Introduction to

Information Retrieval

Lecture 7: Scoring and results assembly

Recap: tf-idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$\mathrm{w}_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \times \log_{10}(N / \mathrm{df}_t)$$

- Best known weighting scheme in information retrieval
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection

Recap: Queries as vectors

- Key idea 1: Do the same for queries: represent them as vectors in the space
- Key idea 2: Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors

Recap: cosine(query,document)

$$\cos(\vec{q},\vec{d}) = \frac{\overbrace{\vec{q} \bullet \vec{d}}^{\text{Dot product}}}{|\vec{q}||\vec{d}|} = \underbrace{\frac{\vec{q}}{|\vec{q}|} \bullet \frac{\vec{d}}{|\vec{d}|}}_{\text{Unit vectors}} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

$\cos(\vec{q},\vec{d})$ is the cosine similarity of $\vec{q}$ and $\vec{d}$ … or, equivalently, the cosine of the angle between $\vec{q}$ and $\vec{d}$.

This lecture

- Speeding up vector space ranking
- Putting together a complete search system
  - Will require learning about a number of miscellaneous topics and heuristics

Question: Why don't we just use the query processing methods for Boolean queries?

Term-at-a-time

Computing cosine scores

CosineScore(q)

float $Scores[N] = 0$
float $Length[N]$
for each query term t
do calculate $w_{t,q}$ and fetch postings list for t
    for each pair(d, $\mathrm{tf}_{t,d}$) in postings list
    do $Scores[d] += w_{t,d} \times w_{t,q}$
Read the array *Length*
for each d
do $Scores[d] = Scores[d]/Length[d]$
return Top K components of $Scores[]$

Efficient cosine ranking

- Find the *K* docs in the collection "nearest" to the query $\Rightarrow$ *K* largest query-doc cosines.
- Efficient ranking:
  - Computing a single cosine efficiently.
  - Choosing the *K* largest cosine values efficiently.
    - Can we do this without computing all *N* cosines?

Efficient cosine ranking

- What we're doing in effect: solving the *K*-nearest neighbor problem for a query vector
- In general, we do not know how to do this efficiently for high-dimensional spaces
- But it is solvable for short queries, and standard indexes support this well

Special case – unweighted queries

- No weighting on query terms
  - Assume each query term occurs only once
- Then for ranking, don't need to normalize query vector
  - Slight simplification of algorithm from Lecture 6

Faster cosine: unweighted query

```
FastCosineScore(q)
 float Scores[N] = 0
 for each d
 do Initialize Length[d] to the length of doc d
 for each query term t
 do calculate w_{t,q} and fetch postings list for t
     for each pair(d, tf_{t,d}) in postings list
     do add wf_{t,d} to Scores[d]
 Read the array Length[d]
 for each d
 do Divide Scores[d] by Length[d]
 return Top K components of Scores[]
```

Figure 7.1 A faster algorithm for vector space scores.

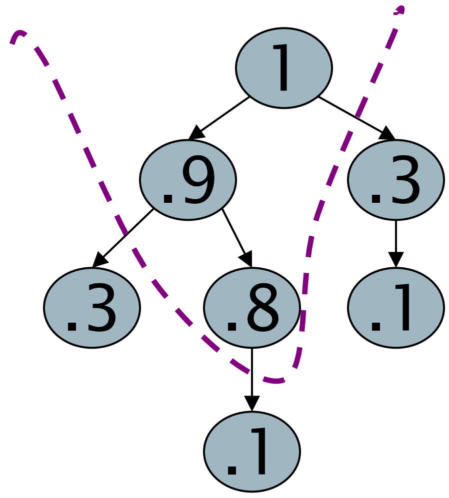
Computing the *K* largest cosines: selection vs. sorting

- Typically we want to retrieve the top *K* docs (in the cosine ranking for the query)
  - not to totally order all docs in the collection
- Can we pick off docs with *K* highest cosines?
- Let *n* of docs with nonzero cosines
  - We seek the *K* best of these *n*

http://en.wikipedia.org/wiki/Binary_heap

Use heap for selecting top *K* /1

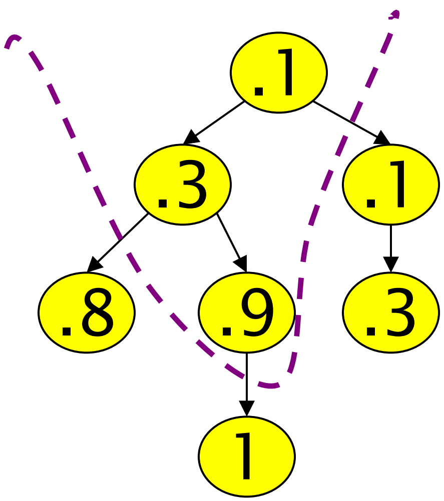
- Max-heap:
  - Binary tree in which each node’s value > the values of children
- Takes 2*n* operations to construct, then each of *K* “winners” read off in 2log *n* steps
- Total time is O(n + K*log(n)); space complexityis O(n)
- For *n*=1M, *K*=100, this is about 10% of the cost of sorting.



http://en.wikipedia.org/wiki/Binary_heap

Use heap for selecting top *K* /2

- What about using a min-heap?
- Use the min-heap to maintain the top k scores so far.
- For each new score, s, scanned:
  - H.push (s)
  - H.pop()
- Total time is O(n*log(k) + k*log(k)); space complexityis O(k)



http://en.wikipedia.org/wiki/Quickselect

Quick Select

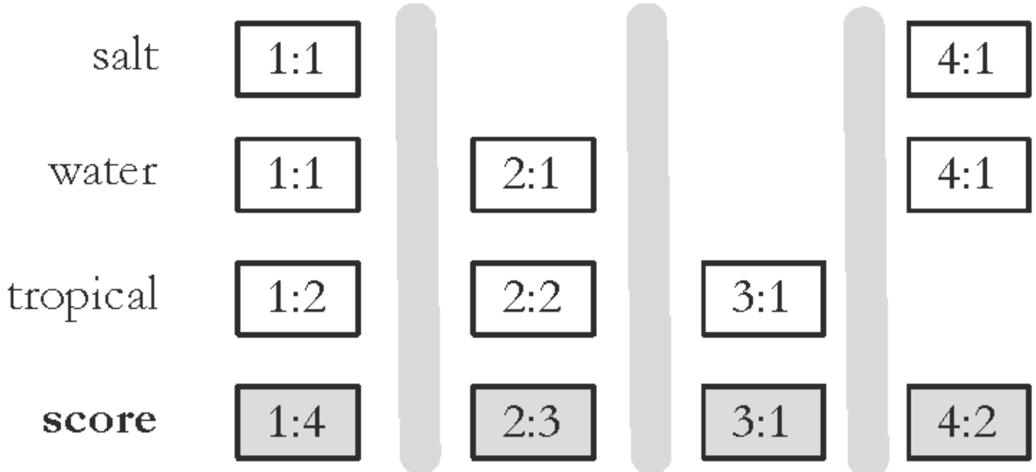
- QuickSelect is similar to QuickSort to find the top-K elements from an array
  - Takes O(n) time (in expectation)
- Sorting the top-K items takes O(K*log(K)) time
- Total time is O(n + K*log(K))

[CMS09].begin

Query Processing

- Document-at-a-time
  - Calculates complete scores for documents by processing all term lists, one document at a time
- Term-at-a-time
  - Accumulates scores for documents by processing term lists one at a time
- Both approaches have optimization techniques that significantly reduce time required to generate scores
  - Distinguish between safe and heuristic optimizations

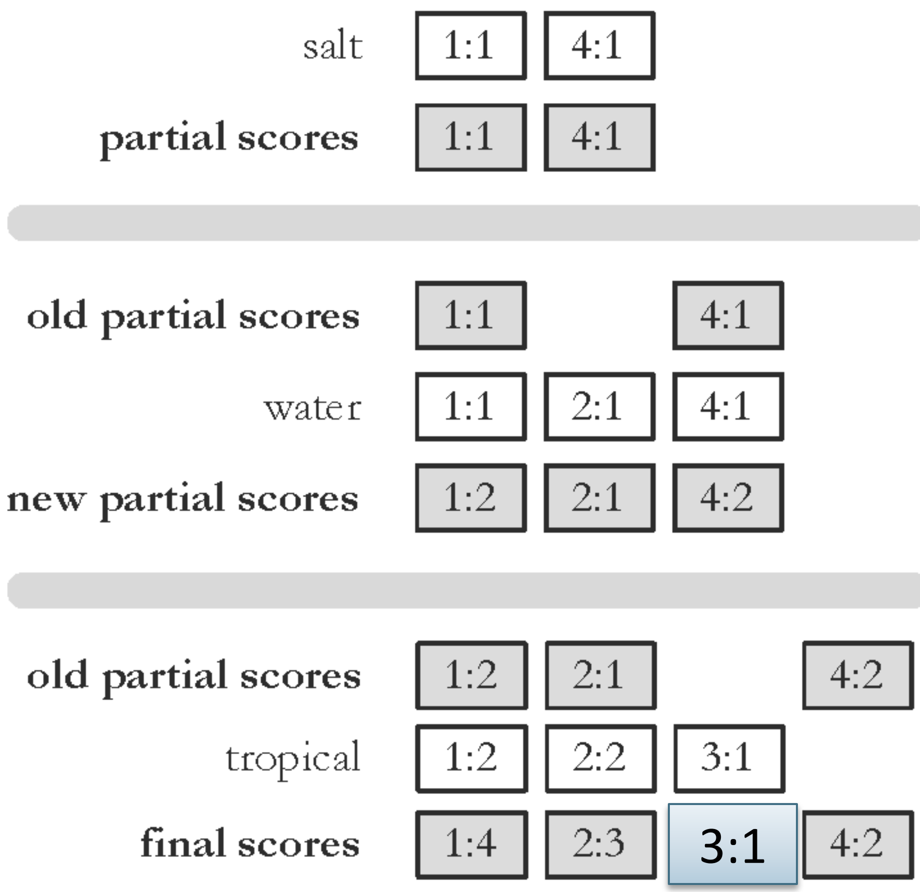
Document-At-A-Time



Document-At-A-Time

procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    $L \leftarrow$ Array()
    $R \leftarrow$ PriorityQueue(k)
    for all terms w_i in Q **do**
        $l_i \leftarrow$ InvertedList(w_i, I)
        L.add(l_i)
    end for
    for all documents $d \in I$ **do**
        for all inverted lists l_i in L **do**
            if l_i points to d **then**
                $s_D \leftarrow s_D + g_i(Q) f_i(l_i)$ ▷ Update the document score
                l_i.movePastDocument(d)
            end if
        end for
        R.add(s_D, D)
    end for
    return the top k results from R
end procedure

Term-At-A-Time



Term-At-A-Time

procedure $\text{TermAtATimeRetrieval}(Q,\ I,\ f,\ g\ k)$
    $A \leftarrow \text{HashTable}()$
    $L \leftarrow \text{Array}()$
    $R \leftarrow \text{PriorityQueue}(k)$
    for all terms w_i in Q **do**
        $l_i \leftarrow \text{InvertedList}(w_i,\ I)$
        $L.\text{add}(\ l_i\ )$
    end for
    for all lists $l_i \in L$ **do**
        while l_i is not finished **do**
            $d \leftarrow l_i.\text{getCurrentDocument}()$
            $A_d \leftarrow A_d + g_i(Q) f(l_i)$
            $l_i.\text{moveToNextDocument}()$
        end while
    end for
    for all accumulators A_d in A **do**
        $s_D \leftarrow A_d$ ▷ Accumulator contains the document score
        $R.\text{add}(\ s_D, D\ )$
    end for
    return the top k results from R
end procedure

// accumulators

// A_d contains partial score

Optimization Techniques

- Term-at-a-time uses more memory for accumulators, but accesses disk more efficiently
- Two classes of optimization
  - Read less data from inverted lists
    - e.g., skip lists
    - better for simple feature functions
  - Calculate scores for fewer documents
    - e.g., conjunctive processing
    - better for complex feature functions

Conjunctive Processing

- Requires the result document containing all the query terms (i.e., conjunctive Boolean queries)
  - More efficient
  - Can also be more effective for short queries
  - Default for many search engines
- Can be combined with both DAAT and TAAT (see pseudocodes next)

Conjunctive Term-at-a-Time

procedure TermAtATimeRetrieval(Q, I, f, g, k)
  $A \leftarrow$ Map()
  $L \leftarrow$ Array()
  $R \leftarrow$ PriorityQueue(k)
  for all terms w_i in Q **do**
    $l_i \leftarrow$ InvertedList(w_i, I)
    L.add(l_i)
  end for
  for all lists $l_i \in L$ **do**
    $d_0 \leftarrow -1$
    while l_i is not finished **do**
      if $i = 0$ **then**
        $d \leftarrow l_i$.getCurrentDocument()
        $A_d \leftarrow A_d + g_i(Q)f(l_i)$
        l_i.moveToNextDocument()
      else
        $d \leftarrow l_i$.getCurrentDocument()
        $d' \leftarrow A$.getNextAccumulator(d)
        A.removeAccumulatorsBetween(d_0,d')
        if $d = d'$ **then**
          $A_d \leftarrow A_d + g_i(Q)f(l_i)$
          l_i.moveToNextDocument()
        else
          $l_i$skipForwardToDocument(d')
        end if
        $d_0 \leftarrow d'$
      end if
    end while
  end for
  for all accumulators A_d in A **do**
    $s_d \leftarrow A_d$ ▷ Accumulator contains the document scor
    R.add(s_d, d)
  end for
  return the top k results from R
end procedure

Fig. 5.20. A term-at-a-time retrieval algorithm with conjunctive processing

Conjunctive Document-at-a-Time

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    d ← −1
    while all lists in L are not finished do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() > d then
                d ← l_i.getCurrentDocument()
            end if
        end for
        for all inverted lists l_i in L do
            l_i.skipForwardToDocument(d)
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q)f_i(l_i)            ▷ Update the document score
                l_i.movePastDocument( d )
            else
                d ← −1
                break
            end if
        end for
        if d > −1 then R.add( s_d, d )
        end if
    end while
    return the top k results from R
end procedure
```

Fig. 5.21. A document-at-a-time retrieval algorithm with conjunctive processing

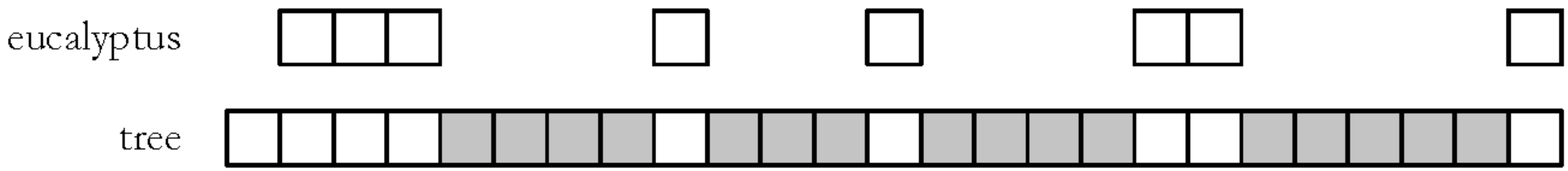
Threshold Methods

- Threshold methods use number of top-ranked documents needed (k) to optimize query processing
  - for most applications, k is small
- For any query, there is a *minimum score* that each document needs to reach before it can be shown to the user
  - score of the kth-highest scoring document
  - gives *threshold* τ
  - optimization methods estimate τ' to ignore documents

Threshold Methods

- For document-at-a-time processing, use score of lowest-ranked document so far for τ'
  - for term-at-a-time, have to use k_{th}-largest score in the accumulator table
- *MaxScore* method compares the maximum score that remaining documents could have to τ'
  - *safe* optimization in that ranking will be the same without optimization

MaxScore Example



- Compute max term scores, μ_t, of each list and sort them in decreasing order (fixed during query processing)
- Assume k =3, τ' is lowest score of the current top-k documents
- If $\mu_{tree} < \tau'$ ➔ any doc that scores higher than τ' must contains *at least one of* the first two keywords (aka *required term set*)
  - Use postings lists of required term set to "drive" the query processing
  - Will only check **some of** the white postings in the list of "tree" to compute score ➔ at least all gray postings are skipped.

[CMS09].end

Other Approaches

- Early termination of query processing
  - ignore high-frequency word lists in term-at-a-time
  - ignore documents at end of lists in doc-at-a-time
  - *unsafe* optimization
- List ordering
  - order inverted lists by quality metric (e.g., PageRank) or by partial score
  - makes unsafe (and fast) optimizations more likely to produce good documents

Bottlenecks

- Primary computational bottleneck in scoring: cosine computation
- Can we avoid all this computation?
- Yes, but may sometimes get it wrong
  - a doc *not* in the top *K* may creep into the list of *K* output docs
  - Is this such a bad thing?

Cosine similarity is only a proxy

- Justifications
  - User has a task and a query formulation
  - Cosine matches docs to query
  - Thus cosine is anyway a proxy for user happiness
- Approximate query processing
  - If we get a list of *K* docs “close” to the top *K* by cosine measure, should be ok

Generic approach

- Find a set A of *contenders*, with $K < |A| << N$
  - A does not necessarily contain the top K, but has many docs from among the top K
  - Return the top K docs in A
- Think of A as pruning non-contenders
- The same approach is also used for other (non-cosine) scoring functions
- Will look at several schemes following this approach

Index elimination

- Basic algorithm FastCosineScore of Fig 7.1 only considers docs containing at least one query term
- Take this further:
  - Only consider high-idf query terms
  - Only consider docs containing many query terms

High-idf query terms only

- For a query such as *catcher in the rye*
- Only accumulate scores from *catcher* and *rye*
- Intuition: *in* and *the* contribute little to the scores and so don't alter rank-ordering much
- Benefit:
  - Postings of low-idf terms have many docs → these (many) docs get eliminated from set *A* of contenders

Docs containing many query terms

- Any doc with at least one query term is a candidate for the top *K* output list
- For multi-term queries, only compute scores for docs containing several of the query terms
  - Say, at least 3 out of 4
  - Imposes a "soft conjunction" on queries seen on web search engines (early Google)
- Easy to implement in postings traversal

Can generalize to WAND method (safe)

3 of 4 query terms

Antony	3	4	8	16	32	64	128	
Brutus	2	4	8	16	32	64	128	
Caesar	1	2	3	5	8	13	21	34
Calpurnia	13	16	32					

Scores only computed for docs 8, 16 and 32.

Champion lists

- Precompute for each dictionary term *t*, the *r* docs of highest weight in *t*'s postings
  - Call this the champion list for *t*
  - (aka fancy list or top docs for *t*)
- Note that *r* has to be chosen at index build time
  - Thus, it's possible that $r < K$
- At query time, only compute scores for docs in A = $\cup_{t \in Q}$ ChampionList(t)
  - Pick the *K* top-scoring docs from amongst these

Inspired by "fancy lists" of Google:
http://infolab.stanford.edu/~backrub/google.html

Exercises

- How do Champion Lists relate to Index Elimination? Can they be used together?
- How can Champion Lists be implemented in an inverted index?
  - Note that the champion list has nothing to do with small docIDs