Advanced topics in Computer Science

Jiaheng Lu
Department of Computer Science
Renmin University of China
www.jiahenglu.net
Recap of the last lecture

- Parametric and field searches
  - Zones in documents
- Scoring documents: zone weighting
  - Index support for scoring
- $tf \times idf$ and vector spaces
This lecture

- Vector space scoring
- Efficiency considerations
  - Inexact top-K retrieval
- Components of a search system
Efficient cosine ranking

- Find the $K$ docs in the corpus “nearest” to the query $\implies K$ 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?
Recall basic scoring algorithm

\textbf{COSINESCORE}(q)

1. \texttt{float Scores}[N] = 0
2. Initialize \texttt{Length}[N]
3. \texttt{for} each query term \texttt{t}
4. \texttt{do}
5. \hspace{1em} calculate \(w_{t,q}\) and fetch inverted list for \texttt{t}
6. \hspace{1em} \texttt{for} each pair\((d, tf_{t,d})\) in inverted list
7. \hspace{2em} \texttt{do}
8. \hspace{3em} add \(w_{f_{t,d}} \times w_{t,q}\) to \texttt{Scores}[d]
9. \hspace{1em} \texttt{Read the array \texttt{Length}[d]}
10. \hspace{1em} \texttt{for} each \texttt{d}
11. \hspace{2em} \texttt{do} Divide \texttt{Scores}[d] by \texttt{Length}[d]
12. \hspace{1em} \texttt{return} Top \textit{K} components of \texttt{Scores}[]
Detail: inner loop of score

- Can use inverted index
- Traverse postings for all $t$ concurrently

```
for each $t$ do
    calculate $w_{t,q}$ and fetch inverted list for $t$
    for each pair $(d, tf_{t,d})$ in inverted list do
        add $wf_{t,d} \times w_{t,q}$ to $Scores[d]$
        Read the array $Length[d]$
    for each $d$ do
        Divide $Scores[d]$ by $Length[d]$
```
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 totally order all docs in the corpus
  - can we pick off docs with $K$ highest cosines?
Use heap for selecting top $k$

- Binary tree in which each node’s value > values of children
- Takes $2N$ operations to construct, then each of $k \log N$ “winners” read off in $2\log N$ steps.
- For $N=1M$, $K=100$, this is about 10% of the cost of sorting.
Inexact top-$K$ retrieval
Efficient cosine ranking

- What we’re doing in effect: solving the $K$-nearest neighbors problem for a query vector $z$
- In general, do not know how to do this efficiently for high-dimensional spaces
  - Short of computing all $N$ cosines
- Yes, but may occasionally get an answer wrong
  - a doc *not* in the top $K$ may creep into the answer
  - (and docs in the top $K$ get omitted)
Is inexact top-$K$ retrieval a bad thing?

- Cosine scores are a proxy for user happiness
- Find $K$ docs with cosine scores close to top $K$
- Hope to save in computation
- For user, no material impact?
Generic approach

- Find a set $A$ of documents that are *contenders*, where $K << |A| << N$.
  - $A$ does not necessarily contain the $K$ top-scoring documents for the query, but is likely to have many with scores near those of the top $K$.
- Return the $K$ top-scoring documents in $A$.
- Will see many ideas following this approach.
Index elimination

- Traverse only postings of high-idf terms
- The rest don’t contribute much to scores
- Added benefit: low idf terms have long postings
  - Now ignored

- Only compute scores for docs containing all (or most) query terms
- Danger: may end up with fewer than $K$ candidates
Champion lists (Fancy lists ...) 

- Precompute, for each term $t$, the set of the $r$ docs with the highest weights for $t$
  - Call this set of $r$ docs the *champion list* for term $t$
  - For tf-idf weighting, these would be the $r$ docs with the highest tf values for $t$
- Given a query $q$ take the union of the champion lists for the terms in $q$ to create set $A$
- Restrict cosine computation to only the docs in $A$
- Note: $r$ is fixed at index construction whereas
Static quality scores

- In many search engines, we have available a quality measure $g(d)$ for each document $d$
  - $g(d) \in [0,1]$ is query-independent and thus static
  - E.g., for news stories on the web: $g(d)$ may be derived from the number of favorable reviews
- Order postings by $g(d)$
- Still can perform all postings merge operations as before! Don’t need docID ordering.
  - Suppose
    \[
    \text{net-score}(q,d) = g(d) + \frac{\tilde{V}(q) \cdot \tilde{V}(d)}{|\tilde{V}(q)||\tilde{V}(d)|}.
    \]
Postings ordering by $g(d)$

- Can compute complete cosine scores
  - Then get exact top-$K$
- Can perform inexact top top-$K$ in one of several ways:
  - Stop postings traversal for term $t$ when $g()$ drops below a threshold
  - For each $t$, maintain *global champion list* of the $r$ documents with the highest values of $g(d) + (\text{tf-idf})_{t,d}$

(Now treat as champion lists before.)
High and low champions

- For each term maintain two disjoint postings:
  - *High*, with the champions
  - *Low*, with the rest
- In query processing, try to get $K$ documents from the *High* champions
- If we fail to get $K$, then “fall back” to the *Low* postings
Impact ordering

- Thus far, all postings always contain docs in same consistent order
  - We’ve used docIDs or $g(d)$ for this ordering
- Now will consider a scheme where the postings are *not* all in the same ordering
- But will still obtain a benefit for fast but inexact cosine computation
Recall basic scoring algorithm

\[ \text{COSINE SCORE}(q) \]

1. \( \text{float Scores}[N] = 0 \)
2. \( \text{Initialize Length}[N] \)
3. \( \text{for each query term } t \)
4. \( \text{do} \)
5. \( \text{calculate } w_{t,q} \text{ and fetch inverted list for } t \)
6. \( \text{for each pair } (d, tf_{t,d}) \text{ in inverted list} \)
7. \( \text{do} \)
8. \( \text{add } wf_{t,d} \times w_{t,q} \text{ to } Scores[d] \)
9. \( \text{Read the array Length}[d] \)
10. \( \text{for each } d \)
11. \( \text{do } \text{Divide } Scores[d] \text{ by } Length[d] \)
12. \( \text{return Top K components of } Scores[] \)
Frequency-ordered postings

- Order docs $d$ in the postings of term $t$ by decreasing order of $tf_{t,d}$
  - Thus varying orders for different postings
  - Must compute scores one term at a time
- When traversing postings for query term $t$, stop after considering a prefix of the postings
- Consider query terms in decreasing order of $idf$

Use various criteria for early termination
Cluster pruning: preprocessing

- Pick $\sqrt{N}$ docs at random: call these leaders
- For each other doc, pre-compute nearest leader
  - Docs attached to a leader: its followers;
  - Likely: each leader has $\sim \sqrt{N}$ followers.
Cluster pruning: query processing

- Process a query as follows:
  - Given query $Q$, find its nearest leader $L$.
  - Seek $K$ nearest docs from among $L$’s followers.
Visualization

Leader

Follower

Query
Why use random sampling

- Fast
- Leaders reflect data distribution
General variants

- Have each follower attached to $a=3$ (say) nearest leaders.
- From query, find $b=4$ (say) nearest leaders and their followers.
- Can recur on leader/follower construction.
Exercises

- To find the nearest leader in step 1, how many cosine computations do we do?
  - Why did we have $\sqrt{N}$ in the first place?
- What is the effect of the constants $a, b$ on the previous slide?
- Devise an example where this is *likely to fail* – i.e., we miss one of the $k$ nearest docs.
  - *Likely* under random sampling.
Putting together a system

- We have most of the pieces for a fairly modern search system; two further notions
- Tiered Indexes
- General machine-learned scoring
Tiered indexes

- Generalization of high and low champions
- Try answering query using a fraction of the postings for each term
- If we fail to get enough matching docs
  - Fall back to the next fraction of the postings …
  - Then the next, and so on
Query-term proximity

- Given a query with two or more query terms, \( t_1, t_2, \ldots, t_k \)
- Let \( \omega = \) width of the smallest window in doc \( d \) containing all \( t_1, t_2, \ldots, t_k \)
  - E.g., if the doc is *The quality of mercy is not strained*, for the query *strained mercy* \( \omega = 4 \)
- Intuition: docs for which \( \omega \) is small should score higher
  - *How much smaller?*
General issue

- We have many features on which to base our scoring
  - Cosine score, zone scores, proximity $\omega$, Length(d), $g(d)$, ...
- How do we combine all these into a scoring function?
- Machine-learned scoring
Machine Learned Scoring, part II

- Given
  - A test corpus
  - A suite of test queries
  - A set of relevance judgments
- Functional form of target scoring function
- Learn a set of weights such that relevance judgments matched
Example

- Suppose we deem the score to be linear in the cosine score $\alpha$ and proximity window $\omega$

\[ \text{Score}(\alpha, \omega) = a\alpha + b\omega + c, \]

- Here the functional form is the class of linear functions, but we could choose any we want, e.g.
  - Quadratics in $\alpha$ and $\omega$
  - Decision trees: “If $\alpha > 0.25$ and $\omega < k + 2$, or if $\alpha > 0.15$ and $\omega < k + 20$”, then the doc is relevant, else not
Training set

- Given examples, each of which is a docID, query and relevance judgment
- We can compute $\alpha$ and $\omega$ for each example
- Quantizing Relevant/Non-relevant as 1/0 as before

<table>
<thead>
<tr>
<th>Example</th>
<th>DocID</th>
<th>Query</th>
<th>Cosine score</th>
<th>$\omega$</th>
<th>Judgment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Phi_1$</td>
<td>37</td>
<td>linux operating system</td>
<td>0.032</td>
<td>3</td>
<td>Relevant</td>
</tr>
<tr>
<td>$\Phi_2$</td>
<td>37</td>
<td>penguin logo</td>
<td>0.02</td>
<td>4</td>
<td>Non-relevant</td>
</tr>
<tr>
<td>$\Phi_3$</td>
<td>238</td>
<td>operating system</td>
<td>0.043</td>
<td>2</td>
<td>Relevant</td>
</tr>
<tr>
<td>$\Phi_4$</td>
<td>238</td>
<td>runtime environment</td>
<td>0.004</td>
<td>2</td>
<td>Non-relevant</td>
</tr>
<tr>
<td>$\Phi_5$</td>
<td>1741</td>
<td>kernel layer</td>
<td>0.022</td>
<td>3</td>
<td>Relevant</td>
</tr>
<tr>
<td>$\Phi_6$</td>
<td>2094</td>
<td>device driver</td>
<td>0.03</td>
<td>2</td>
<td>Relevant</td>
</tr>
<tr>
<td>$\Phi_7$</td>
<td>3191</td>
<td>device driver</td>
<td>0.027</td>
<td>5</td>
<td>Non-relevant</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

...
Learning and its use

- We “learn” the constants $a, b, c$ in

$$\text{Score}(\alpha, \omega) = a \alpha + b \omega + c,$$

- Given any other query and doc
  - Compute the score function as above
  - Use scores to rank
  - If need be, threshold scores by some cutoff

We will consider this last version.
View examples in the $\alpha$- $\omega$ plane

- Score function is a plane “above” the slide.
- Thresholding score $\Rightarrow$ a line on this plane.
- This line has a projection on the $\alpha$- $\omega$ plane.
Thresholding linear scores

- Thus, using a threshold to determine if a doc is Relevant or not ⇔ linear separator of training examples
- Cf. Support Vector Machines
  - Which of many possibly lines to choose
  - When training set not linearly separable, how to choose line …
- Key – can extend approach uniformly to many more “features” (like $\alpha$ and $\omega$)
Putting it all together

Documents → Parsing linguistics → Indexers

User query → Free text query parser → Spell correction → Scoring and ranking → Results page

Zone and field indexes → Inexact top K retrieval → Tiered inverted positional index → k-gram

Indexes

Scoring parameters
MLR

Heap

Training set