Department of Computer Science
The University of Melbourne
COMP90042 WEB SEARCH AND TEXT ANALYSIS (Semester 1, 2018)

Workshop exercises: Week 10

Discussion

1. Compare using a **term–document matrix** vs. an **inverted index** for resolving a ranked query efficiently.

   - Assuming Term-at-a-Time processing (not WAND) 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.
   - Algorithms such as WAND can further improve processing of the inverted index lists.

2. Using the TF-IDF vector space model, using raw term frequency $f_{t,d}$, $\log \frac{N}{f_t}$ as the inverse document frequency formulation, find the ranking for the query `apple ibm`, based on calculated over the following collection. You should use cosine similarity, but for this question, we will skip the document normalisation step (for time reasons.)

	`apple`	`ibm`	`lemon`	`sun`
D_1	4	0	1	1
D_2	5	0	5	0
D_3	2	5	0	0
D_4	1	0	1	7
D_5	0	1	3	0

   - The similarity metric is $S_{\text{TF-IDF}}(d, Q) = \sum_{t \in Q} f_{d,t} \times \log \frac{N}{f_t}$ which has to be computed for the query and each document. Where $f_{d,t}$ is the number of times term q occurs in document d and f_t is the total number of times q occurs in the collection. N is the total number of documents in the collection..
   - Now, let's consider `apple`, which occurs in 4 documents ($f_a = 5; N = 4$):
   - `apple` occurs in D_1 4 times ($f_{1,a} = 4$), so its TF-IDF weight becomes:

$$w_{1,a} \quad = \quad f_{1,a} \times \log \frac{N}{f_a} = 4 \times \log \frac{5}{4} = 0.89$$

- Performing the same computation for all documents yields with term `apple`:

$$
\begin{aligned}
w_{2,a} &= f_{2,a} \times \log \frac{N}{f_a} = 5 \times \log \frac{5}{4} = 1.12 \\
w_{3,a} &= f_{3,a} \times \log \frac{N}{f_a} = 2 \times \log \frac{5}{4} = 0.45 \\
w_{4,a} &= f_{4,a} \times \log \frac{N}{f_a} = 1 \times \log \frac{5}{4} = 0.22 \\
w_{5,a} &= f_{5,a} \times \log \frac{N}{f_a} = 0 \times \log \frac{5}{4} = 0
\end{aligned}
$$

- And then the other terms (`ibm, lemon, sun`). First we'll calculate the idf terms:

	`apple`	`ibm`	`lemon`	`sun`
idf	$\log \frac{5}{4} = 0.22$	$\log \frac{5}{2} = 0.92$	$\log \frac{5}{4} = 0.22$	$\log \frac{5}{2} = 0.92$

  And use these to produce the remaining TFxIDF scores:

	`apple`	`ibm`	`lemon`	`sun`
D_1	0.89	0	0.22	0.92
D_2	1.12	0	1.12	0
D_3	0.45	4.58	0	0
D_4	0.22	0	0.22	6.41
D_5	0	0.92	0.67	0

- The IDF component controls the "importance" of each term in the query. The term `apple` occurs in most documents so it contributes less towards identifying relevant documents. Imagine a large text collection of English documents. Most documents will contain multiple occurrences the term "the". So, the IDF adjusts the importance of terms to the overall similarity score of documents.
- On the other hand, consider the term `ibm`, which occurs in only a few documents. Thus, the magnitude of the $w_{,i}$ weights is much higher. The property that less frequent documents are generally more important is also utilised the WAND algorithm (see below) which exploits this property to avoid scoring documents which only contain terms that occur frequently.
- finding the final scores for each document is quite easy by comparison: we simply sum the TF-IDF values for the terms in the query:

$$
\begin{aligned}
D_1 &: w_{1,a} + w_{1,i} = 0.89 + 0 = 0.89 \\
D_2 &: w_{2,a} + w_{2,i} = 1.12 + 0 = 1.12 \\
D_3 &: w_{3,a} + w_{3,i} = 0.45 + 4.58 = 5.03 \\
D_4 &: w_{4,a} + w_{4,i} = 0.22 + 0 = 0.22 \\
D_5 &: w_{5,a} + w_{5,i} = 0 + 0.92 = 0.92
\end{aligned}
$$

- The most "relevant" document according to our TF-IDF metric is D_3 by a large margin because it contains a rare term ($f_i = 2$), multiple times!

- You were instructed to skip the document normalisation step. This corresponds to normalising each row vector (document) by its vector magnitude (sum of squares). I.e., The normalisation factor $W_d = \sqrt{\sum_t \left(f_{d,t} \times \log \frac{N}{f_t}\right)^2}$. For completeness, here's how you would do it:

	`apple`	`ibm`	`lemon`	`sun`	magnitude
D_1	0.89	0	0.22	0.92	1.30
D_2	1.12	0	1.12	0	1.58
D_3	0.45	4.58	0	0	4.6
D_4	0.22	0	0.22	6.41	6.43
D_5	0	0.92	0.67	0	1.14

And then divide through each row:

	`apple`	`ibm`	`lemon`	`sun`
D_1	0.69	0	0.17	0.71
D_2	0.71	0	0.70	0
D_3	0.10	0.99	0	0
D_4	0.03	0	0.03	0.999
D_5	0	0.81	0.59	0

For our query of `apple` and `ibm`, the top scoring document is still D_3 with a score of $0.1 + 0.99 = 1.09$.

3. Recall the Okapi BM25 term weighting formula:

$$w_t = \log \frac{N - f_t + 0.5}{f_t + 0.5} \times \frac{(k_1 + 1) f_{d,t}}{k_1((1-b) + b\frac{L_d}{L_{\text{avg}}}) + f_{d,t}} \times \frac{(k_3 + 1) f_{q,t}}{k_3 + f_{q,t}}$$

What are its parameters, and what do they signify? How do the components relate to TF (term frequency) and inverse document frequency (IDF)?

(a) What are its parameters, and what do they signify?

- k_1 controls the weight of the term frequencies: high values mean that the weight for a term in the query is mostly linear in the frequency of the term in a document; low values mean that the term frequencies are mostly ignored, and only their presence/absence in the documents matters.
- k_2 isn't in this model. In fact, in practice, $k_2 = k_3 = 0$ and $k_1 = 0.9$, $b = 0.4$ gives good performance. These values were determined empirically and will be need to be "tuned" for different text collections.
- k_3 controls the term frequency within the query — so that, where terms are repeated in the query, they contribute more to the document ranking. (Note that this is often set to 0, so that query terms are only binary.)
- b controls the penalty for long documents: when b is close to 0, document length is mostly excluded from the model; when b is large, short documents are strongly preferred to long documents (all else equal).
- Note that there are many different similarity metrics out there. In practice we are mainly interested in similarity metrics that can be computed efficiently while giving a reasonable relevance ranking as the results will

often be *reranked* (scored again) by a more complex neural model containing many more features such as PageRank etc.

(b) How do the components relate to TF (term frequency) and inverse document frequency (IDF)?

- As discussed above, k_1 controls the importance of the term frequencies (TF).
- Term weights (IDF) in the BM25 model can be negative, if the term appears in greater than half of the documents in the collection. This is unlikely to be a problem in many non-trivial collections, as such a common word is a stop–word, and we would simply exclude it from the collection.

4. Data compression of a **postings list** in an inverted index can help reduce space usage of the index.

(a) What is the intuition behind compression algorithms used for postings list compression? Why do they work?

- Data compression seeks to reduce space usage by removing redundancy from the data. As a general principle, we want to spend fewer bits representing objects that occur often. For example, a regular ASCII symbols always 8 bits. However, in English text symbols such as ' ' or 'e' occur much more often. Thus, when storing English text we would like to represent frequent symbols with less than 8 bits. On the other hand symbols that occur infrequently (such as 'q' or 'z') could even be represented using more than 8 bits!
- In terms of postings list compression we generally have two sequences of integers associated with each term. The ids of documents containing that term (in the range [1,N] where N is the number of documents in the collection) and the frequencies with which each term occurs in a specific document.
- The list of document identifiers is generally stored as an increasing sequence of integers often containing large numbers. Frequencies are generally small (a given word only occurs a few times in each document).
- Instead of storing the increasing list of document identifiers we instead store the *gaps* between subsequent document ids. This transforms the postings lists data into a set of integers consisting of mostly small values. Thus, we seek to create integer compression schemes which are able to store small numbers efficiently, while spending more bits on large numbers (which occur infrequently).

(b) What is **Variable Byte Compression** and how does it compress an integer?

- Variable Byte compression is an integer encoding scheme which allows storing small numbers in fewer bytes (compared to using a fixed 32/64 bits per integer).
- Conceptually it chunks up the binary representation of a number X into 7 bit chunks. Each chunk is stored in an individual byte where the top bit is used to indicate if the current byte stores a complete number or additional bytes have to processed.

- Small numbers (less than 128) can be stored in one byte only. In practice this corresponds to 95% of all postings that have to be stored in an inverted index.

(c) Determine the values of integers X and Y that were encoded as the byte sequence [52,34,147,42,197] using the Variable Byte algorithm described in the lecture slides 9/10.

- Note, the first version of the slides contained an incorrect version of the decoding algorithm (which is now fixed). The works as follows:
  i. Read byte 52 which corresponds to `00110100`.
  ii. The top bit is not set so we read the next byte $34 =$ `00100010`.
  iii. Again the top bit is not set. We read the next byte $147 =$ `10010011`.
  iv. Now the top bit is set, we have read all the bytes to decode the first number.
  v. We wrote the most significant 7 bits first so the 7 bits coming from 52 correspond to the left-most 7 bit chunk of the final number: `0110100|xxxxxxx|xxxxxxx`
  vi. Next, the 7 bits extracted from 34 are added: `0110100|0100010|xxxxxxx`
  vii. Finally, the last 7 bits are appended to the bit representation: `0110100|0100010|0010011`. Which is 856339.
  viii. The second number is decoded the same way as $42 =$ `00101010`, $197 =$ `11000101` which is `101010|1000101` $= 5445$.
- Interestingly, Vbyte codes are self-delimiting, meaning we don't have to store how many numbers there are or where they start and end.

5. Algorithms such as WAND help speed up query processing.

(a) What is the intuition behind WAND? What is the output produced by WAND?

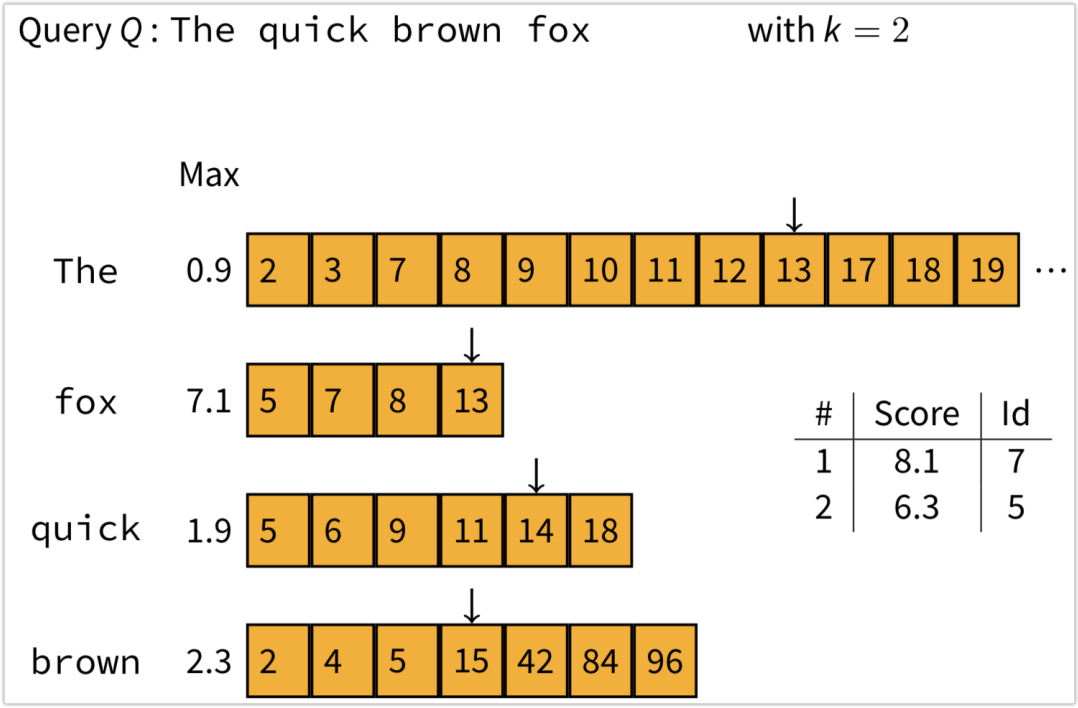
- In practice, search engines (and users!) are only interested in the first $10/100$ documents so evaluating everything is very inefficient. Even if a single evaluation of one document is fast, for a large number of documents, searching would still take seconds which is unacceptable for most users.
- WAND is a top-K query processing algorithm. Thus, it returns only the top-K highest scoring documents.
- As no complete ranking of all documents is required, we can skip evaluating documents that would/could not enter the final top-K results list.

(b) What extra information is stored for each term to allow algorithms like WAND to skip evaluating documents? How is it computed? What restriction does it place on the query process?

- For each term in the collection one additional floating point number called the *maximum contribution* is stored.
- The maximum contribution is used to overestimate the score any term of the query can have on the total score of a document. Thus, if a document score including these overestimated contributions is LESS than the lowest scoring term in our current top-K list, it can never be part of the final top-K result set. Therefore we do not have to evaluate it.

- The maximum contribution for each term is computed by computing the similarity metric for a query that contains only that term for all documents in the index (at construction time!). For example, to compute the maximum contribution for the term "house" we run the query "house" and compute the score for all documents. The highest score is then stored as the maximum contribution the term "house" can have to the score of ANY document. Note that this introduces a dependency between the similarity metric (e.g. BM25) and the index which is stored on disk. Thus, we cannot use a different metric at query time.
- Additionally, we would want the underlying postings list representation to support the `GEQ` operation which allows us to efficiently skip over documents without decompressing large parts of the index.

(c) Assume Document 13 has just been evaluated. In the setting below, what is the next document that will be evaluated?



- 13 is evaluated and might enter the top-K list.
- After 13 is evaluated the list for the term `fox` is finished. Thus, it can no longer contribute to the score of any document.
- To decide which is the next *potential* document to be evaluated, we (1) sort the list by increasing current id and (2) sum up the maximum contributions of the lists (top to bottom) until the sum is larger than the smallest score in our top-K list.
- However, the maximum scores for the remaining lists (fox is finished) is 0.9, 1.9 and $2.3 = 0.9 + 1.9 + 2.3 = 5.1$. So, even if 13 does not get added to the top-K list, the minimum score in the heap is already larger than the score of ANY remaining document we have not evaluated. Thus, the WAND algorithm finishes and we return the top-K result list.

Programming

1. Issue some queries using the small IR engine given in the iPython notebook `WSTA_N15_information_retrieval`. Read (some of) the documents that are returned: confirm that the keyword(s) is/are present, and judge whether you think these documents are relevant to your query.
2. Work on the project! `:-)`

Catch-up

- What is an **information retrieval engine**?
- What does it mean for a document to be **relevant** to a query?
- What is a **vector space model**? How can we find **similarity** in a vector space?

Get ahead

- What effect do the various preprocessing regimes have on the efficiency (time) and effectiveness (relevant results) of querying with the system (note: not building the index)? In particular, consider:
  1. Stemming
  2. Stopping
  3. Tokenisation (e.g. of non-alphabetic tokens)