School of Computing and Information Systems
The University of Melbourne
COMP90042
WEB SEARCH AND TEXT ANALYSIS (Semester 1, 2019)

Sample solutions for discussion exercises: Week 4

Discussion

1. Discuss the process of static **inverted index construction** and how it can be used to perform in incremental index construction.

   Solution:

   - Space efficient inverted index construction splits a static document collection into blocks.
   - For each block an individual inverted index is constructed (space efficiently)
   - At the end, the inverted indexes for each block are merged to create a single large inverted index for the whole collection.
   - In the incremental setting we are working with a similar setting. We have a collection that is already indexed (somewhere stored on disk or in memory). As new documents arrive we form a block and build an inverted index for those.
   - At query time we have to query all of the inverted indexes at the same time and merge the results

2. Why is a **logarithmic** index layout useful? What are the disadvantages of such an index structure?

   Solution:

   - Lets say we would index $N = 10$ billion postings of the course of a year.
   - However, at each time we only store $n = 1$ million postings in RAM.
   - If the index becomes too large we store it to disk by merging it with a larger inverted index containing the postings we have stored so far.
   - Thus, we would want to merge the small index containing n postings with the larger index.
   - At the start of the year we start with having nothing on disk and n things in memory.
   - Merging the small index to disk would just mean storing n postings to disk.
   - The next time we have n postings in memory, we also have n postings on disk. Thus merging them requires writing out $2n$ postings.
   - The next time we have n postings in memory, we would have to write $3n$, $4n$, $5n$, …, N postings as we do the merging throughout the year until at the end we have an index on disk containing roughly $N = 10$ billion postings.
   - However, because we throughout the year we performed lots of merge operations which amounts to $N^2/n \approx n + 2n + 3n + 4n \ldots N$ IO operations. This is very expensive.

- Instead we keep a logarithmic number of indexes of different size. For example, our first level stores at most n postings. The next level stores at most $2n$ postings, the next level stores at most $4n$ postings and so on. If the a level fills up we only merge with the next higher level. So to incorporate n new postings, we merge with the $2n$ level. If that fills up we merge that with the $4n$ level etc. As the size at the different level increases, merging becomes cheaper as less IOs are required in total.
- The main disadvantage of the logarithmic layout is that, at query time a logarithmic number of indexes have to be queried. This introduces additional complexity and runtime overhead as the results of the different indexes have to be merged.

3. Based on the following top-6 retrieval results from a collection of 100 documents, and the accompanying binary relevance judgements

doc	score	relevance
a	0.4	0
b	1.2	0
c	2.2	1
d	0.5	1
e	0.1	1
f	0.8	0

compute the following evaluation metrics:

(a) precision@3

The first step is to form the vector of relevance judgments, ranked by document score

$$\langle c, b, f, d, a, e \rangle = \langle 1, 0, 0, 1, 0, 1 \rangle$$

Now taking the first 3 items, $\langle 1, 0, 0 \rangle$ (c,b,f), one is correct, from 3 total. So we have $P@3 = \frac{1}{3}$.

(b) average precision (do you need to make any assumptions about the document collection?); and

AP is normalised by the number of relevant documents in the collection. We weren't told this in the question, all we know is that of the retrieved documents 3 are relevant (c,d,e). Based on this, we could have anywhere between 3 and 97 relevant documents. Here we'll be optimisic, and use 3:

$$AP = \frac{1}{3} \times (\frac{1}{1} + \frac{2}{4} + \frac{3}{6}) = \frac{1}{3} \times 2 = \frac{2}{3}$$

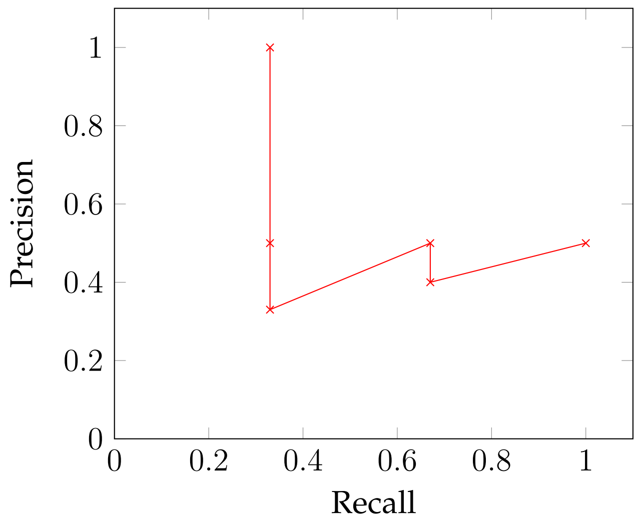
Under the most pessimistic view, the AP is much lower, 2%. Witness the massive difference between the two results, which is worrying as most often we don't know the full relevant set. Why? Too many documents (millions or billions) which we would have to look at to determine if they are relevant.

(c) rank-biased precision (RBP), with $p = 0.5$

$$\begin{aligned} RBP &= (1-p) \times \sum_i r_i p^{i-1} && (1)\\ &= (1-0.5) \times \left(0.5^0 + 0.5^3 + 0.5^5\right) && (2)\\ &= \frac{1}{2} \times \left(1 + \frac{1}{8} + \frac{1}{32}\right) && (3)\\ &= \frac{1}{2} \times \frac{1}{32} \times (32 + 4 + 1) && (4)\\ &= \frac{37}{64} && (5) \end{aligned}$$

RBP with $p = 0.5$ means the user has a persistence of $p = 0.5$ to move from the fist to the second document and p^2 to the third etc. Higher values of p represent a more patient user model. For impatient users only the top documents are important and the RBP score reflects that. Note for the RBP formula that the factor $(1-p)$ originates from the *expected* number of documents that will be evaluated. This is a normalization factor and for a given p is just a constant.

(d) plot the precision-recall graph, where you plot (precision, recall) point for the top k documents, $k = 1, 2, \ldots 6$.



Note this figure could be drawn in a more disconnected way, as it's questionable connecting the points because you can't return a fractional document.

(e) what are the strengths and weaknesses of the methods above for evaluating IR systems?

precision@k is very easy to evaluate and easily understood, however it doesn't differentiate by rank within k, and doesn't incorporate any adjustment for the size of the relevant set;

avg precision is less easily understood, but adds differentiation by rank, and adjusts for the size of the relevant set, however this requires knowing the relevant set (we typically don't know this); Averaging over multiple queries (mean average precision) with different number of relevant results also distorts the metric and it is very unintuitive of what the MAP score actually means.

rank biased precision has a similar formulation to AP, but it a little more intuitive. It includes differentiation by rank, and avoids the need to know the relevant set beyond the top ranked documents (as the contribution to RBP of lower ranked documents diminishes quickly, and can be upper bounded). A small issues is the need to know the free parameter, "p", which may differ between users and queries.

4. How can a retrieval method be learned using supervised machine learning methods? Consider how to frame the learning problem, what data will be required for supervision, and what features are likely to be useful.

   Learning to rank deals with the problem of taking the top k results from a heuristic IR system, and reordering these to improve the user acceptability of the results. This can be framed as a learning in several ways, with the simplest being a "pointwise" objective whereby a learning algorithm is trained to give a high score to documents that are relevant, and a low score to irrelevant documents. This could be training a binary classifier, or a multi-class classifier (or a regression system or ordinal regression). The supervision for this are relevance judgments over the top k results, or, more commonly, user click data from query logs, or other related extrinsic data such as whether the user reformulates the query, makes a purchase, etc. There are a plethora of features that can be useful, deriving from the user behaviour, the document url and its text, and the query.