Relevance Scoring
CS 635

Soumen Chakrabarti

(+ many slides from MRS course)

Ranked retrieval

- Thus far, our queries have all been Boolean.
  - Documents either match or don't.
- Works for expert users with precise understanding of their needs and the collection.
  - Also good for applications: Applications can easily consume 1000s of results.
- Not good for the majority of users.
  - Most users incapable of writing Boolean queries (or they are, but they think it's too much work).
  - Most users don't want to wade through 1000s of results.
    - This is particularly true of web search.

The need for relevance ranking

- Boolean search returns a *set* of docs without any scores or ranking
- Average Web query is 2–3 words long
- Bits of entropy in query ≈ 10 which ‘addresses’ 1024 items
- Meanwhile corpus has > 10 G docs
- Typical query results in > 10 M hits

Boolean search problem: feast or famine

- Boolean queries often result in either too few (=0) or too many (1000s) results.
- Query 1: “windows dlink 650” → 200,000 hits
- Query 2: “windows dlink 650 no card found” → 0 hits
- It takes a lot of skill to come up with a query that produces a manageable number of hits.
  - AND gives too few; OR gives too many

Ranked retrieval models

- Rather than an unordered set of documents satisfying a query expression, in ranked retrieval, the system returns an ordered list of the (top) documents in the collection for a query (response choice)
- Rather than a query language of operators and expressions, the user's query is just one or more words in a human language (query 'language' choice)
- In principle, there are two separate choices here, but in practice, ranked retrieval has normally been associated with free text queries and vice versa

	Query style	Response style
	(w1 OR w2) near/4 w3	Document set without scores
~1996 through ~2010	'Telegraphic': best sound headset 2024	Ranked 1d list (with diversity?)
~2008 through now	Natural: What are the headsets to buy in 2024 that have the best sound?	Clusters, callouts to tables and KG, direct answers

Scoring as the basis of ranked retrieval

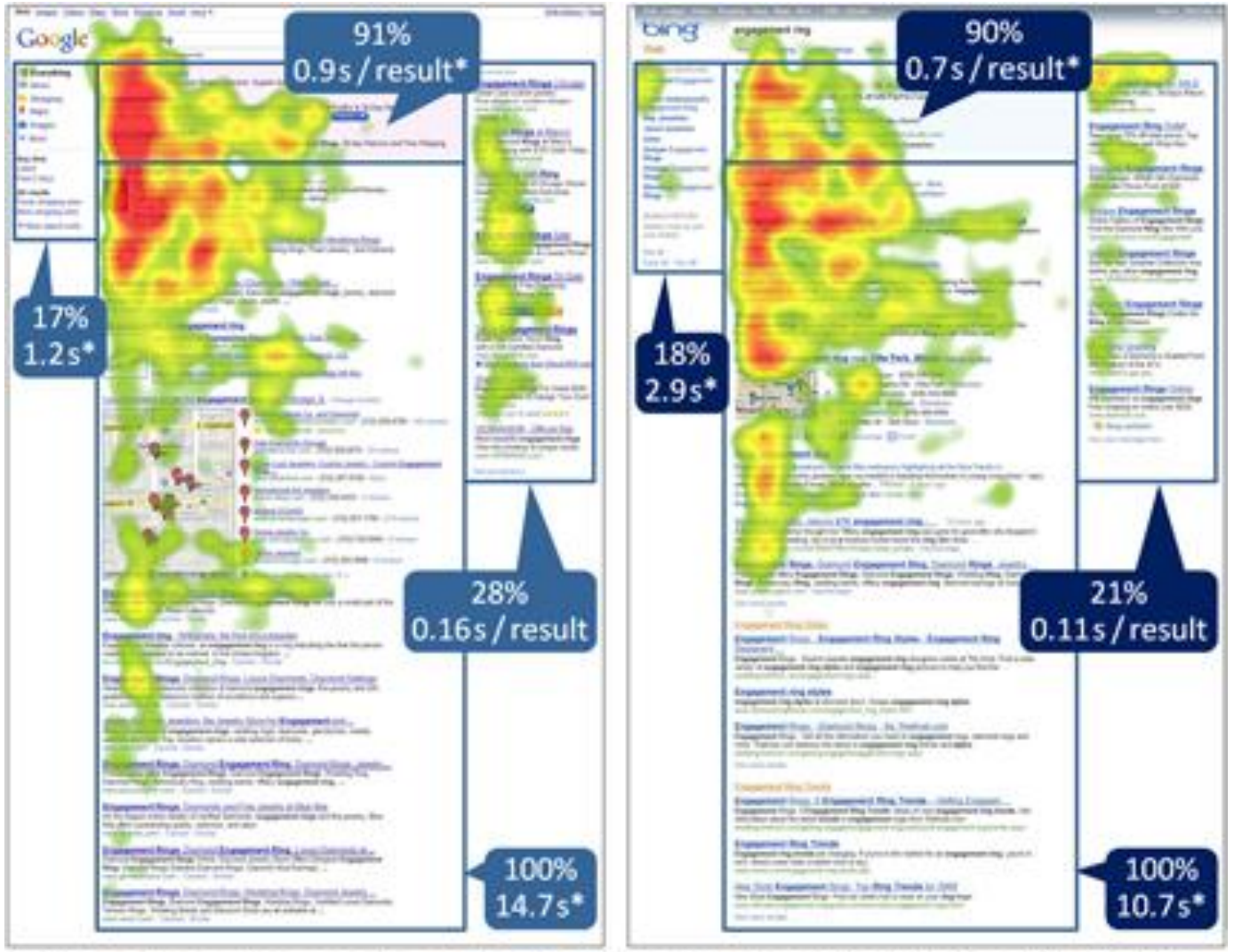
- We wish to return in order the documents most likely to be useful to the searcher
- How can we rank-order the documents in the collection with respect to a query?
- Assign a score – say in [0, 1] – to each document
- This score measures how well document and query “match”
- Query categories: navigational, informational
- Cumulative satisfaction with rank

Advantage of ranked retrieval

- Feast or famine: not a problem in ranked retrieval
- When a system produces a ranked result set, large result sets are not an issue
  - We just show the top k (≈ 10) results
  - We don't overwhelm the user
  - Premise: the ranking algorithm 'works'
- How do users react to ranked responses from search engines?

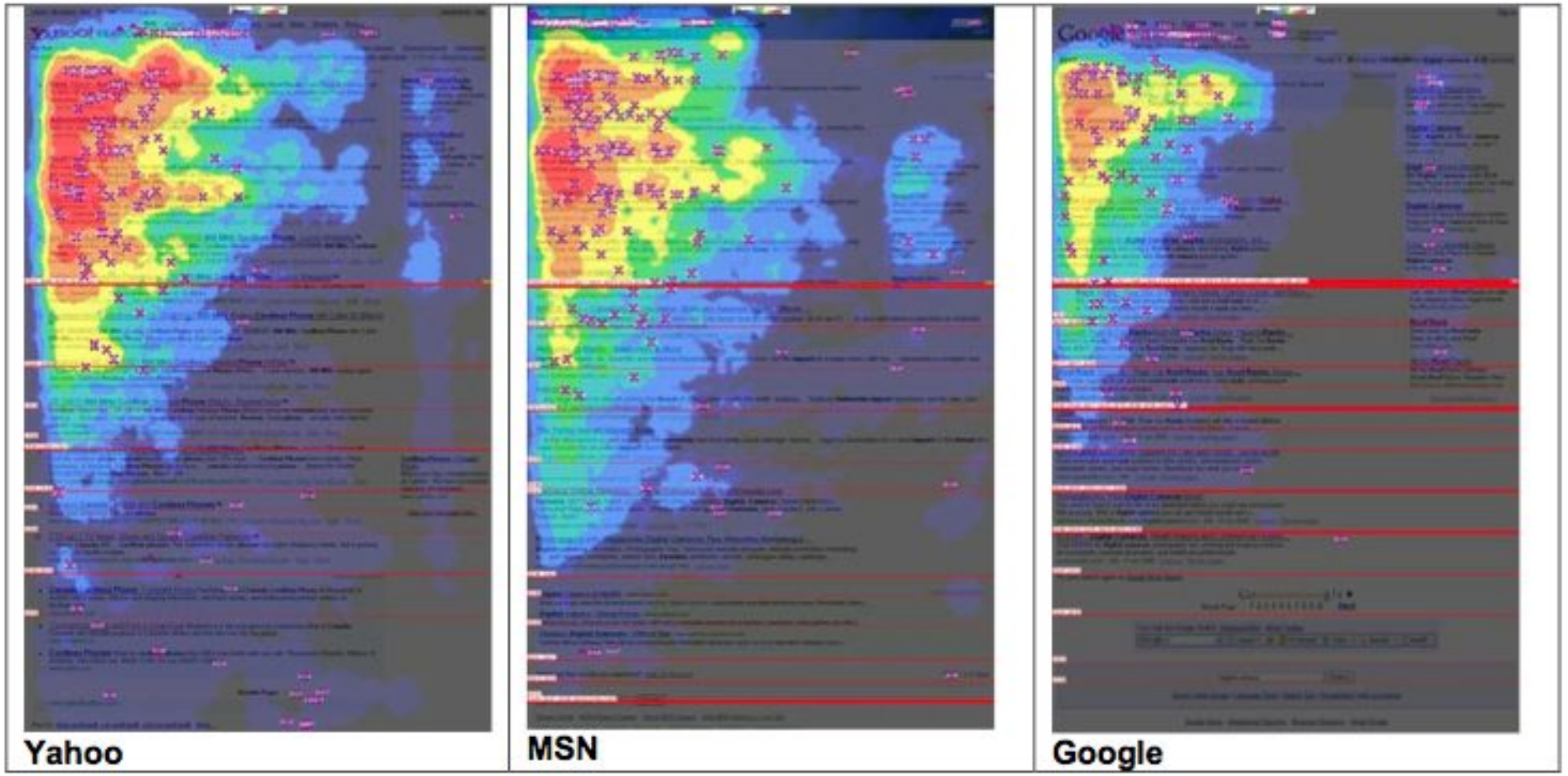
Eye tracking study on search results ~2007

Google Bing



Heatmaps showing the aggregate gaze time of all 24 participants on Google (left) and Bing (right) for one of the transactional tasks. The red color indicates areas that received the most total gaze time (4.5 seconds and above). Each callout includes the percentage of participants who looked at the area and the time (in seconds) they spent looking there. The numerical data are an average across all four tasks. Asterisks indicate values that were significantly different between Google and Bing at alpha = .1.

Yahoo, MSN, Google ~2015



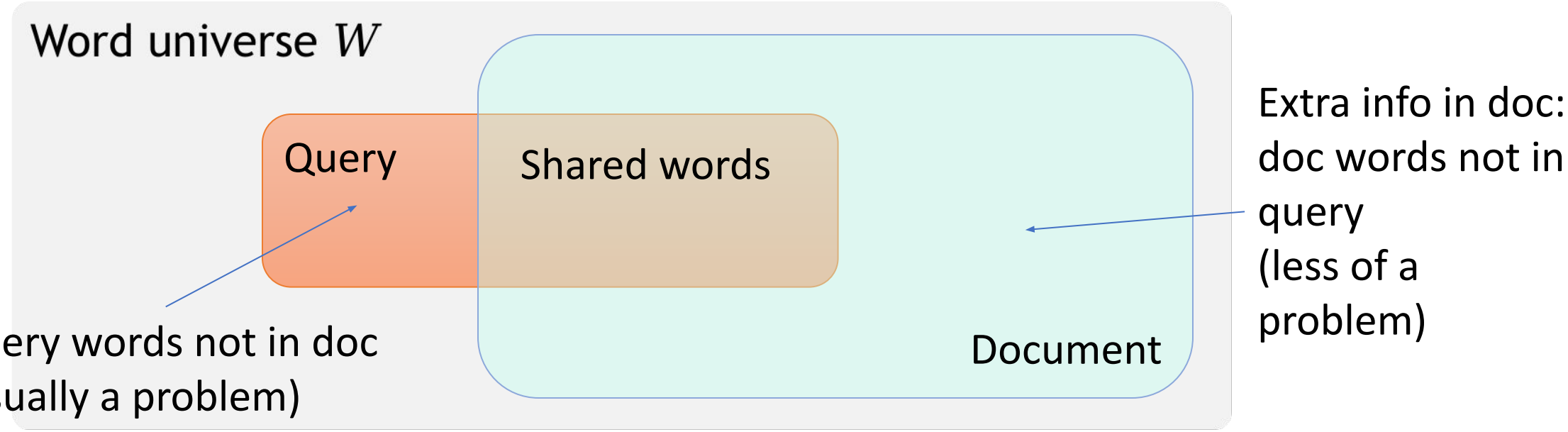
https://www.forbes.com/sites/roberthof/2015/03/03/how-do-you-google-new-eye-tracking-study-reveals-huge-changes/

Eye tracking summary

- (Cautionary tale: users are programmed by interface)
- Overall, users spend a lot of time gazing at “organic” results
- Mostly scanning down from rank 1 in search of search satisfaction, possibly ending in good and bad abandonment
- Spending more time inspecting top ranks
- There are very sophisticated models of summary inspection, skip, click, satisfaction, backoff, etc.
- For now, the clear first principle is:
  present results most relevant first
- How to measure relevance? Any axioms?

Factors in scoring relevance

- Assumption: only words shared between query and document decide relevance
- How common in the corpus are the shared words? (discriminative power) How often are the words repeated in the doc?
- How rare are query words missing in doc?
- Length of the candidate document (not always appropriate)



Other important factors (later)

- Proximity between query word match positions in the document
- Rarity/discriminativeness of query words not found in the candidate document
- Synonymy, polysemy, relations between words
- Redundancy between candidate docs
- History of clickthroughs
- PageRank/popularity of page/site

The "vector space model"

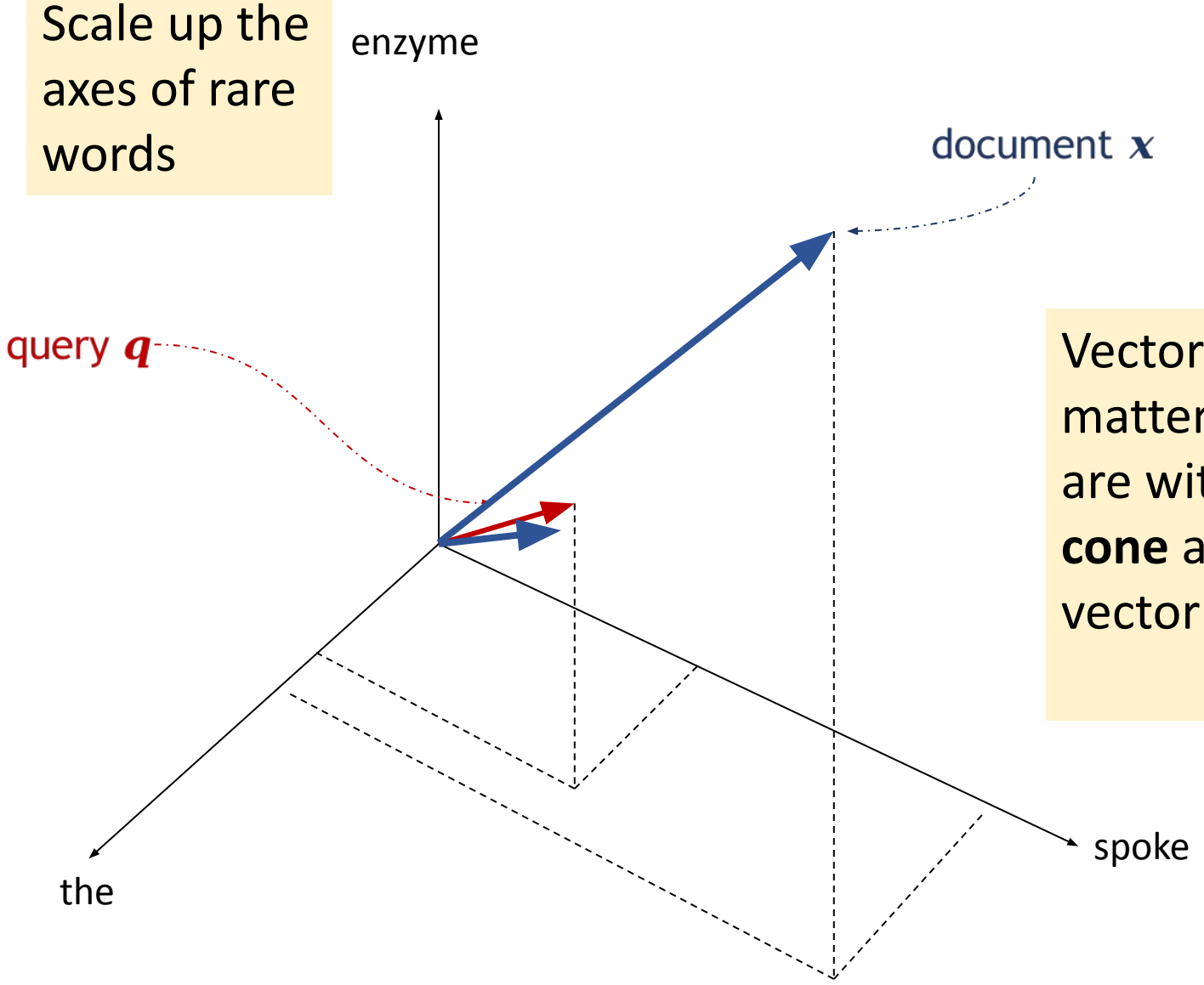
- Let $\boldsymbol{x} \in \mathbb{R}^W$ denote a doc vector
- One axis per word w in vocab $[W] = \{1, \ldots, W\}$
- Query vector $\boldsymbol{q} \in \mathbb{R}^W$ in same space
- Boolean: $x[w] = [\![w \in \text{doc } x]\!]$
  - $\boldsymbol{q} \cdot \boldsymbol{x}$ = number of shared words= $|q \cap x|$
  - For *ranking* docs, q is fixed, so no need to account for $|q \setminus x|$ explicitly
  - But we may want to normalize by doc 'length': $\frac{|q \cap x|}{|x|}$
- Word counts (bag): $x[w]$ = number of times word w occurs in doc x
  - Here doc length normalization is more critical: A doc copied five times over is not more relevant

TF-IDF representation

- Raw word count may be over-reactive
  - 'Hiroshima' is rare in a general corpus
  - But if you see it k times in a doc, you expect to see it $(k+1)$-st time
  - Marginal surprise decreases rapidly with k
  - 'Bursty' word counts should be squashed
- Not all word matches are equally informative of the relevance of doc x given query q
  - Fraction of docs in corpus where word w occurs (at least once), i.e., word rarity

Vector space visualization

Scale up the axes of rare words



Scale down the axes of frequent words

Vector lengths do not matter; relevant docs are within a narrow **cone** around query vector

Quick review of entropy

- Random variable X can take values $x_1, \ldots, x_N$
- Multinomial distribution $p_1, \ldots, p_N$
  - $p_n \geq 0 \ \forall n; \sum_n p_n = 1$ … the "unit simplex"
- Entropy ('uncertainty') $H(p) = -\sum_n p_n \log p_n$
  - Greatest when all $p_n = 1/N$
- If some info about random variable Y is revealed, the entropy of X cannot increase
- Conditional distribution $p(y|x)$
- Conditional entropy $H(y|x) = -\sum_y p(y|x) \log p(y|x)$
- Reduction in entropy $H(y) - H(y|x)$ tells us how correlated X and Y are
- Mutual information

$$I(X,Y) = H(Y) - H(Y|X) = H(X) - H(X|Y) = I(Y,X)$$

Inverse document frequency

- Assume $p(d|t) = {}^{1}/_{|\{d \in \mathcal{D}, d \ni t\}|}$

e doc corpus

- If query specifies term t as present, and this is the only info we have, then (assume) all docs containing t are equally likely

D is a random variable taking values from the set of doc IDs

- $H(D|t) = -\sum_d p(d|t) \log p(d|t) = -\log \frac{1}{|\{d \in \mathcal{D}, d \ni t\}|}$
- Rewrite as $\log \frac{|\{d \in \mathcal{D}, d \ni t\}|}{|\mathcal{D}|} + \log |\mathcal{D}|$, which is $-\text{idf}(t) + \log |\mathcal{D}|$
- Where we define $\text{idf}(t) = \log \frac{|\mathcal{D}|}{|\{d \in \mathcal{D}, d \ni t\}|}$
- A squashed form of term rarity, defined as $|\mathcal{D}|/|\{d \in \mathcal{D}, d \ni t\}|$

TFIDF, significance

- Meanwhile, unconditional entropy $H(D) = -\log \frac{1}{|\mathcal{D}|} = \log |\mathcal{D}|$
- Drop in entropy by knowing t is present is

  $H(D) - H(D|t) = \log |\mathcal{D}| + \log \frac{1}{|\{d \in \mathcal{D}, d \ni t\}|} = \text{idf(t)}$
- Summing over all terms t, we get corpus-global mutual information

  $$I(D,T) = \sum_t p(t)[H(D) - H(D|t)] = \sum_t p(t)\ \text{idf}(t)$$
- Using $p(t) = \sum_{d \in \mathcal{D}} p(d)\ p(t|d)$ we get

  $I(D,T) = \frac{1}{|\mathcal{D}|} \sum_{d,t} \frac{n(d,t)}{n(d)} \text{idf}(t)$
- Natural explanation of TFIDF representation
- In practice, even TF benefits from squashing

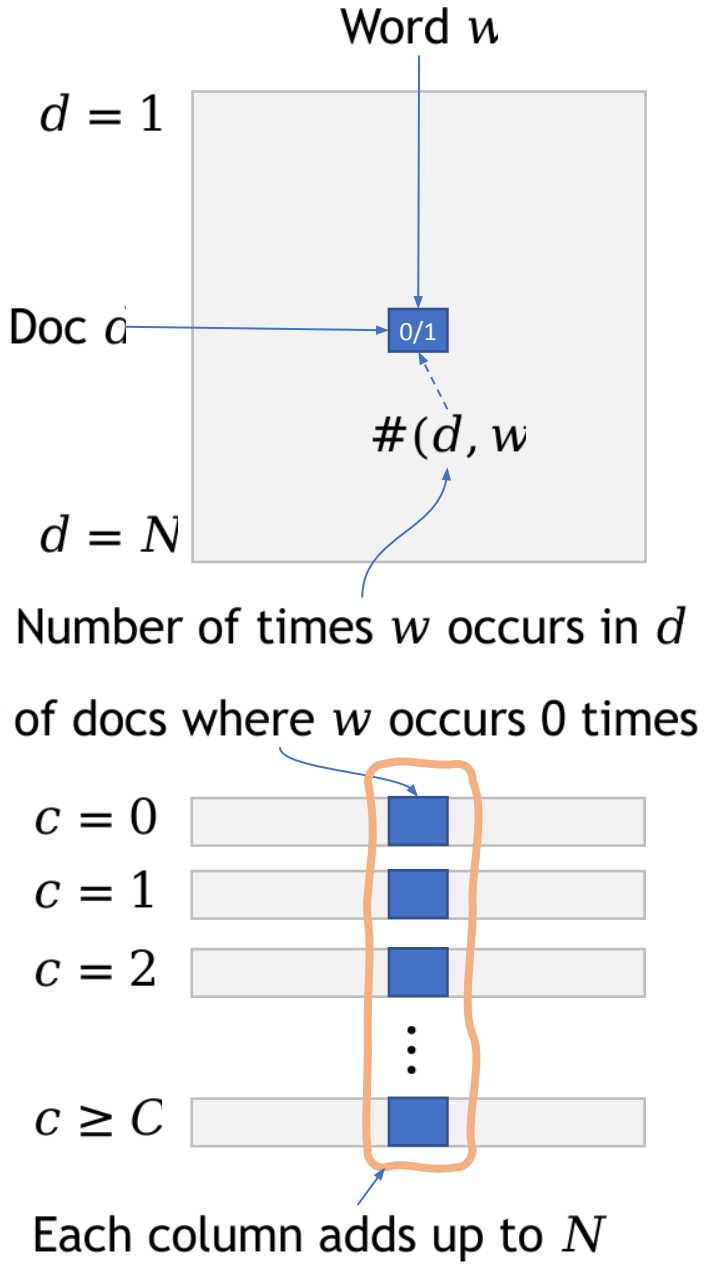
Unsquashed form of term frequency

A theory of term count squashing (TF)

- From raw input matrix $\#(d, w)$ build an aggregated summary matrix $n(w, c)$
- Let $n(w, c)$ be the number of docs where w occurs c times, with $\sum_{0 \le c \le C} n(w, c) = N$
- Model each column w using a multinomial distribution $\Pr(c|\phi_w)$, with $\sum_c \Pr(c|\phi_w) = 1$
- Φ represents model parameters over all w
- Assume columns (words) are independent
- Maximize log likelihood of observed matrix:

$$\max_{\Phi} \log \prod_{w=1}^{W} \prod_{c=0}^{C} \Pr(c|\Phi)^{n(w,c)}$$

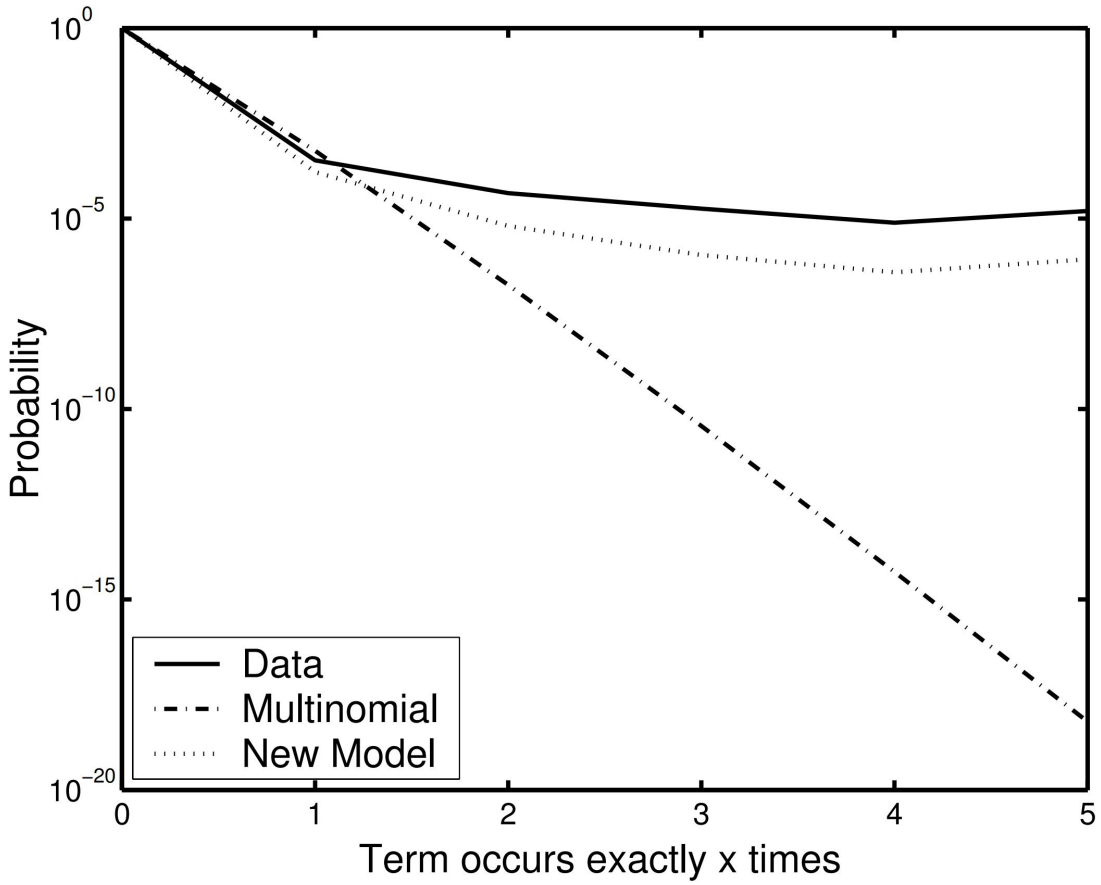
- Next: propose a parametric form for $\Pr(c|\Phi)$

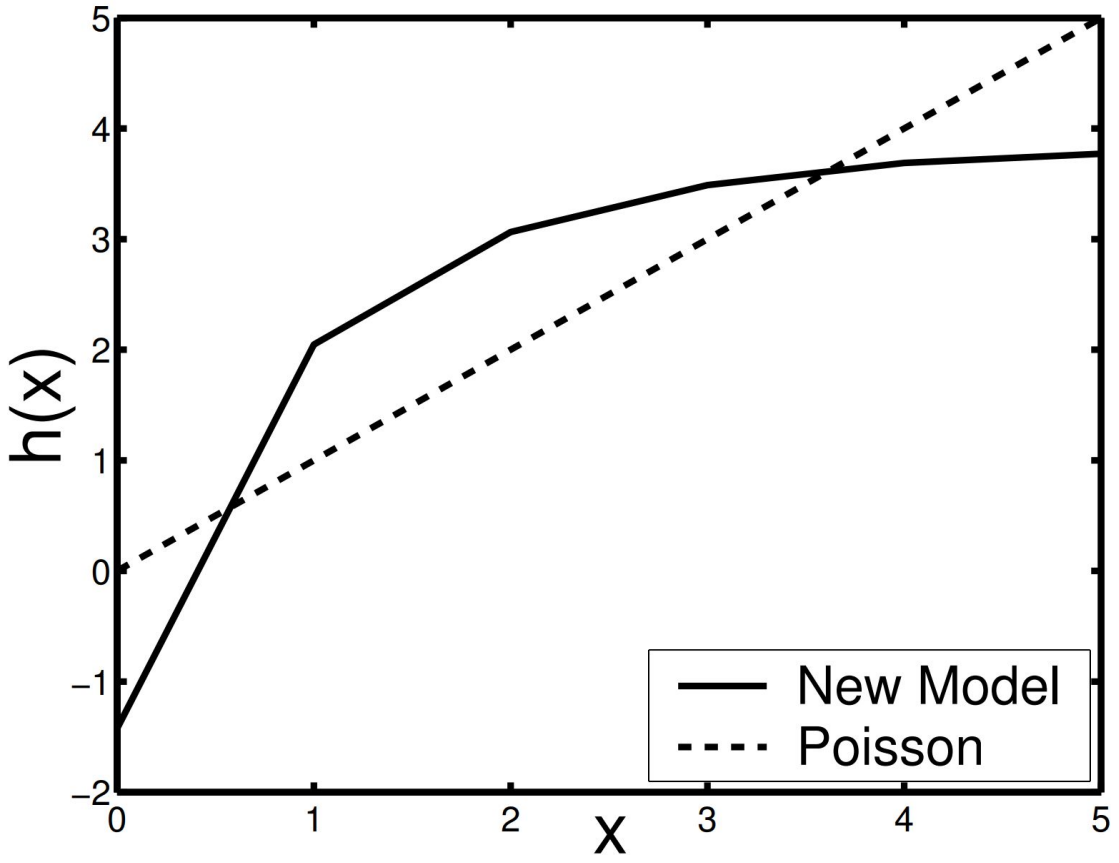


$\Pr(c|\Phi)$ from exponential family

- $\Phi = \{\phi_w : w \in [W]\} \cup \{h(c) : c \in [0, C]\} \cup \{f(c) : c \in [0, C]\}$
- $\Pr(c|\Phi) = g(\phi_w)\, f(c) \exp(\phi_w \cdot h(c))$
  - This form is very general
  - Can represent Gaussian, Poisson, Beta, Gamma, etc.
  - $g(\phi_w)$ is just a normalization to ensure $\sum_c \Pr(c|\phi_w) = 1$ for each w
  - $h(c)$ is the key count squashing function we will fit
- For simplicity, let $\phi_w,\ h(c), f(c) \in \mathbb{R}$
- Implement $f(c)$ and $h(c)$ as two tables $f[0, \ldots, C]$ and $h[0, \ldots, C]$ shared across all words
- We will fit these and ϕ_w for all w using observed $n(w, c)$
- Particularly interested in what $h[0, \ldots, C]$ looks like after training

Learnt squash function sample





- New model $\Pr(c|\phi_w)$ fits real data
- Both make term burstiness evident

- Squash function $h(c)$ makes sense
- Approximated by $\log(1 + \log(n(d,t))$

TFIDF pitfalls

- Surprisingly useful, given obvious limitations
- Proximity and order
  - does oneplus nord battery last longer than samsung
  - The Samsung Galaxy boasts a 6000mAh battery which lasts longer than the OnePlus
  - The OnePlus is longer than the Samsung, despite its 4500mAh battery
  - Partly solved by proximity models (this course)
  - … and modern language models (with caveats 🔌CS728)
- Named entities
  - jack cunningham top albums
  - Best albums of John Cunningham
  - Partly solved by named entity recognition (🔌CS728)

TFxIDF: vector length and impact

- Multiply TF and IDF together
- $\ell_w(\mathbf{x}) = \mathrm{TF}(x, w) \cdot \mathrm{IDF}(w)$
- Raw 'length' of doc

$$L(\mathbf{x}) = \sqrt{\sum_w \ell_w(\mathbf{x})}$$

- Normalize to final vector components (impact)

$$\mathrm{impact}(x, w) = x_w = \ell_w(\mathbf{x}) / L(\mathbf{x})$$

- Queries rarely repeat words, only TF?
- Double-include IDF for doc and query?

BM25

Given a query Q, containing keywords $q_1, \ldots, q_n$, the BM25 score of a document D is:

$$\text{score}(D, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

where $f(q_i, D)$ is the number of times that q_i occurs in the document D, $|D|$ is the length of the document D in words, and avgdl is the average document length in the text collection from which documents are drawn. k_1 and b are free parameters, usually chosen, in absence of an advanced optimization, as $k_1 \in [1.2, 2.0]$ and $b = 0.75$.[1] $\text{IDF}(q_i)$ is the IDF (inverse document frequency) weight of the query term q_i. It is usually computed as:

$$\text{IDF}(q_i) = \ln\left(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} + 1\right)$$

where N is the total number of documents in the collection, and $n(q_i)$ is the number of documents containing q_i.

Learning-to-rank algorithm

Index organization and query processing

(Sparse retrieval)

CS635

Common theme for sparse retrieval

- Sparse query and document representation
  - Non-zero elements only for terms present in query or document
- Elements are length-normalized, IDF-squashed impacts
- Score is (close to) inner product of the two sparse vectors
- Next: how to store impact in indices for scoring documents wrt given query
  - (At least) two ways to organize postings
- Query execution has to adjust to index organization
  - DAAT (document-at-a-time)
  - TAAT (term at a time)

Two ways to organize postings

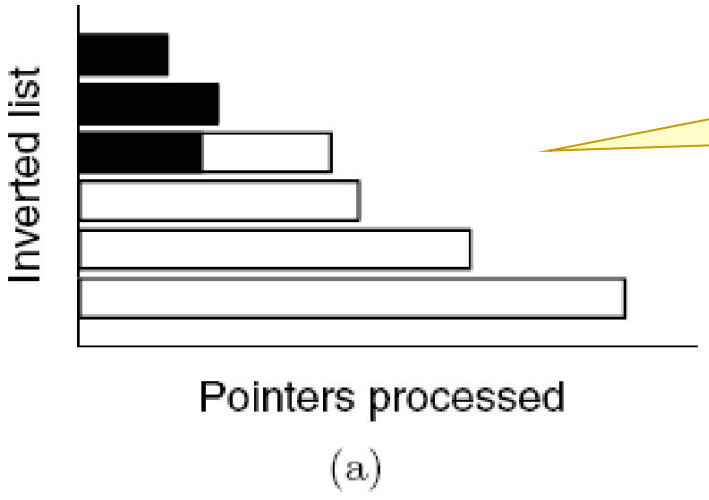
- Doc IDs are assigned arbitrarily before indexing
- Posting lists ordered by doc ID
  - Cursors advance on postings for all query words
  - Best-scoring doc may be at the end of posting
  - Therefore, must scan to end of posting
  - Might omit posting list of low-IDF query term entirely
  - Can complete scoring of one doc before moving to next
- Posting lists ordered by impact
  - Doc IDs come in different orders on different lists
  - Impact of unvisited docs less than min of visited
  - Can abandon a posting midway
  - Maintain data structure with incompletely scored docs

Simpler and more popular implementation

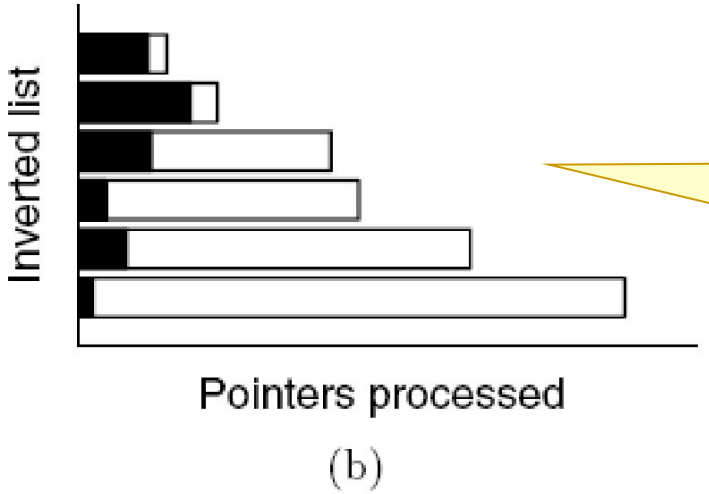
Query processing using inverted lists

- How exactly does TFIDF retrieval happen using inverted lists? DAAT
  - Interested in top-k documents only
  - Speed up using various pruning and termination heuristics
- TAAT: A general framework started by Fagin et al.
  - Worst case (pessimistic) guarantees
  - Depends on impact ordering
- Later, probabilistic bounds
  - Incorrect ranking with small probability

DAAT quit vs. TAAT early termination



IDF-ordered quit heuristic



Impact-ordered continuation and/or pruning heuristic

http://resources.mpi-inf.mpg.de/d5/teaching/ws11_12/irdm/slides/

Where is time spent?

- For queries with relatively rare terms
  - Management of score accumulators
  - Sparse or dense map?
- For queries with some frequent terms
  - Bit processing for decompressing postings
  - Arithmetic to update accumulators
  - Wasted effort in computing scores to throw away
- You can't really afford (updating) a billion accumulators for 300 million queries a day
  - Be sloppy when no one is looking

Index organization: A third option

- Keep posting in doc ID order
- At each posting entry store
  - Doc ID (and positions if desired)
  - Impact of current token on this doc ID
  - Max impact of current token on all docs after current doc ID (to end of posting list)
- Impacts are real numbers
  - Compressed, but still costs space

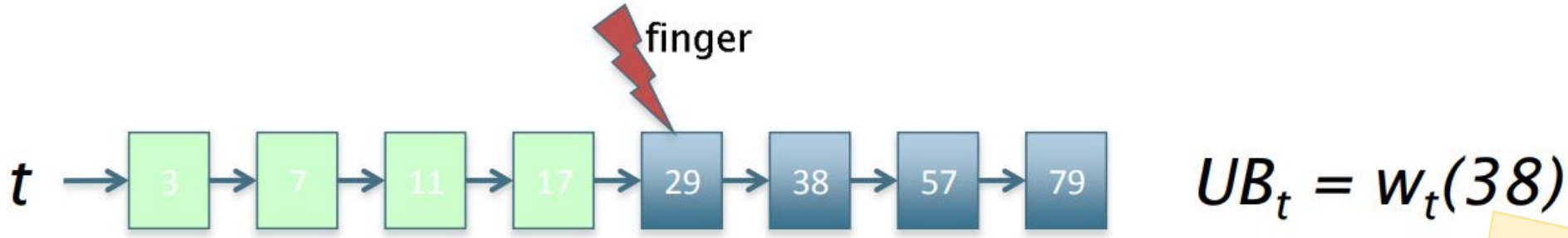
(Docid ordered) inverted list scan 'API'

- For a given query word w
- Cursor/'finger'is positioned at doc ID d
- Can read off d and $\mathrm{impact}(d, w)$ from cursor
- cursor.next(d') for $d' > d$ advances the cursor to the smallest doc ID that is at least d'
- cursor.high() returns the largest impact of w on any doc ID greater than (or equal to?) d
  - Non-increasing as cursor advances
  - Need to store two numbers per doc
  - (Would be trivial for impact-ordered posting lists)

How is `next` implemented?
Skip pointers

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)



True for all posting ordering schemes

Only if posting is impact ordered, otherwise, need to precompute a suffix-max on each posting list

As finger moves right, *UB* drops

Storing impact in postings

- Compress real number into b bits
- Uniform/linear $\left\lfloor 2^b \frac{x-L}{U-L+\epsilon} \right\rfloor$
- Left.Geom $\left\lfloor 2^b \frac{\log x - \log L}{\log U - \log L + \epsilon} \right\rfloor$
- Limit multiplicative error
- Right.Geom for approximating highest impacts

TREC

Web

%corpus

	TREC	Web
	9.4	6.0
	7.8	5.0
	7.0	4.5
	6.3	4.0
	5.6	3.6
	5.1	3.3

TREC, precision at rank 10

	Left.Geom	*Uniform*
$b = 16$	.2174	.2174
$b = 12$	.2174	.2174
$b = 10$	.2174	.2174
$b = 8$	.2196	.2174
$b = 6$	.2130	.2109
$b = 5$	.2109	.2087
$b = 4$	.1891	.1935
$b = 3$	.1674	.1891
$b = 2$	.1609	.1174

Top-K accumulators

- In doc ID ordered postings, once all fingers/cursors are past doc 29, it is completely scored
- Emit $\langle 29, \mathrm{score} \rangle$ to score accumulators
- Crude way to finish: sort N scores, keep K
- Better: keep only K scores, evict on overflow
  - Heaps can do this in $O(\log K)$ time per insert

DAAT (brute force $O(N \log K)$)

Accumulators store a mapping from doc ID to currently computed partial scores

Because postings are doc ID ordered, full score of a doc can be computed before moving on to the next doc

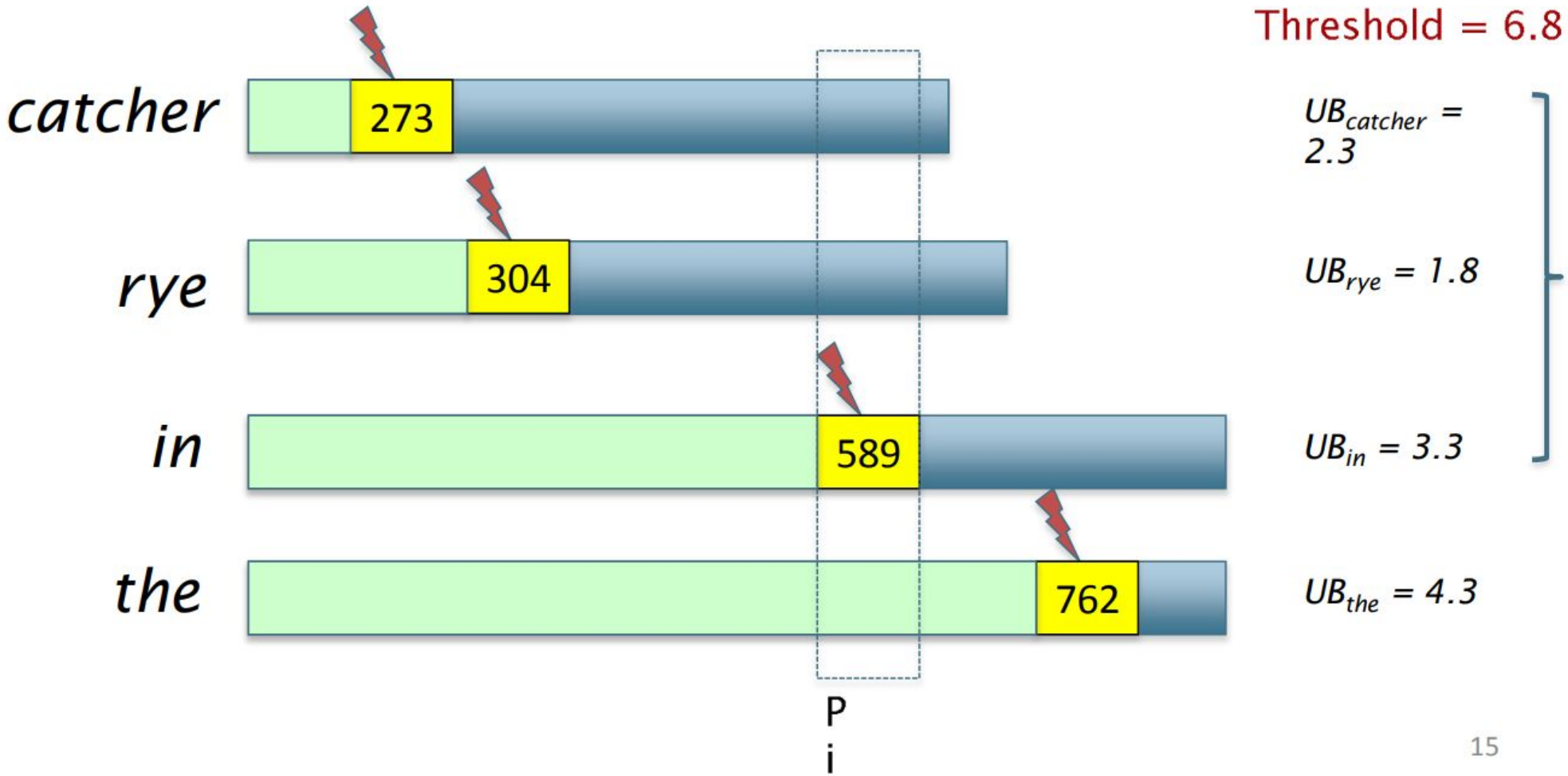
When a doc gets evicted from a **min**-heap, we know there are K better candidates

Weak-AND (WAND) = DAAT+pruning

- Like branch and bound
- Maintain running threshold score, e.g., (lower bound on) Kth largest score computed so far
  - Scores can only increase
- Prune docs whose max possible scores (at the end of scan) are guaranteed to be below threshold
- Compute exact scores for only surviving docs
- Cursors on postings of all query words
- Each doc ID passed over by all cursors has
  - Either been pruned away,
  - Or their full score has been computed

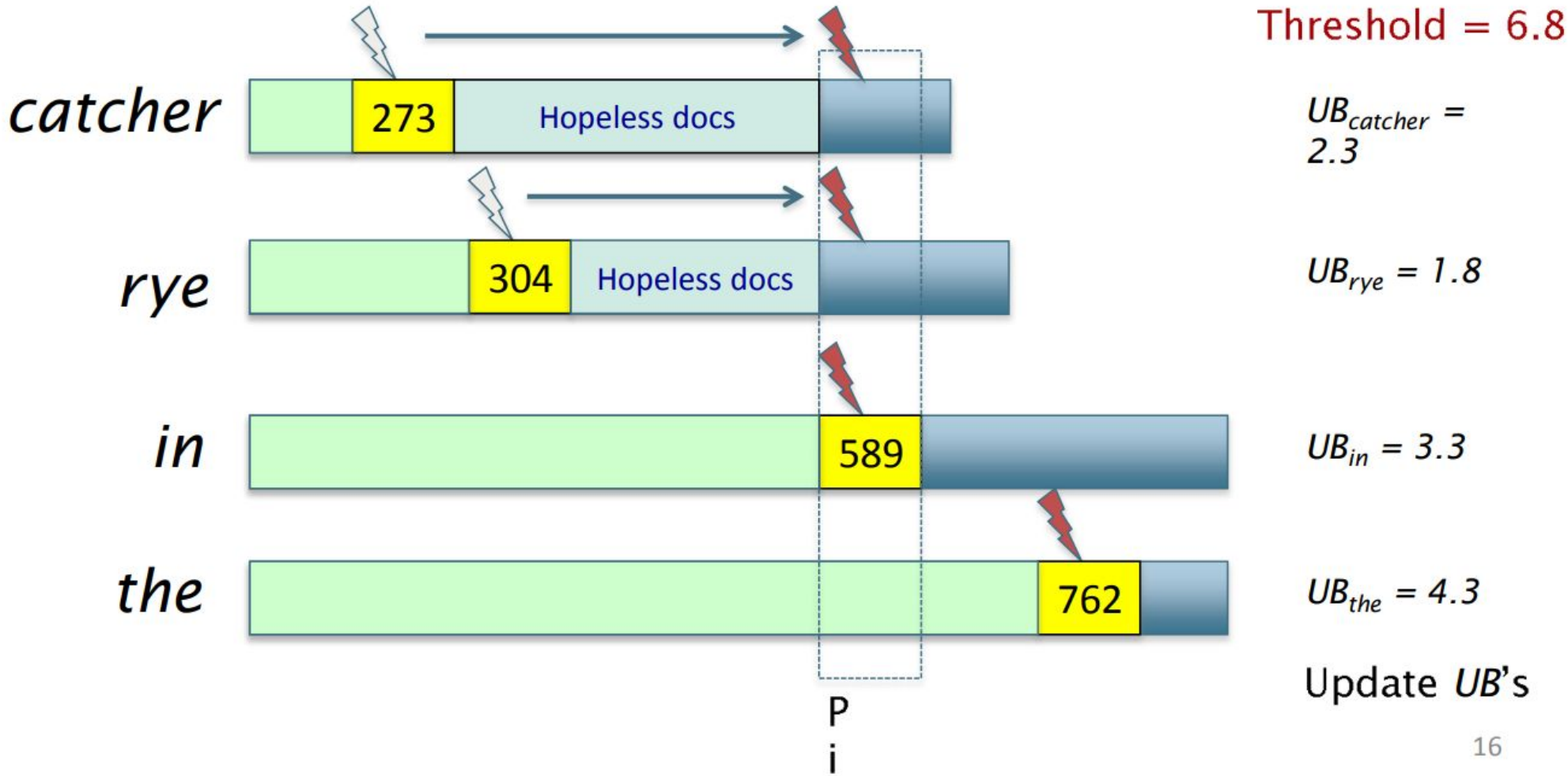
Pivoting

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



Prune docs that have no hope

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



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 …

TAAT using impact-ordered index

- Query word set $\{q_1, \ldots, q_m\}$
- Cartesian space of documents $D_1 \times D_2 \times \ldots \times D_m$
  - Every doc in doc set D_i contains word q_i
  - These sets are expected to intersect
- Per-word impact $s(q_i, d)$, aggregate similarity = $\oplus_i s(q_i, d)$
  - $\oplus$ is a benign (commutative, additive) aggregator
- Sorted access to each dimension in decreasing order of $s(q_i, d)$
  - Can be ensured in impact-sorted IR
  - We are lucky that queries are composed of (few) terms
  - What if they were term pairs? Small graphs? (Later)
- Keep upper bounds and lower bounds on scores of candidate docs

Generic pseudocode

i ranges over dims

Any future record will have a lower score

Explored dimensions of item d

Eviction

Note: need to fully evaluate winner scores

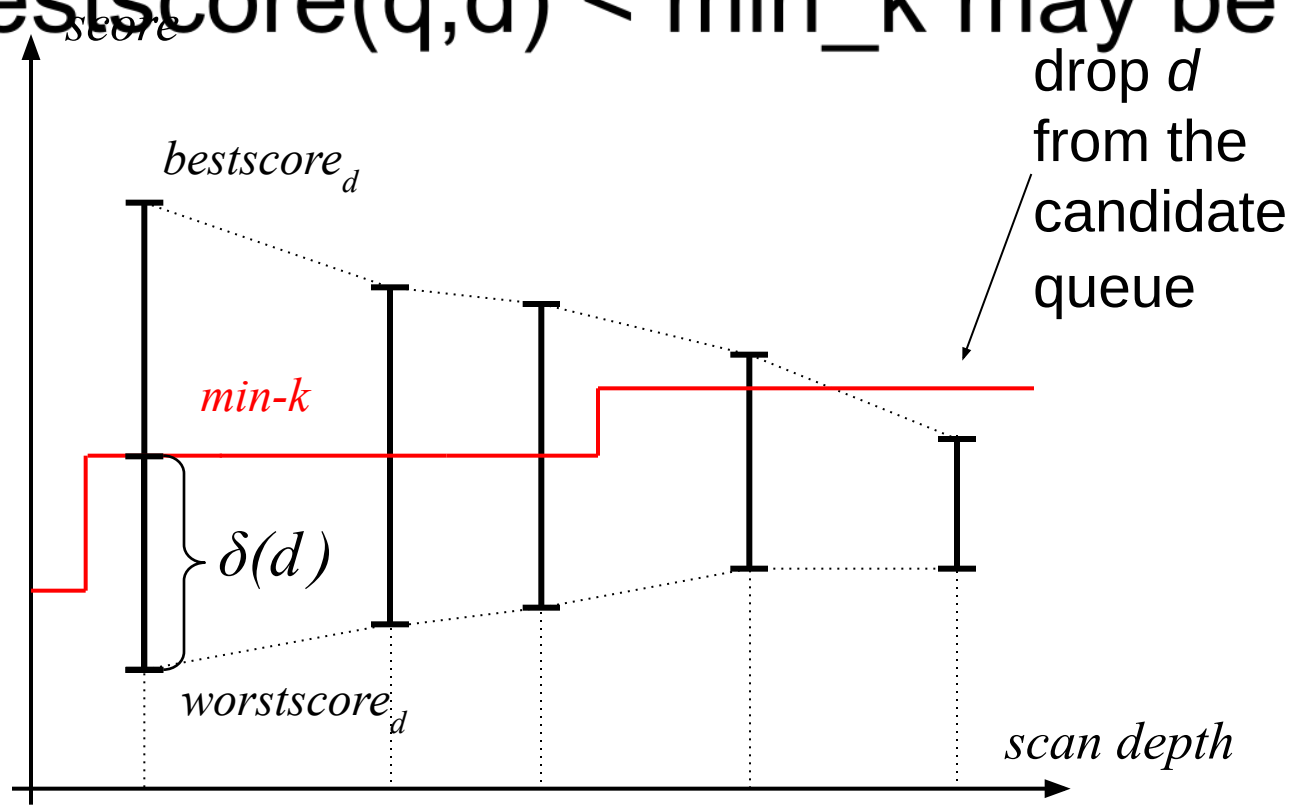
Upper bound

LB assuming all remaining records have zero impact

TA-sorted:
top-k :={dummy$_1$, …, dummy$_k$}; // *with s (dummy$_v$)=0*
min-k := 0;
candidates := ∅;
scan all lists L_i (i = 1..m) in parallel:
// e.g., *round-robin or merged in descending order of s_i values*
  consider item d at position pos_i in L_i;
  if d ∉ candidates then
    candidates := candidates ∪ {d};
    E(d) := {i};
  $high_i := s_i(q_i,d)$; // *current score in L_i*
  E(d) := E(d) ∪ {i};
  bestscore(d):=aggr{aggr{$s_v(q_v,d)|v∈E(d)$}
                aggr{$high_v|v∉E(d)$};
  worstscore(d) := aggr{$s_v(q_v,d)|v∈E(d)$};
  if worstscore(d) > min-k then
    if d ∉ top-k then
      remove $argmin_{d'}${worstscore(d')|d'∈top-k} from top-k;
      candidates := candidates ∪ {d'};
      add d to top-k
    min-k := min{worstscore(d') | d' ∈ top-k};
  if bestscore(d) ≤ min-k then candidates := candidates − {d};
  threshold := max {bestscore(d') | d'∈ candidates};
  if threshold ≤ min-k then exit;

Some properties

- worstscore(q,d) ≤ s(q,d) ≤ bestscore(q,d)
- aggr(worstscore(q,d), aggr{high_i|i∉E(d)}) = bestscore(q,d)
- The test bestscore(q,d) < min_k may be conservative



Refined rankings

- Additional signals from word proximity, geo location, personalization
- Usually inverted index does the first stage heavy lifting
- Then more expensive machine learning methods are applied to rerank

Adding proximity credit to vector space scoring

An exploration of proximity measures in information retrieval

A general scoring model for cooccurrence

A Markov Random Field Model for Term Dependencies