Department of Computer Science
The University of Melbourne
COMP90042 WEB SEARCH AND TEXT ANALYSIS (Semester 1, 2017)

Workshop exercises: Week 9

Discussion

1. Compare using a **term–document matrix** vs. an **inverted index**:
   (a) for resolving a **Boolean** query efficiently.
      - Our term–document matrix will only contain binary (0 or 1) values; consequently, the results can be obtained through bitwise operators (e.g. a conjunctive query means that we take the bitwise AND of the two lists, suitably interpretted as bit arrays or whatever). We need an operation for each document, but each operation is *really* cheap.
      - Assuming that the document indices are sorted in the inverted index, for a conjunctive query, we need to examine each element in each postings list of the query terms exactly once (you can see an example of this in the notebook). Each operation is more costly (equality test over a long unsigned integer), but there are fewer of them (assuming that all of the query terms aren't stop–words).
      - Note that we need to get a little creative for other Boolean operators on the inverted index, but it can still be done.

   (b) for resolving a **ranked** query efficiently.
      - Assuming some kind of TF–IDF vector space model using cosine similarity, using **accumulators** to keep track of each document's score:
      - For the term–document matrix, we will need to read the TDM value of **every** document for **each** query term. (Note that shortcuts — like skipping documents with a term weight of 0 — save little time here.)
      - For the inverted index, once more we only need to examine each entry in each postings list once; except that, here we incrementing the accumulator weights rather than comparing documents identifiers.

2. Say we wished to resolve a ranked query over a document collection: we want a TF-IDF model which utilises the cosine similarity in the resulting vector space. What are the advantages and disadvantages, when applying this model using the following representation (in the inverted index):
   (a) Raw term–frequencies are recorded in the index;
      - We're going to need two extra data structures: one recording the IDF of each term in the collection (let's call it I), and one recording the length of each document in the collection (let's call it L).
      - The inverted index is a list of (document identifier, weight) pairs: each weight in this case is an unsigned integer (term frequency). It turns out that these values are very amenable to **compression**, which means that this index can be stored much more compactly than the two described below.

- Querying proceeds as follows:
  - For each term t in the query: read the corresponding IDF value from I
    - For each document d in the postings list of t: calculate TF–IDF and increment the accumulator for d
  - Divide each accumulator by the corresponding document length from L
  - Sort the accumulator list (perhaps partially) to find the top k results, and return them
- In summary, the index is probably more compact, but querying is slower because we need to calculate all of those TF–IDF values at *query time* (usually involving logarithms, which are slow).

(b) TF-IDF weights are recorded in the index;

- We still need L, but I is no longer necessary, as those values are already encoded in the index.
- The inverted index is a list of (document identifier, weight) pairs: each weight in this case is a floating–point value (TF–IDF weight). This can't be compressed as effectively as the TF integers.
- Querying proceeds as follows:
  - For each term t in the query:
    - For each document d in the postings list of t: read off TF–IDF and increment the accumulator for d
  - Divide each accumulator by the corresponding document length from L
  - Sort the accumulator list (perhaps partially) to find the top k results, and return them
- In summary, the index is probably larger, but querying is faster because we don't need to calculate all of those TF–IDF values.

(c) Documents are normalised to length 1, corresponding weights of term are recorded in the index?

- Assuming that we normalised the TF–IDF values, we don't need I or L (all of the lengths are 1).
- The inverted index is a list of (document identifier, weight) pairs: each weight in this case is a floating–point value between 0 and 1. These values are basically incompressible.
- Querying proceeds as follows:
  - For each term t in the query:
    - For each document d in the postings list of t: read off TF–IDF and increment the accumulator for d
  - Sort the accumulator list (perhaps partially) to find the top k results, and return them
- In summary, the index is larger again, but querying is even faster: this time, the operations are just additions, and we don't need to even divide through by the document lengths at the end.