Information Retrieval 3

Databases of text

tf-idf weighting

$tf_{ij} = f_{ij} / max_i\{f_{ij}\}$

- Commonly normalize *term frequency* (*tf*) by dividing by the frequency of the most common term in the document:

df_i = document frequency of term i

= number of documents containing term i

idf_i = inverse document frequency of term i,

$= \log_2 (N / df_i)$

(N: total number of documents)

Computing TF-IDF -- An Example

Given a document containing tokens with frequencies:

A(3), B(2), C(1)

Assume collection contains 10,000 documents and document frequencies of these terms are:

A(50), B(1300), C(250)

Then:

A: tf = 3/3; idf = $\log_2(10000/50)$ = 7.6; tf-idf = 7.6

B: tf = 2/3; idf = $\log_2(10000/1300)$ = 2.9; tf-idf = 2.0

C: tf = 1/3; idf = $\log_2(10000/250)$ = 5.3; tf-idf = 1.8

Similarity Measure
Inner Product

- Similarity between vectors for the document $\boldsymbol{d}_i$ and query $\boldsymbol{q}$ can be computed as the vector inner product (a.k.a. dot product):

$$\text{sim}(\boldsymbol{d}_j, \boldsymbol{q}) = \boldsymbol{d}_j \boldsymbol{\cdot} \boldsymbol{q} = \sum_{i=1}^{t} w_{ij} w_{iq}$$

where w_{ij} is the weight of term i in document j and w_{iq} is the weight of term i in the query

- For binary vectors, the inner product is the number of matched query terms in the document (size of intersection).
- For weighted term vectors, it is the sum of the products of the weights of the matched terms.

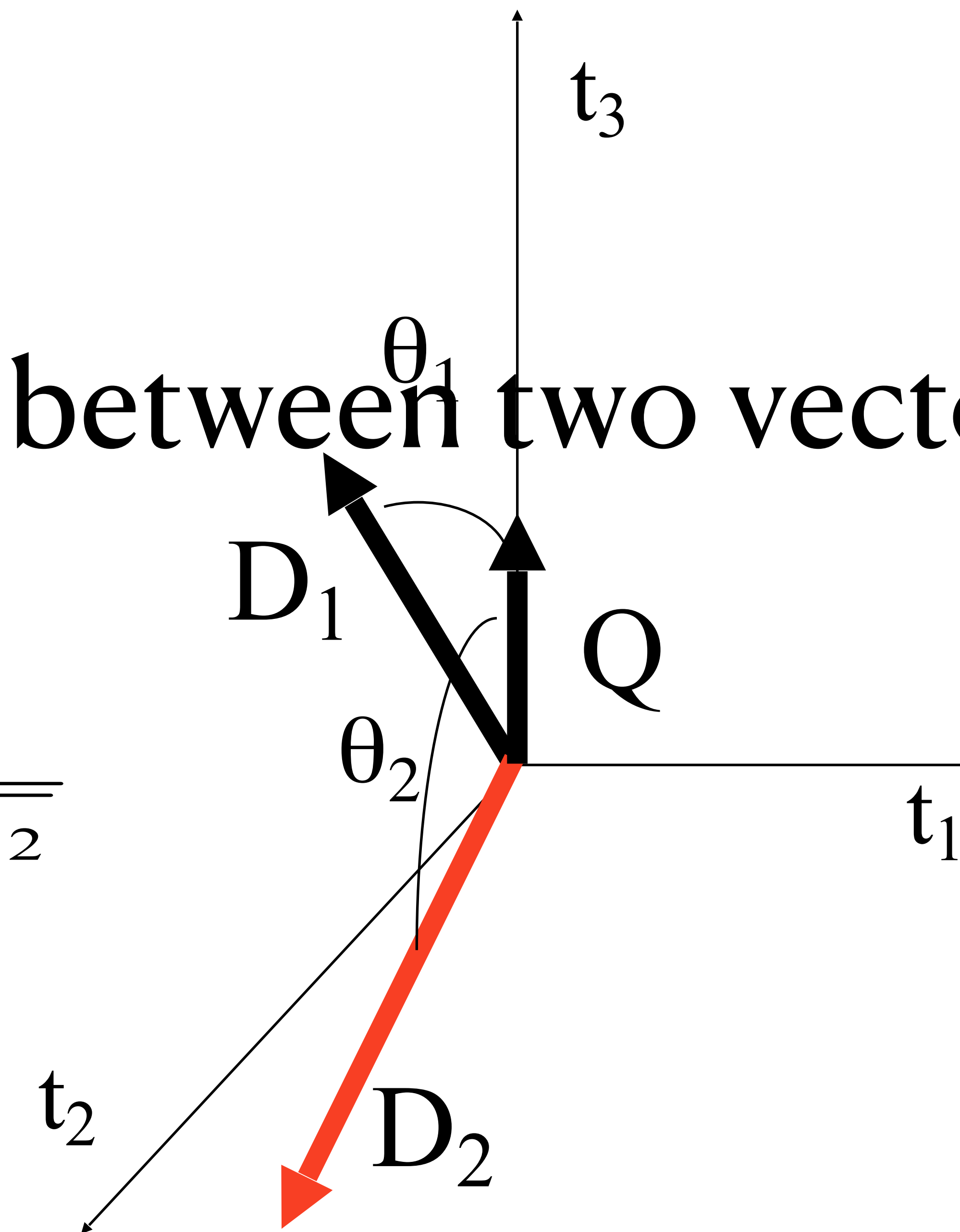
Properties of Inner Product

- The inner product is unbounded.
- Favors long documents with a large number of unique terms.
- Measures how many terms matched but not how many terms are *not* matched.

Cosine Similarity Measure

- Cosine similarity measures the cosine of the angle between two vectors.
- Inner product normalize vector lengths.

$$\text{CosSim}(\mathbf{d}_j, \mathbf{q}) = \frac{\sum_{i=1}^{t} (w_{ij} \cdot w_{iq})}{\sqrt{\sum_{i=1}^{t} w_{ij}^2 \cdot \sum_{i=1}^{t} w_{iq}^2}}$$



$D_1 = 2T_1 + 3T_2 + 5T_3$ $\quad$ $\text{CosSim}(D_1, Q) = 10 / \sqrt{(4+9+25)(0+0+4)} = 0.81$

$D_2 = 3T_1 + 7T_2 + 1T_3$ $\quad$ $\text{CosSim}(D_2, Q) = 2 / \sqrt{(9+49+1)(0+0+4)} = 0.13$

$Q = 0T_1 + 0T_2 + 2T_3$

D_1 is 6 times better than D_2 using cosine similarity but only 5 times better using inner product.

Simple Implementation

Convert all documents in collection D to *tf-idf* weighted vectors, $\boldsymbol{d_j}$, for keyword vocabulary V.

Convert query to a *tf-idf*-weighted vector $\boldsymbol{q}$.

For each $\boldsymbol{d_j}$ in D do

Compute score s_j = cosSim($\boldsymbol{d_j}, \boldsymbol{q}$)

Sort documents by decreasing score.

Present top ranked documents to the user.

Time complexity: O(|V|·|D|)   Bad for large V & D !

|V| = 10,000; |D| = 100,000; |V|·|D| = 1,000,000,000

Comments on Vector Space Models

- Simple, mathematically based approach.
- Considers both local (*tf*) and global (*idf*) word occurrence frequencies.
- Provides partial matching and ranked results.
- Tends to work quite well in practice despite obvious weaknesses.
- Allows efficient implementation for large document collections.

Better Implementation

Using an inverted index

- Tokens that are not in both the query and the document do not effect cosine similarity.
  - Product of token weights is zero and does not contribute to the dot product.
- Usually the query is fairly short, and therefore its vector is *extremely* sparse.
- Use inverted index to find the limited set of documents that contain at least one of the query words.

Suppose Documents look like

docID				
1	cat	home	ball	…
2	ball	park	home	…
3	home	paint	people	…

Then inverted index is

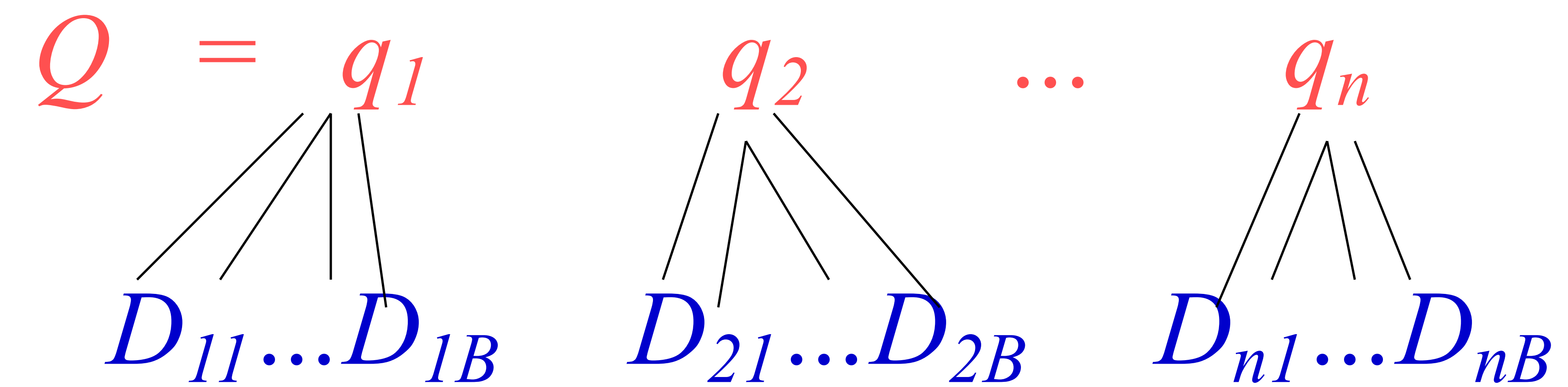
wordID				
cat	doc1			…
ball	doc2	doc1		…
home	doc3	doc1	doc2	…

Using Inverted Index

- Identify those documents that have the query terms
  - Question do you do intersection, union???
- Compute TF-IDF match on that set.

Inverted Query Retrieval Efficiency

- Assume that, on average, a query word appears in B documents:



- Then retrieval time is O($|Q|\ B$), which is typically, **much** better than naïve retrieval
  - Naive: O($|V|\ N$),
    - $|Q| << |V|$
      - $4 < 13{,}000{,}000$
    - $B << N$.
      - "About 9,660,000 results" << 50,000,000,000,000

https://www.kevin-indig.com/googles-index-is-smaller-than-we-think-and-might-not-grow-at-all/

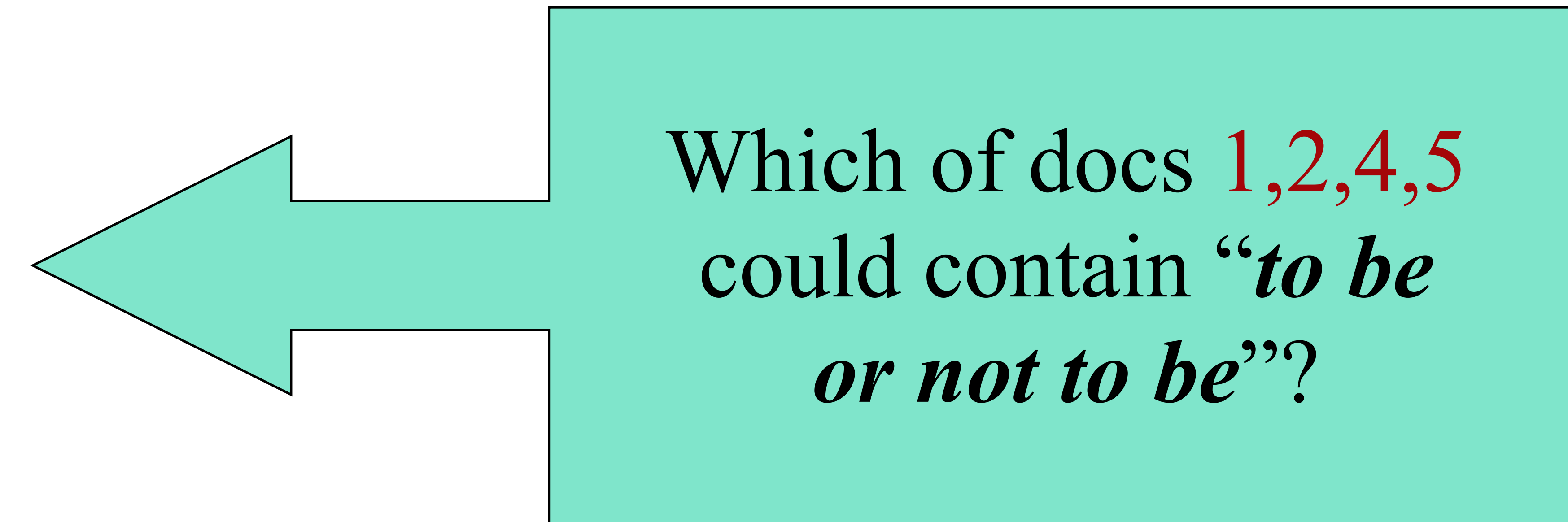
Problem: Phrases
Solution: Positional indexes

- In the inverted inded, store, for each *term* the position(s) in which it appears:

  - <***term***, number of docs containing ***term***;
  - *doc1*: position1, position2 … ;
  - *doc2*: position1, position2 … ;
  - etc.>

Positional Index Example

<***be***: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>



Processing a Phrase Query

- Extract inverted index entries for each distinct term: *to, be, or, not.*
- Merge their *doc:position* lists to enumerate all positions with "*to be or not to be*".
  - *to:*
    - *2*:1,17,74,222,551; *4*:8,16,190,429,433; *7*:13,23,191; ...
  - *be:*
    - *1*:17,19; *4*:17,191,291,430,434; *5*:14,19,101; ...

- Same general method for proximity searches
  - "LIMIT! /3 STATUTE /3 FEDERAL /2 TORT"
    - /*k* means "within *k* words of"
  - Positional indexes can be used for such queries; phrase indexes cannot.

Positional Index Size

TINFL

1. A positional index expands postings storage *substantially*
    1. Even though indices can be compressed
    2. Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries
2. Need an entry for each occurrence, not just once per document
    1. Index size depends on average document size and average frequency of each term
        1. Average web page has <1000 terms
        2. SEC filings, books, even some epic poems … easily 100,000 terms
3. Rule of Thumb
    1. A positional index is 2–4 as large as a non-positional index
    2. Positional index size 35–50% of volume of original text