Introduction to

Information Retrieval

CS276
Information Retrieval and Web Search
Chris Manning, Pandu Nayak and Prabhakar Raghavan
Efficient scoring
Today’s focus

- **Retrieval** – get docs matching query from inverted index

- **Scoring+ranking**
  - Assign a score to each doc
  - Pick $K$ highest scoring docs

- Our emphasis today will be on doing each of these efficiently, rather than on the quality of the ranking
  - We’ll consider the impact of the scoring function – whether it’s simple, complicated etc.
  - In turn, some “efficiency tricks” will impact the ranking quality
Background

- Score computation is a large (10s of %) fraction of the CPU work on a query
  - Generally, we have a tight budget on latency (say, 250ms)
  - CPU provisioning doesn’t permit exhaustively scoring every document on every query
- Today we’ll look at ways of cutting CPU usage for scoring, without compromising the quality of results (much)
- Basic idea: avoid scoring docs that won’t make it into the top $K$
Recap: Queries as vectors

- We have a weight for each term in each doc
- **Key idea 1**: Do the same for queries: represent them as vectors in the space
- **Key idea 2**: Rank documents according to their cosine similarity to the query in this space
- Vector space scoring is
  - Entirely query dependent
  - Additive on term contributions – no conditionals etc.
  - Context insensitive (no interactions between query terms)
- We’ll later look at scoring that’s not as simple …
TAAT vs DAAT techniques

- TAAT = “Term At A Time”
  - Scores for all docs computed concurrently, one query term at a time
- DAAT = “Document At A Time”
  - Total score for each doc (incl all query terms) computed, before proceeding to the next
- Each has implications for how the retrieval index is structured and stored
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?
Safe vs non-safe ranking

- The terminology “safe ranking” is used for methods that guarantee that the $K$ docs returned are the $K$ absolute highest scoring documents
  - (Not necessarily just under cosine similarity)
- Is it ok to be non-safe?
- If it is – then how do we ensure we don’t get too far from the safe solution?
  - How do we measure if we are far?
SAFE RANKING
We first focus on safe ranking

- Thus when we output the top $K$ docs, we have a proof that these are indeed the top $K$
- Does this imply we always have to compute all $N$ cosines?
  - We’ll look at pruning methods
- Do we have to sort the resulting cosine scores? (No)
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 $J =$ number of docs with nonzero cosines
  - We seek the $K$ best of these $J$
Use heap for selecting top $K$

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

- An instance of DAAT scoring
- Basic idea reminiscent of branch and bound
  - We maintain a running threshold score – e.g., the $K^{th}$ highest score computed so far
  - We prune away all docs whose cosine scores are guaranteed to be below the threshold
  - We compute exact cosine scores for only the un-pruned docs

Index structure for WAND

- Postings ordered by docID
- Assume a special iterator on the postings of the form “go to the first docID greater than X”
- Typical state: we have a “finger” at some docID in the postings of each query term
  - Each finger moves only to the right, to larger docIDs
- Invariant – all docIDs lower than any finger have already been processed, meaning
  - These docIDs are either pruned away or
  - Their cosine scores have been computed
Upper bounds

- At all times for each query term $t$, we maintain an upper bound $UB_t$ on the score contribution of any doc to the right of the finger
  - Max (over docs remaining in $t$’s postings) of $w_t(doc)$

$UB_t = w_t(38)$

As finger moves right, $UB$ drops
Pivoting

- **Query:** catcher in the rye
- **Let’s say the current finger positions are as below**

```
catcher 273
rye 304
in 589
the 762
```

Threshold = 6.8

- $UB_{catcher} = 2.3$
- $UB_{rye} = 1.8$
- $UB_{in} = 3.3$
- $UB_{the} = 4.3$
Prune docs that have no hope

- Terms sorted in order of finger positions
- Move fingers to 589 or right

**catcher** 273 Hopeless docs

**rye** 304 Hopeless docs

**in** 589

**the** 762

Threshold = 6.8

\[ UB_{catcher} = 2.3 \]

\[ UB_{rye} = 1.8 \]

\[ UB_{in} = 3.3 \]

\[ UB_{the} = 4.3 \]

Update UB’s
Compute 589’s score if need be

- If 589 is present in enough postings, compute its full cosine score – else some fingers to right of 589
- Pivot again ...

- catcher
- rye
- in
- the
WAND summary

- In tests, WAND leads to a 90+% reduction in score computation
  - Better gains on longer queries
- Nothing we did was specific to cosine ranking
  - We need scoring to be *additive* by term
- WAND and variants give us *safe ranking*
  - Possible to devise “careless” variants that are a bit faster but not safe (see summary in Ding+Suel 2011)
  - Ideas combine some of the non-safe scoring we consider next
NON SAFE RANKING

We’ll speak of cosine scores, but most of these ideas are general and a recap of the Coursera video
Non-safe (cosine) ranking

- Return $K$ docs whose cosine similarities to the query are high
  - Relative to the safe top $K$
  - Reminiscent of normalization in NDCG
- Can we prune more aggressively?
- 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

- User has a task and a query formulation
- Cosine matches docs to query
- Thus cosine is anyway a proxy for user happiness
- If we get a list of $K$ docs “close” to the top $K$ by cosine measure, should be ok
- All this is true for just about any scoring function
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*
  - Unlike WAND, pruning here can be *lossy*

- The same approach is also used for other (non-cosine) scoring functions

- Will look at several schemes following this approach

- Often $A$ may not be explicitly spelled out a priori
Index elimination

- Basic cosine computation algorithm 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 (DAAT)

- 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
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 the champion list of some query term
  - Pick the \( K \) top-scoring docs from amongst these
High and low lists

- For each term, we maintain two postings lists called *high* and *low*
  - Think of *high* as the champion list
- **When traversing postings on a query, only traverse *high* lists first**
  - If we get more than $K$ docs, select the top $K$ and stop
  - Else proceed to get docs from the *low* lists
- A means for segmenting index into two tiers
Tiered indexes

- Break postings up into a hierarchy of lists
  - Most important
  - ...
  - Least important
- Inverted index thus broken up into tiers of decreasing importance
- At query time use top tier unless it fails to yield $K$ docs
  - If so drop to lower tiers
  - Common practice in web search engines
Example tiered index

Tier 1
- auto → Doc2
- best
- car → Doc1 → Doc3
- insurance → Doc2 → Doc3

Tier 2
- auto
- best → Doc1 → Doc3
- car
- insurance

Tier 3
- auto → Doc1
- best
- car → Doc2
- insurance
RECAP OF SOME FINAL SCORING IDEAS
Document dependent scoring

- Sometimes we’ll have scoring functions that don’t add up term-wise scores
- We’ll look at two instances here, but industry practice is rife with these
  - Static document goodness measures
  - Term proximity
Static quality scores

- We want top-ranking documents to be both *relevant* and *authoritative*.
- *Relevance* is being modeled by cosine scores.
- *Authority* is typically a query-independent property of a document.
- Examples of authority signals:
  - Wikipedia among websites
  - Articles in certain newspapers
  - A paper with many citations
  - Many bitly’s, likes or social referrals marks
  - (Pagerank)
Modeling authority

- Assign to each document a *query-independent quality score* in [0,1] to each document $d$
  - Denote this by $g(d)$
- Thus, a quantity like the number of citations is scaled into [0,1]
Net score

- Consider a simple total score combining cosine relevance and authority

\[
\text{net-score}(q,d) = g(d) + \cosine(q,d)
\]
  - Can use some other linear combination
  - Indeed, any function of the two “signals” of user happiness – more later

- Now we seek the top \( K \) docs by net score
Top K by net score – fast methods

- First idea: Order all postings by $g(d)$
- **Key:** this is a common ordering for all postings
- Thus, can concurrently traverse query terms’ postings for
  - Postings intersection
  - Cosine (or other) score computation
Why order postings by $g(d)$?

- Under $g(d)$-ordering, top-scoring docs likely to appear early in postings traversal
- In time-bound applications (say, we have to return whatever search results we can in 50 ms), this allows us to stop postings traversal early
  - Short of computing scores for all docs in postings
Champion lists in $g(d)$-ordering

- Can combine champion lists with $g(d)$-ordering
- Maintain for each term a champion list of the $r$ docs with highest $g(d) + \text{tf-idf}_{td}$
- Seek top-$K$ results from only the docs in these champion lists
- Combine with other heuristics we’ve seen ...
Different idea – Query term proximity

- **Free text queries**: just a set of terms typed into the query box – common on the web
- **Users prefer docs in which query terms occur within close proximity of each other**
- Let $w$ be the smallest window in a doc containing all query terms, e.g.,
- For the query *strained mercy* the smallest window in the doc *The quality of mercy is not strained* is 4 (words)
- Would like scoring function to take this into account – how?
Scoring factors

- The ideas we’ve seen are far from exhaustive
- But they give some of the principal components in a typical scoring function
  - They reflect some intuition of how users phrase queries, and what they expect in return
- Scoring goes beyond adding up numbers
  - E.g., if we get too few hits – how should we increase recall on the fly?
  - If it’s an obvious “nav query” how do we cut recall?
Non-additive scoring

- Free text query from user may in fact spawn one or more queries to the indexes, e.g., query *rising interest rates*
  - Run the query as a phrase query
  - If <\(K\) docs contain the phrase *rising interest rates*, run the two phrase queries *rising interest* and *interest rates*
  - If we still have <\(K\) docs, run the vector space query *rising interest rates*
  - Rank matching docs by vector space scoring
- This sequence is issued by a query handler