Inverted Index Write Up

Shivangi Singh

1) **10%) Description of the system, design tradeoffs, questions you had and how you resolved them, etc. Document your index and retrieval APIs.**

Inverted Index:
An array [DocID, NumofTermsInDoc = n, Positions(n), DocIDNumofTermsInDoc = n, Positions(n),...]
Where 1st term is DocID followed by number of terms in the doc, followed by positions of the words in the documents.

Uncompressed: Is stored in a Binary File UCIndex.txt where each element is represented in 3 bytes. (That unnecessarily added more space to the file- had to do 3 byte encoding due to a python quirk) as the byte() function could not represent more than 255.

Compressed: Is stored in a Binary File CIndex.txt where the positions are delta encoded to reduce space and further I have done vbyte compression and written to the file. This method is likely to save I/O time but it takes time to decode back to original stuff.

Problems: As we don't have a designated number of bytes for int (they can expand into float). I had some problem writing the pseudocode directly. Had to use some math concepts like dividing by 2^n shifts the bits right. (Spent a lot of time on this)

Also, when I was running the Dice Coefficient it was taking a long time to run as I was opening the LookUpIndex file to find where exactly the term was stored in the file.

Querying: Document at a Time, gives the top 5 documentID's and the name of the Documents can be found from the DocID file which is a json whose (k,v) = (docID, scene)

APIs:
UncompressedLookup: return a dictionary whose key is the term and values contain {offset,size, number of documents in the collection containing the term
, number of terms in collection } for the uncompressed index binary file.

CompressedLookup: return a dictionary whose key is the term and values contain {offset,size, number of documents in the collection containing the term
, number of terms in collection } for the Compressed Index Binary File.

GetVocab(): returns the vocab of the corpus

DocIDtoScene(): returns the scene given the docID

QueryTermsUnCompressedBIG(): Returns the top 5 documents for words found through Dice Coefficients. Using the uncompressed Index.
QueryTermsCompressedBig(): Returns the top 5 documents for words found through Dice Coefficients. Using the compressed Index.

2) **(5%) Why might counts be misleading features for comparing different scenes? How could you fix this?**
//misleading for n gram queries
Counts could be misleading as in such a model all terms are weighted equally while assessing the relevancy on a query. The different scenes have different length and there could be a case where we want to query n-words for example “venice scene” and let's assume that venice is a rare term and scene occurs frequently. So while retrieving documents on count we might get many redundant documents who have a lot of sense but a few ‘venice’ which might not be useful for us.

We could fix n-gram query retrieval by thinking about a score which weights in less frequent words accordingly and giving them higher weightage. One such solution is idft scoring for the term

$$idft = (log(N/dft)$$

Where N is the total number of documents in the collection and dft = number of documents contained in the term.
Idft is high for a rare term and low for a frequent term.

3) **(5%)**
   a) **What is the average length of a scene?** 1204
   b) **What is the shortest scene?** Antony and Cleopatra:3.10
   c) **What is the longest play?** Richard_iii
   d) **The shortest play?** Comedy_of_errors

4) **(25%) Present your experimental results. Does the compression hypothesis hold? What conditions might change the results of the experiment, and how would you expect them to change?**

Hypothesis: Compression reduces the I/O by reducing the storage requirements for the index.
Result of our Experiment:
The compressed Index was a little faster in retrieving the terms compared to the uncompressed one by 0.316402912 seconds.

Conditions that may disprove the hypothesis:
If we are looking at a small corpus in which

1) Each document is less than 256 words.(<=255).
2) Number of Documents hence DocId is less than 256 (<=255).
3) Total number of occurrences of a term in a document is less than 256 (<=255).

In such a scenario all the values in the inverted lists can be represented in 1 byte at most. For those values = 255 when we perform vbyte encoding any term that is 255 will be represented in 2 bytes. So we are actually using more space in this case, so our I/O cost is bigger than for an uncompressed index. Also, in such a scenario we have decode first the vbyte and then the positional information which we also add to our processing(runtime) time.

From what I noticed IO was taking more time than decoding.