount of tokens for each document. We will use this to find the total amount of docs, the average amount of tokens per document, to calculate a score(Document, query) value for ranked-retrieval.
We also modified our index data structure from TreeMap<String, ArrayList<Integer>> to TreeMap<String, Map<Integer, Integer>>. This will allow us to store for each term, the document and term frequency of that term in that document. We will later use this frequency value for the score calculation of the basic term frequency query processing algorithm.
For the BM25 and TF algorithms, I used a new map<docID, score> to store the document ID and score for that document.

Inverted Index Data Structure:

TreeMap<String, Map<Integer, Integer>>. It is a TreeMap with the term being a key, and the value being another Map that stores docID and term frequency of that term in that document.

High Level Description of the two implementations with file names and line numbers to where we can see the implementation:

TF algorithm implementation: (Implementation available in Spimi.java at line 740)
Create a docsFrequencies map<Integer, Double>
Retrieve query
Process it, delete trash data
stem words
add words to list

For each term in list
  For each document (in our index) of that word
    if document exists in our docsFrequencies ===> new score = old score + term frequency of that term
    if document does not exist ===> add document + term frequency of that term

//Sort
Create a new TreeMap<Double, ArrayList<Integer>> sortedFrequencies
Iterate through docsFrequencies.
For Each Entry
if (frequency exists in sortedFrequencies) ==> grab list of docs having that frequency ===> add to that list
if (frequency does not exist in sortedFrequencies) ==> create new docs arraylist ===> add doc to that list

//Retrieve top 5 documents
Iterate till elements are done or till we reach 5 elements
Add top5 documents
Return List

Pass that list to our retrieveDocumentsWithIDs method

Since we are adding the same word (that may appear in a query more than once) multiple times in our query terms list, we do not need to multiply the amount of times it appears by the document score/frequency.

BM25: (Implementation available in Spimi.java at line 916)
Create a docsScore map<Integer, Double>
Retrieve query
Process it, delete trash data
stem words
add words to list

For each term in list
  For each document (in our index) of that word
    if document exists in our docsScore===> new score = old score + new score calculated of that term
    if document does not exist ===> add document + initial score calculated (using q0)

//Sort
Create a new TreeMap<Double, ArrayList<Integer>> sortedScore
Iterate through docsScore.
For Each Entry
if (score exists in sortedScores) ==> grab list of docs having that score ===> add to that list
if (frequency does not exist in sortedScores ==> create new docs arraylist ===> add doc to that list

//Retrieve top 5 documents
Iterate till elements are done or till we reach 5 elements
Add top5 documents
Return List

Pass that list to our retrieveDocumentsWithIDs method

Influence of k:

BM25 is used to rank a set of documents based on the query terms and other variables. Using each term's frequency, the number of documents containing each term (individually), the length of each document, the average length of documents, and the total number of documents, we compute a score for each document. The top 5 documents are selected. The BM25 algorithm takes 2 variables as parameters. We set b at 0.75 and fiddle with k.
On one extreme, we have k = 0, and on the other we have k = 50.

We notice that when k = 0, the algorithm behaves similarly to a boolean retrieval model (weighted zone scoring). "Weighted zone scoring assigns to the pair (q, d) a score in the interval [0, 1], by computing a linear combination of zone scores, where each zone of the document contributes a Boolean value". (Information retrieval book)

Document ID: 20614 Score:29.40692384683372
dow: 1 jone: 1 great: 3 depress: 4

Document ID: 20860 Score:29.40692384683372
dow: 2 jone: 1 great: 1 depress: 1

Document ID: 20928 Score:29.40692384683372
dow: 1 jone: 1 great: 1 depress: 1

Document ID: 20964 Score:29.40692384683372
dow: 1 jone: 1 great: 1 depress: 1

Document ID: 14505 Score:22.808311557057028
dow: 1 jone: 1 great: 0 depress: 1

Document ID: 20021 Score:22.808311557057028
dow: 1 jone: 1 great: 0 depress: 1

We also notice that the combined term frequencies for the query terms are somewhat in descending order.

dow (=1) + jone(=1) + great(=3) + depress(=4) = 9
dow (=2) + jone(=1) + great(=1) + depress(=1) = 5
dow (=1) + jone(=1) + great(=1) + depress(=1) = 4
dow (=1) + jone(=1) + great(=1) + depress(=1) = 4
dow (=1) + jone(=1) + great(=0) + depress(=1) = 3

It tries to find the best documents where all query terms contribute with minimal attention to how much words contribute, or the overall content.

We notice that for k = 50,

Document ID: 10007 Score:71.04560817659126
dow: 0 jone: 12 great: 0 depress: 0

Document ID: 20964 Score:58.595000062324786
dow: 1 jone: 1 great: 1 depress: 1

Document ID: 20992 Score:54.927431958886956
dow: 2 jone: 1 great: 0 depress: 0

Document ID: 3775 Score:43.48428187181787
dow: 0 jone: 7 great: 0 depress: 0

Document ID: 12132 Score:41.18571574039457
dow: 7 jone: 0 great: 0 depress: 0

Document ID: 10012 Score:40.32113904898843
dow: 3 jone: 2 great: 0 depress: 0

There is no weighted average or anything, the algorithm is a bit dull, it relies only on term frequencies. This means that it doesn't actually check the validity of the content. A document that scores high due to a high frequency (1st document has 12 occurrences of "jone", no occurrence of remaining terms) is ranked as important as any other document. This doesn't really make sense, we don't want documents that only mention "jone" to be our top-rated document.
The algorithm resembles the basic Term Frequency scoring algorithm, which we will explore later. If we look closer, we notice that the first document in both algorithms is the same.
We conclude that as k reaches a higher value, it stops taking into consideration the overall query, but adds greater importance to term frequencies, resulting in frequent occurrences of individual words only.

We notice that for k = 1.2, the algorithm is half-way there

Document ID: 20964 Score:40.681513752964094

dow: 1 jone: 1 great: 1 depress: 1

Document ID: 14505 Score:27.181967878905652
dow: 1 jone: 1 great: 0 depress: 1

Document ID: 20992 Score:25.89012871792292
dow: 2 jone: 1 great: 0 depress: 0

Document ID: 10012 Score:23.893510078858505
dow: 3 jone: 2 great: 0 depress: 0

Document ID: 14638 Score:22.147255237516806
dow: 1 jone: 1 great: 0 depress: 0

Document ID: 3987 Score:21.967036655433727
dow: 5 jone: 3 great: 0 depress: 0

The documents in which words share the same impact are the highest scored. We see a pattern in the above top 5 documents for the BM25 algorithm (where k = 1.2). The first document has dow=jone=great=depress=1. The next document lacks the term “great”, it is followed by document that lacks both “great” and “depress” and compensates with an extra “dow” (dow = 2). The document after that, has an extra jone (jone =2) and a 3rd dow (dow=3). We notice that as the term frequencies of “dow” and “jone” increase inconsistently, it disproportionately and negatively affects the overall score.
k=1.2 is a middle balance between a boolean model algorithm (no term frequencies) and k=50 (raw term frequencies)

Differences in querying using term frequency scoring and BM25:

Term Frequency:
Document ID: 10007 Score:12.0
dow: 0 jone: 12 great: 0 depress: 0 = 12

Document ID: 20614 Score:9.0
dow: 1 jone: 1 great: 3 depress: 4 = 9

Document ID: 3987 Score:8.0
dow: 5 jone: 3 great: 0 depress: 0 = 8

Document ID: 18652 Score:8.0
dow: 0 jone: 0 great: 8 depress: 0 = 8

Document ID: 3775 Score:7.0
dow: 0 jone: 7 great: 0 depress: 0 = 7

Document ID: 8009 Score:6.0
dow: 0 jone: 0 great: 3 depress: 3 = 6

The scoring function in the basic term frequency algorithm is obvious: the document that has the most overall terms is ranked highest.

Compared to a more sophisticated algorithm like BM25, which takes into account the average document length, number of documents containing each term (individually), the length of the current document, and the total number of documents, TF only looks at term frequencies (the actual content may be way off and even irrelevant). This means that TF may retrieve irrelevant documents which contain a lot of terms in our query.

Although TF may resemble BM25 when k is high, the documents differ a bit due to the simple reason that BM25 takes extra variables, mentioned above, into consideration.