Text-based Ranking

First generation, based on the text of the document and of the query

Documents as binary arrays

Until now, we represented a document as a binary array where each position corresponds to a term and each value is either zero (term is not in document) or one (term is in document).

We would score a document by overlapping it with the query

$$score(d, q) = |q \cap d|$$

It is very costly and is not very useful in term of similarity because it doesn't take into account the lengths, which usually are very different.

Two approaches for normalizing the sizes:

Jaccard coeff → $sim(d, q) = \frac{|d \cap q|}{|d \cup q|}$ RESPECT TRIANGULAR INEQUALITY

Dice coeff → $sim(d, q) = \frac{2|d \cap q|}{(|d| + |q|)}$ DOESN'T RESPECT TRIANGULAR INEQUALITY

IT'S STILL NOT A GOOD WAY OF RANKING, BECAUSE WE ARE NOT CONSIDERING THE IMPORTANCE OF EACH TERM

Documents as vectors

As already said, with binary vectors we can't consider the term frequency of a document: ideally, we would like to show first the documents that talk the most about our query terms, ignoring those terms that don't add relevance (stop words).

WE WOULD LIKE TO WORK IN A VECTOR SPACE, WHERE THE AXES ARE THE TERMS AND THE POINTS OR VECTORS ARE THE DOCUMENTS AND THE QUERIES

Therefore

1) We want to transform queries and documents in real valued vectors and represent them in a space

2) We want to rank the documents according to their proximity to the query in that space

proximity = similarity of vectors ≃ inverse of distance

Note = A vector space may have millions of dimensions

Note = In our case, the vectors are very sparse (most of the entries are zero)

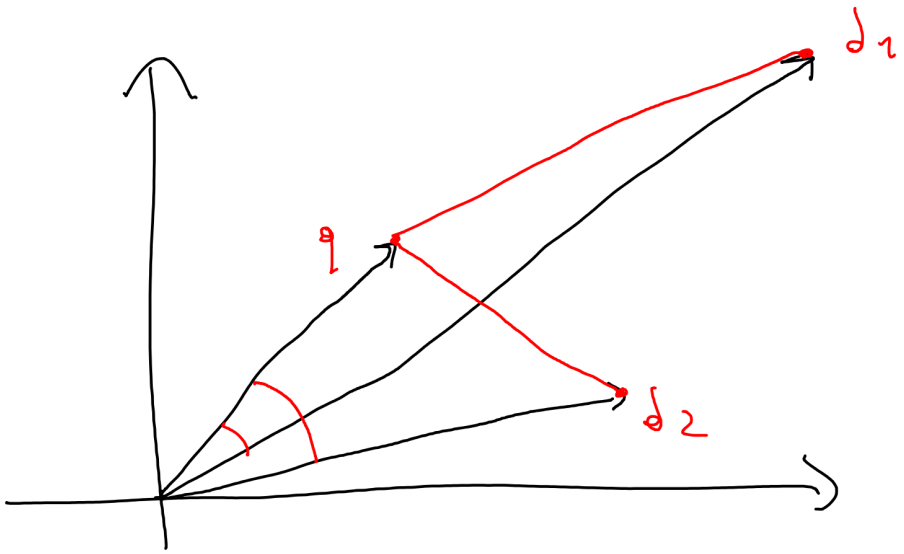
How to estimate the proximity of two vectors?

To rank a document, we need a way to compare two vectors in the vector space (namely, the query and the document)

A GOOD PROXIMITY MEASURE IS THE ANGLE FORMED BY THE TWO VECTORS, NOT THE DISTANCE BETWEEN THEM (i.e., euclidean distance)

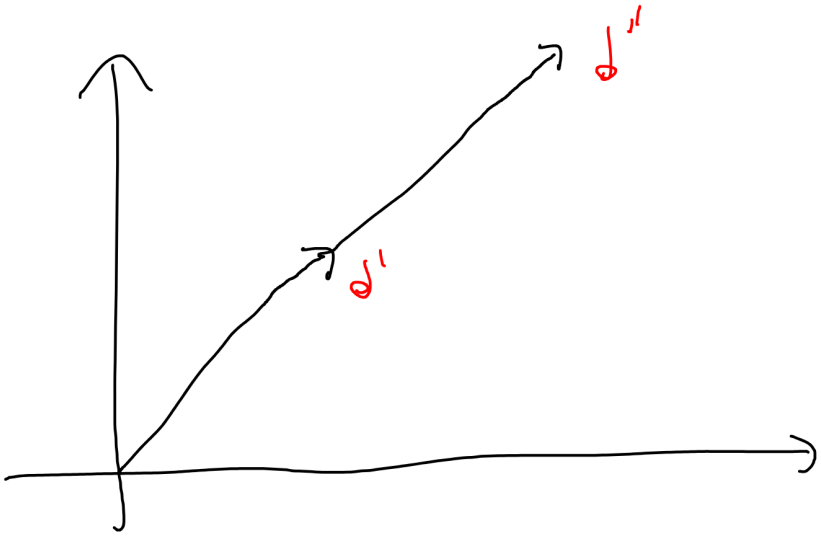
The euclidean distance may be large for vectors of different lengths, even if they are close in space

EXAMPLE



Using euclidean distance, q is more similar to d2 than d1, which is not true.

To better prove that the angle is better than the distance, here is another example



Semantically, d' and d'' have the same content: indeed the angle is zero, but the euclidean distance is large.

Actually, it is easier to use the cosine rather than the angle, but the two notions are equivalent:

- USING ANGLE = VALUES CLOSER TO ZERO ARE BETTER
- USING COSINE = THE COSINE FUNCTION DECREASES MONOTONICALLY IN THE INTERVAL $[0°, 180°]$ AND HIGHER VALUES ARE BETTER $\Longrightarrow$ $\cos(0°) = 1$, $\cos(180°) = -1$

But how do I get the cosine of the angle between two vectors?

COSINE SIMILARITY

$$\cos(\vec{q}, \vec{d}) =$$

$$= \text{Cosine similarity of } \vec{q} \text{ & } \vec{d} =$$

$$= \text{Cosine of the angle between } \vec{q} \text{ & } \vec{d} =$$

$$= \frac{\vec{q} \cdot \vec{d}}{|\vec{q}| \cdot |\vec{d}|} = \frac{\vec{q}}{|\vec{q}|} \cdot \frac{\vec{d}}{|\vec{d}|} =$$

dot product

unit vector

L2 norm

$$= \frac{\sum_{i=1}^{|D|} q_i d_i}{\sqrt{\sum_{i=1}^{|D|} q_i^2} \cdot \sqrt{\sum_{i=1}^{|D|} d_i^2}}$$

weights

NOTE = We transform the vectors in unit vectors by dividing them by their L2 norm ($|\vec{x}| = \sqrt{\sum_i x_i^2}$).

This way the query and the documents have comparable values (weights), even if they have different lengths

NOTE = if the two vectors are already length-normalized

$$\cos(\vec{q}, \vec{d}) = \vec{q} \cdot \vec{d} = \sum_1^{|D|} q_i d_i$$

What are the values g_i and d_i, then?

WE CALL THEM TF-IDF WEIGHTS

TF - IDF

$w_{t,d}$ = weighted score of document d for term t =

$= tf_{t,d} \times idf_t$

if t is not in d or it is not relevant (i.e., t appears in all the documents) $\Rightarrow w_{t,d} = 0$

- $tf_{t,d}$ = term frequency of term t in document d =

  = occurrences of t in d

- idf_t = inverted document frequency of term t =

$$= \log\left(\frac{n}{n_t}\right)$$

n: # documents in the collection

n_t: # documents in the collection that contains term t

You can call it df_t, for consistency

Note that we could rank a document just by the term frequency of t, but then we would be misleaded by stopwords terms (which appear everywhere), and that's why we also consider the inverted document consistency of t (it lowers the score if t is in many documents)

This way we replaced the binary representation of a document with a real valued vector and introduce the VECTOR SPACE MODEL (in contrast with the previous one, the Boolean model)

Example

DOCUMENT VECTOR WITH TF-IDF WEIGHTS

TERM	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	13,1	11,4	0,0	0,0	0,0	0,0
Brutus	3,0	8,3	0,0	1,0	0,0	0,0
Caesar	2,3	2,3	0,0	0,5	0,3	0,3
Calpurnia	0,0	11,2	0,0	0,0	0,0	0,0
Cleopatra	17,7	0,0	0,0	0,0	0,0	0,0
mercy	0,5	0,0	0,7	0,9	0,9	0,3
worser	1,2	0,0	0,6	0,6	0,6	0,0

Regarding the storage usage of such vectors

of course we don't store a matrix like in the example above, but we use the classic inverted list approach, with a couple of additions:

- For every term, the dictionary also stores the length of the corresponding posting list, so that the IDF is computed in constant time
- For every docID in the posting list, we also append the corresponding $tf_{t,d}$, which is small
- You could also add other useful metadata, as seen for the phrase query processing solutions

Okay, but how do we do the actual scoring of the documents for a given query?

COSINE_SCORE : q ——> Top K documents

$\forall d \in Docs$, Returns the K best $\cos(\vec{q}, \vec{d})$ normalized

```
COSINE_SCORE(q):
    float  SCORES[N] = 0          // N = # docs
    float  LENGTHS[N] = { i : i = size(d)  d ∈ docs }
                                  // length of the docs
    for t in q:
        plist = get posting list of t =
              = (docID, tf_{t,docID})

        compute w_{t,q}

DOT PRODUCT PHASE
        for (d, tf_{t,d}) in plist:
            Compute w_{t,d}
            SCORES[d] += w_{t,d} · w_{t,q}

NORMALIZATION PHASE
    for (SCORE, LENGTH) in zip(SCORES, LENGTHS):
        SCORE = SCORE / LENGTH

    return { d : SCORES[d] is a Top K doc }
```

Exercise

Given the four texts: 1) "la bella casa", 2) "le belle case", 3) "casa bella bella", 4) "la casa le case bella rosa"

- Compute the inverted index for the texts, compress with gap encoding, then with γ-code

	NORMAL	GAP
BELLA	1, 3, 4	1, 2, 1
BELLE	2	2
CASA	1, 3, 4	1, 2, 1
CASE	2, 4	2, 2
LA	1, 4	1, 3
LE	2, 4	2, 2
ROSA	4	4

$\gamma(1) = 1$, $\gamma(2) = 01$, $\gamma(3) = 011$, $\gamma(4) = 00100$

- Compute the tf-idf vectors

"JUST TO KEEP IT SOMEWHERE"

TF-IDF	1)	2)	3)	4	idf$_t$
BELLA	$1 \cdot \log(\frac{4}{3})$	0	$2 \cdot \log(\frac{4}{3})$	$1 \cdot \log(\frac{4}{3})$	$\log(\frac{4}{3})$
BELLE	0	$1 \cdot 2$	0	0	$\log(\frac{4}{1}) = 2$
CASA	$1 \cdot \log(\frac{4}{3})$	0	$1 \log(\frac{4}{3})$	$1 \cdot \log(\frac{4}{3})$	$\log(\frac{4}{3})$
CASE	0	$1 \cdot 1$	0	$1 \cdot 1$	$\log(\frac{4}{2}) = 1$
LA	$1 \cdot 1$	0	0	$1 \cdot 1$	$\log(\frac{4}{2}) = 1$
LE	0	$1 \cdot 1$	0	$1 \cdot 1$	$\log(\frac{4}{2}) = 1$
ROSA	0	0	0	$1 \cdot 2$	$\log(\frac{4}{1}) = 2$

- Find the text most similar to 4) using cosine without normalization (simple dot product).

$\rightarrow sim(1,4) = \log\left(\frac{4}{3}\right)\log\left(\frac{4}{3}\right) + 0.0 + \log\left(\frac{4}{3}\right)\cdot\log\left(\frac{4}{3}\right) +$

$+ 0.1 + 1.1 + 0.1 + 0.2 =$

$= 2\cdot\log\left(\frac{4}{3}\right)^2 + 1 \simeq 1.3$

$\rightarrow sim(2,4) = 0.\log\left(\frac{4}{3}\right) + 2.0 + 0.\log\left(\frac{4}{3}\right) +$

$+ 1.1 + 0.1 + 1.1 + 0.2 =$

$= 2$

$\rightarrow sim(3,4) = 2\log\left(\frac{4}{3}\right)\cdot\log\left(\frac{4}{3}\right) + 0.0 +$

$+ \log\left(\frac{4}{3}\right)\cdot\log\left(\frac{4}{3}\right) + 0.1 +$

$+ 0.1 + 0.1 + 0.2 =$

$= 3\log\left(\frac{4}{3}\right)^2 \simeq 0.52$

2) is the most similar to 4)

CONCLUSIONS ON VECTOR SPACES

The vector space is good for bag-of-words queries, but not good with operators (Boolean, wildcard, positional, proximity).

The real problem is that it is really easy for spammers to fuck up everything (just fill the document with the same terms, the tf increases but the idf is still the same), and that's why text-based approaches are not used in modern web search engines.

ANYWAY, WE HAVEN'T FINISHED YET:

↳

computing the cosine is expensive due to the high-dimensionality of a vector space, but I still want those K top scoring documents!

Solutions?

Approximate Retrieval (Fast top-K retrieval)

Find a set A of candidates with $K < |A| << N$, compute the score for each doc (doesn't have to be cosine) and returns the K top scoring docs in A.

Set A doesn't necessarily contain all top-K docs, but has many docs belonging to that (that's why it's an approximation)

How do we find A?

Consider only those docs containing at least one query term (of course), and use one of the following Approaches:

- Only consider docs containing many of the query Terms

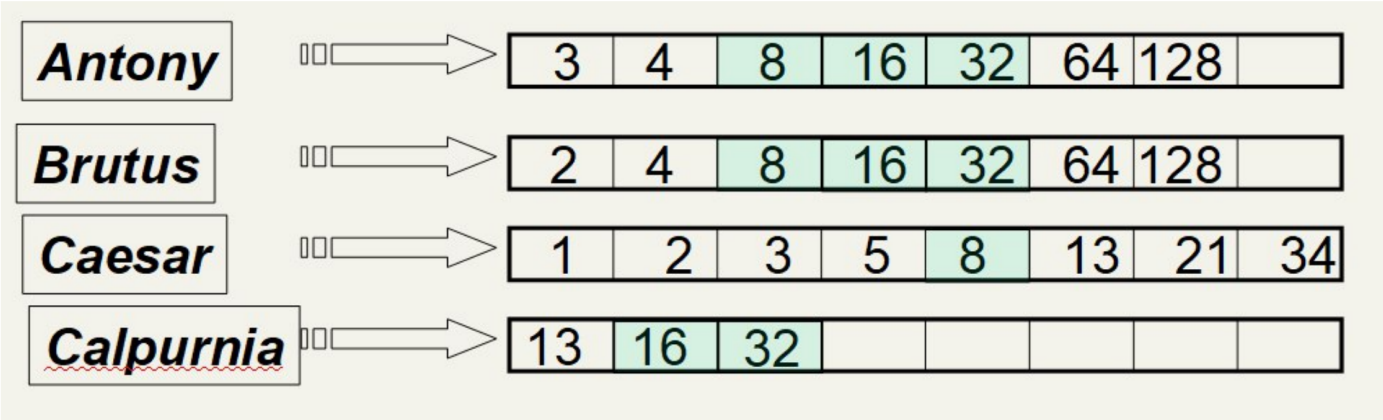
  For multi-term queries, select and score only those docs containing most of the query Terms.

  It works by imposing a soft-AND on the query, where the threshold parameter is high (like, q-1 terms out of q terms of the query should be in the doc).

NOTE EASY TO IMPLEMENT INVERTED INDEX

EXAMPLE

Q = ANTONY BRUTUS CAESAR CALPURNIA



COMPUTE SCORE FOR DOCS 8, 16, 32 ONLY

- Only consider high idf query terms

High inverted document frequency

$\Longrightarrow$

short posting list

$\equiv$

the term is not widely used

the idea is that rare terms are more valid for the search result, while more common terms contribute little to the scores (they don't alter the rank ordering much)

NOTE EASY TO IMPLEMENT, JUST CHANGE IN THE COSINE_SCORE'S PSEUDOCODE THE "for t in q" IN SOMETHING LIKE "for t in High-idf(q)"

- Only consider the docs in the "Champion List"

1) PREPROCESS PHASE = FIND THE m best docs for each term (those with higher tf-idf)

2) SEARCH PHASE = Compute cosine between the query and those docs only! then pick the top k

NOTE WORKS WELL ONLY IF m > k

NOTE m can be different between the terms

How do you retrieve the top-k documents of AND queries?

We want a champion list with docs containing all the terms of the query

1) Take the champion list of all the query term
2) Intersect them
3) Get top-k docs

NOTE: The champion lists on the single terms should contain m > k docs or you won't return enough results.

- Fancy-hits heuristic (Champion List evolution)

1) PREPROCESS PHASE =

- Assign docID by decreasing weights, using PageRank
- Sort docID by decreasing PR weights
- Define FH(t) = m best docs for term t, using tf-idf weights (LIKE THE CHAMPION LIST)
- Define IL(t) = remaining docs containing the term t

OSS
The high scorer are in FH and in the front of IL

2) SEARCH PHASE (for a term):

- Compute the score (tf-idf + PR usually) of all the docs in FH(t), like in champion lists, and keep the top K
- scan IL(t), compute the scores as above and possibly returns some docs alongside the top K.

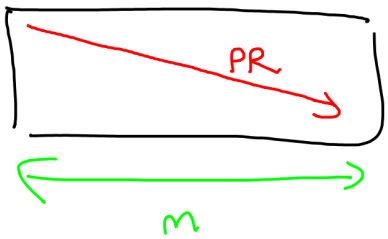
Stopping criterion = stop after M returns or if, given doc d:

$$tf\text{-}idf_d + PR_d < tf\text{-}idf_K \quad (??)$$

EXAMPLE

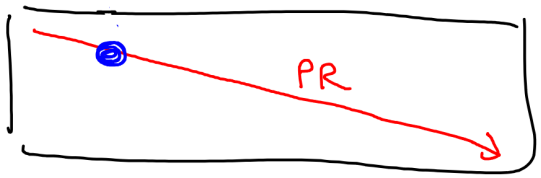
Let's say that Q = "PISA TORRE",
Let's compute FH and IL for both Term (?!)

PISA



Top-m by TF-IDF

FH(t)



IL(t)

How do you retrieve the top-k documents of AND queries?

1) Take FH (include IL too?) of each term

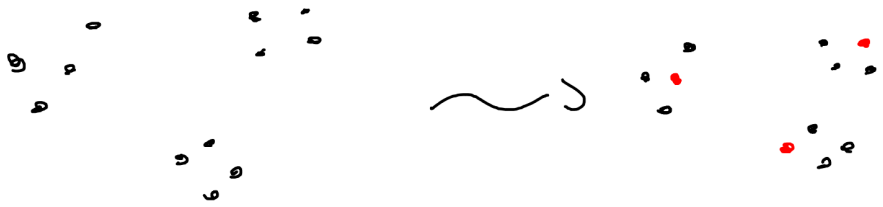
2) Scan (in parallel?) the FHs and returns the docs where all the terms appear
(Kind of merge() in merge sort)

- Clustering

RANDOM SAMPLING IS FAST AND REFLECT THE DATA DISTRIBUTION

1) PREPROCESS PHASE :

- Pick $\sqrt{N}$ docs at random and call them leaders



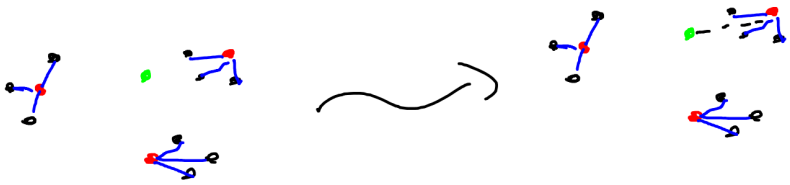
- For every other doc, locates its nearest leader and call it a follower

(Ideally, each leader has $\sim\sqrt{N}$ followers)

YOU COULD ALSO TRY WITH MORE THAN ONE LEADER FOR BETTER TOP-K

2) QUERY PHASE :

- Given a query Q, find its nearest leader L



- Seek for the K nearest docs among L's followers

Exact Retrieval

Given a query Q, find the exact top K docs for Q using some ranking function Ω.

no approximations!!

Simplest strategy

- Find all documents interested by the query
- Compute score $r(d)$ for all those docs
- Sort results by score
- Return top K results

Of course, it is too costly like this and we don't have time to score every document:

we want to avoid scoring docs that won't make it into the top K

LET'S SEE SOME TECHNIQUES

WAND technique

A pruning method that uses a max heap over the real document scores, which is proved to contain the exact top-K docs at the end of the process.

The idea of the algorithm is similar to the Branch & Bound technique:

- Maintain a "running" threshold score θ (which means that it changes overtime), equal to the K^{th} highest score computed so far (the "worst" top-K doc)
- Prune those docs whose scores are guaranteed to be below the threshold
- Compute exact scores for unpruned docs only

Before showing the algo...

- To use WAND, we need a special iterator on the posting lists, that can only move towards the first docID larger then a certain value (since we assume ordered postings, the iterator moves only to the left, therefore towards the larger docIDs)

↳ WE CAN USE SKIP POINTERS AND ELIAS-FANO COMPRESSION

- We assume that $s(t,d)$ = score of t in d

  and that $s(d) = \text{score of } d = \sum_i^{|q|} s(t_i, d)$

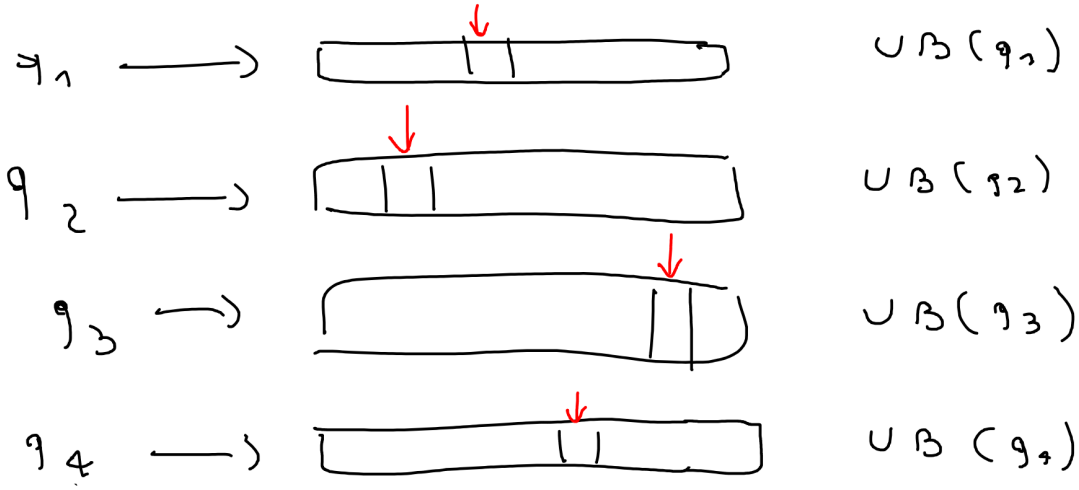
- every query term t have a precomputed upper bound value $UB(t)$ such that $\forall d.\ s(t,d) \leq UB(t)$, therefore it is the maximum possible score for t

- The threshold Θ is initialized to 0, but it raises whenever the "worst" of the current top-K has a score above the threshold; this way

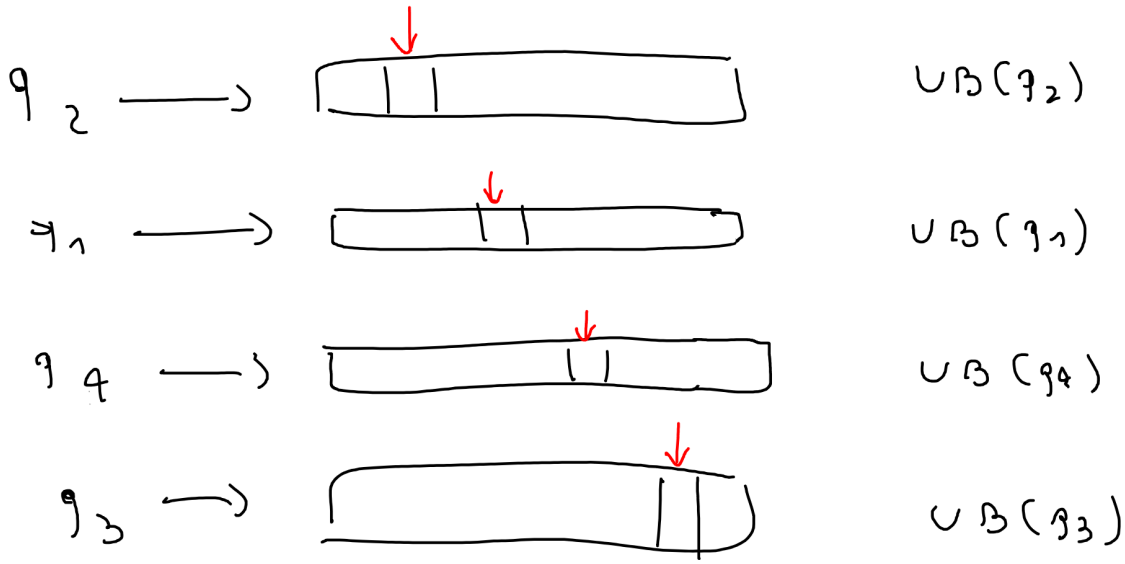
  $s(d) \geq \Theta$ for every d in Top-K

The alg. now

Given the query Q and the postings of the query term, let's consider a generic step of WAND, where the special iterator of every posting is in the following positions:

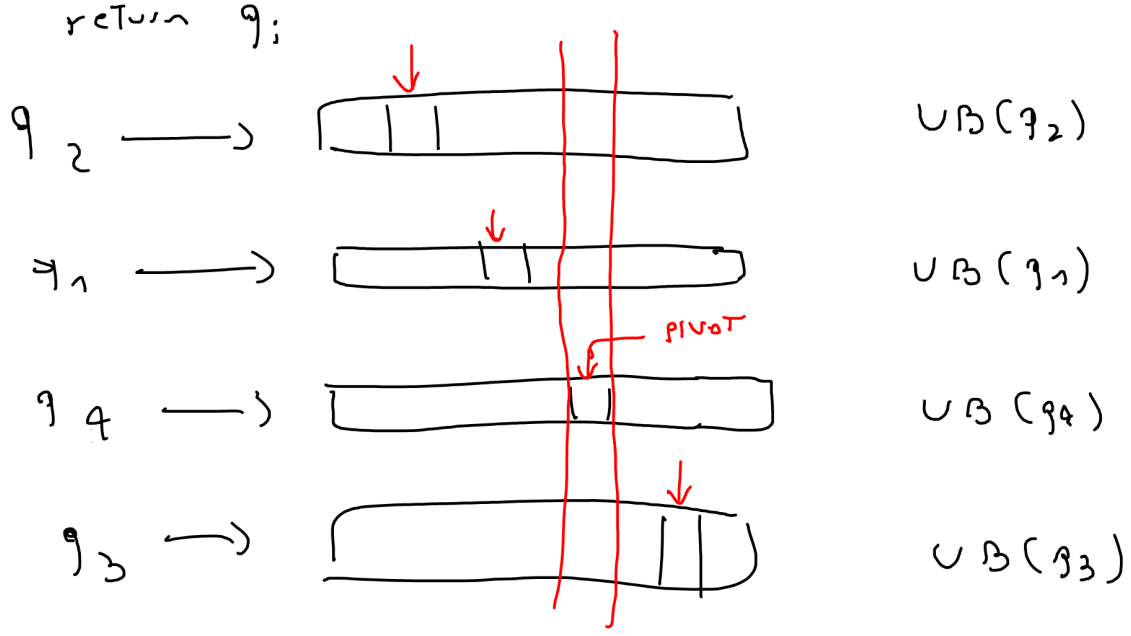


1) Sort the pointers to the postings by increasing docID



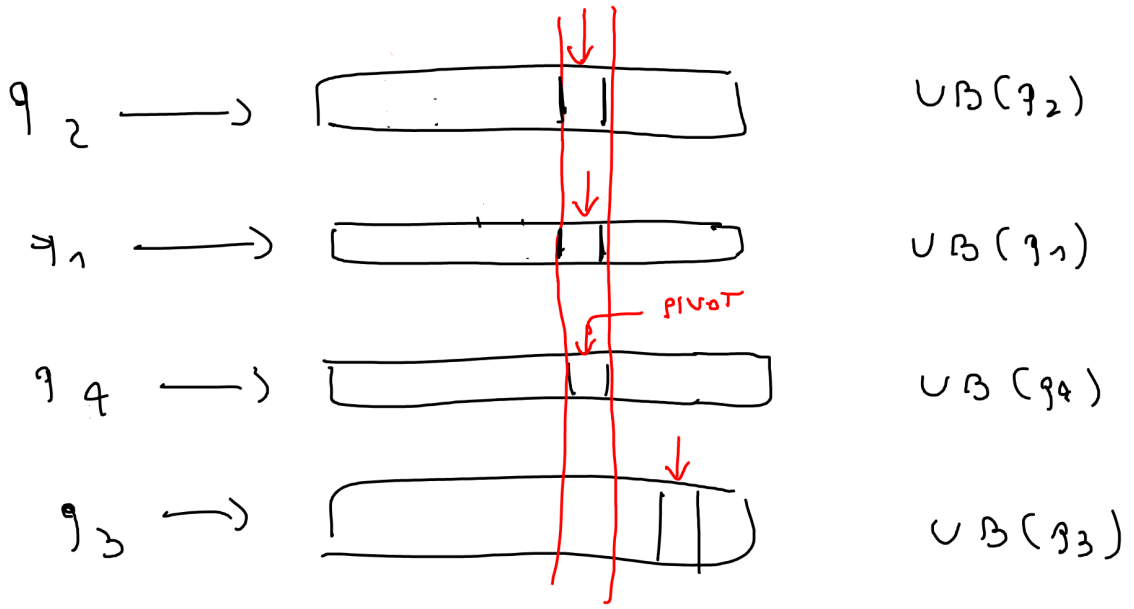
2) Find the "pivot": the first pointer in this order for which the sum of the UBs is at least equal to the threshold

```
i = 0
sum = -1
{q2, q1, q4, q3}  // sorted plist pointers
while sum < Θ :
    sum += UB(qi)
    i++
return qi;
```



3) Move special iterators to the pivot

(this way you prune everything between the current position and the pivot)



4) If the value in the pivot is present in enough postings (soft AND), compute the full score of that doc, else advance the pointers.

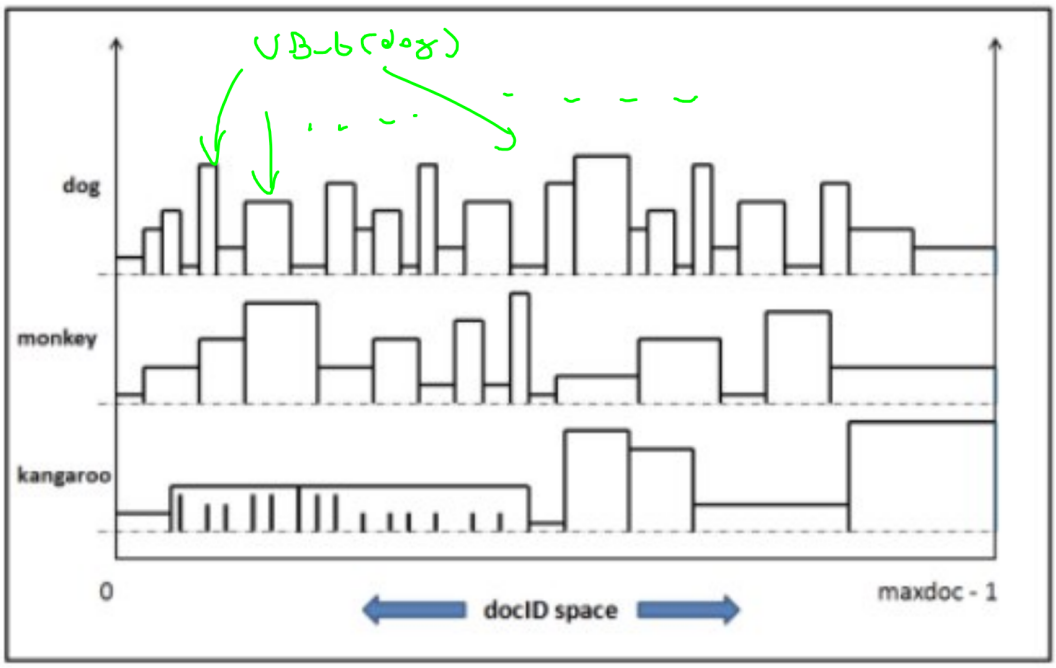
If the pivot is inserted in the current Top-K ($\pi(d) > \theta$), update Threshold θ

5) Repeat from 2)

WAND LEADS TO A 90%+ REDUCTION IN SCORE COMPUTATIONS AND GIVE US A SAFE RANKING. IT PERFORMS EVEN BETTER WITH LONGER QUERIES

Blocked WAND

Like WAND, but instead of having a single UB per t (used on the whole posting), we partition the posting into blocks and store for each block b the maximum score UB_b(t).



Block-Max WAND

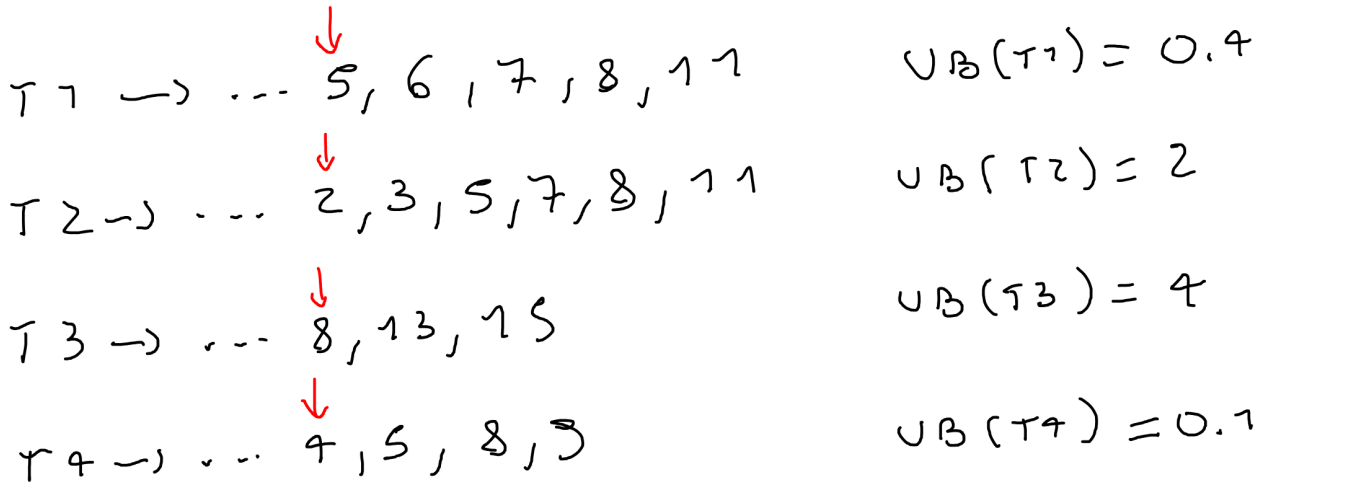
The new version, divided in two steps:

1) Like WAND (p = pivot position
d = pivot doc ID = plist(p)

2)

EXERCISE

Consider the WAND alg. over 4 postings, assuming that the alg. is at the following step (iterators at the head)



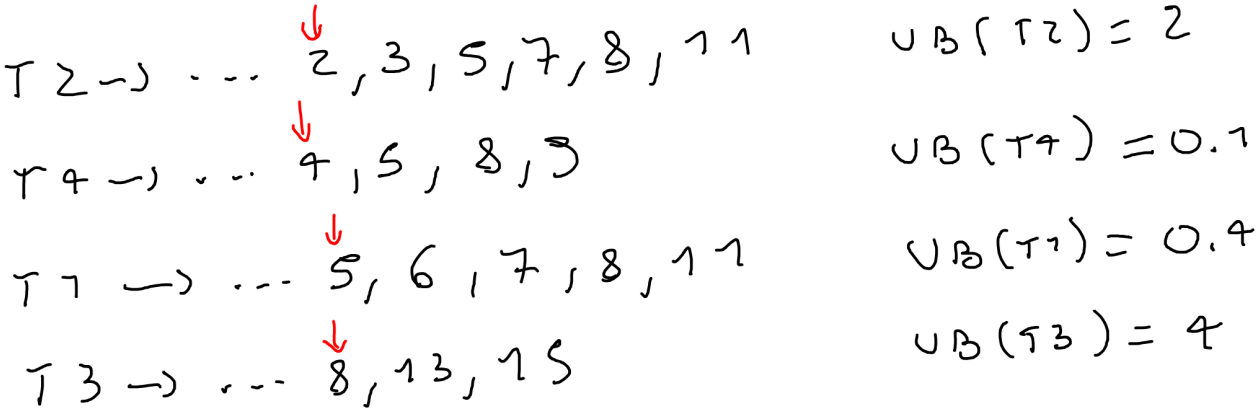
AND $\Theta = 2.3$

WHICH IS THE NEXT DOC TO BE FULL SCORED?

Let's compile the next step

STEP 1

1) Sort by head



2) find "pivot" docID

sum = 0 $\quad \Theta = 2.3$

(T2) sum = 0+2 = 2 $\quad \Theta = 2.3$

(T4) sum = 2+0.1 = 2.1 $\quad \Theta = 2.3$

(T7) sum = 2.1+0.4 = 2.5 $\quad \Theta = 2.3$ stop

docID = 5 pointed by T4 is pivot

3) Move every one to pivot docID

T2 → ... ~~2, 3~~, 5, 7, 8, 11

T4 → ... ~~4~~, 5, 8, 9

T7 → ... 5, 6, 7, 8, 11

T3 → ... 8, 13, 15

4) Does the pivot docID contains most of the terms?

soft-AND ({T7 T2 T3 T4}, 3) = ... = 3 YES!!

Compute the full score for docID = 5.

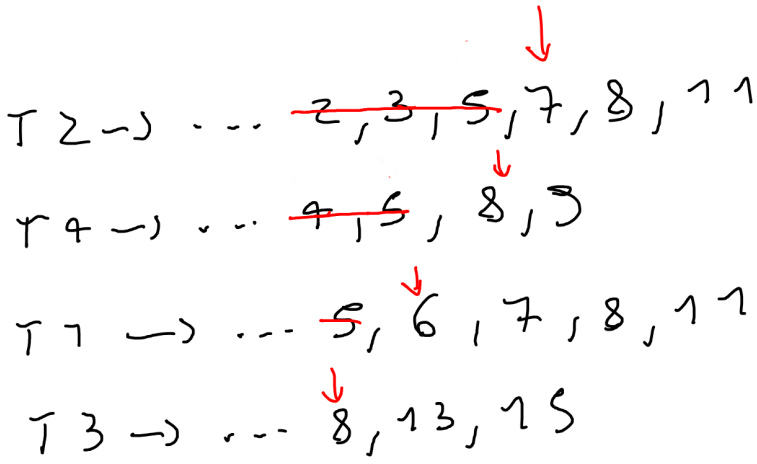
if $s(5) > \Theta$, update Θ and

append docID = 5 to the Top-K results

But what if docID = 5 did not contain term T2?

soft-AND ({T1 T2 T3 T4}, 3) = ... = 2 Not enough!!

Advance the pointers pointing to docID = 5 and start again



Relevance feedback retrieval

"Classic" Relevance feedback

The user is involved in the retrieval process of the initial set of results by giving a relevance feedback

1) The user issues a short and simple query
2) The user marks some results as relevant or non-relevant
3) The system computes a better representation of the information need through the user feedback.

This way the system is able to create a new query that moves towards the relevant docs and away from the irrelevant

Rocchio

$$\vec{q}_{new} = \alpha \cdot \vec{q}_{old} + \beta \frac{1}{|D_r|} \sum_{\vec{d} \in D_r} \vec{d} - \gamma \frac{1}{|D_n|} \sum_{\vec{d} \in D_n} \vec{d}$$

q_{old} is updated by summing the relevant documents D_r and subtracting the irrelevant document D_n, where α, β, γ are control parameters used to modify the "trust factors"

4) Ask the user to try the new query, and repeat

DOCUMENT RE-RANKING !!

problems

- Users often are reluctant to provide explicit feedbacks
- After applying the feedback (through Rocchio), it is often harder to understand why a particular document is now retrieved
- No evidence that we are not simply wasting the user's time

Pseudo-Relevance feedback

The system assumes that the top k documents of a user's query are relevant (automating the user's job), computes relevance feedback using Rocchio and repeat.

Works well on avg, but several iterations can cause query drift

Query expansion vs Relevance feedback

In query expansion, the user gives an additional input on words and phrases, saying if a term is good or bad.

In Relevance Feedback, the user gives an additional input on documents, saying which one are relevant and which one are not.