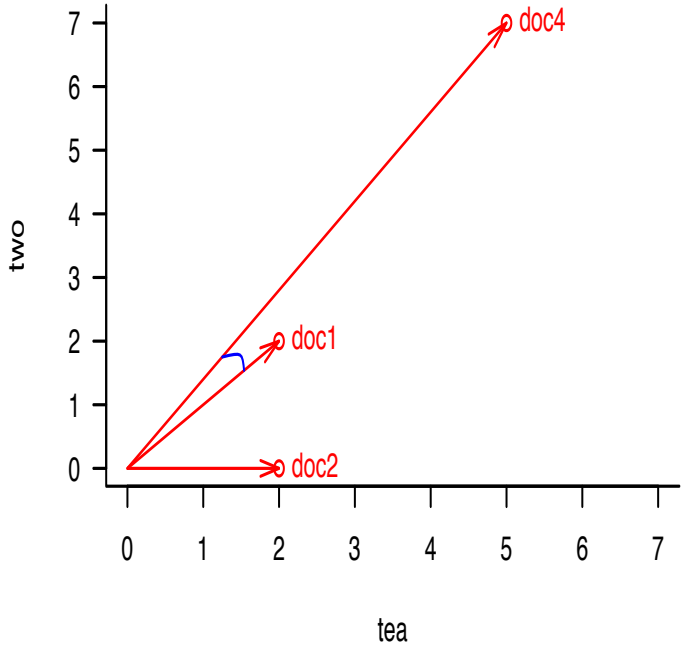


COMP90042 LECTURE 15

INFORMATION RETRIEVAL: BOOLEAN QUERYING & THE VECTOR SPACE MODEL

OVERVIEW

- Concepts of the *Term-Document Matrix* and *inverted index*
- Retrieval with boolean queries
- Vector space measure of query-document similarity

THE PROBLEM OF INFO. RETRIEVAL

- Given a large document collection and a query string
  - which of the documents *are relevant* to the query?
  - which ones are the *most relevant*?
- Raises questions:
  - how to model query-document relevance?
  - how can we service queries efficiently without enumerating all the documents for each query?
  - how can we evaluate effectiveness.

DOCUMENT REPRESENTATION

- Assume each document is a *bag of words,* i.e., discarding word order information, just recording counts
- The whole collection could be modelled as a *list of bag of words*
  - but this doesn't allow efficient access, e.g., to find a specific word
- Solution: the *term-document* matrix
  - rows represent *documents*
  - columns represent *terms* which are typically *word types*
  - matrix cells might be binary indicators, frequency counts or some other kind of 'score' attached to a word and a document

TERM-DOCUMENT MATRIX

doc1	Two for tea and tea for two
doc2	Tea for me and tea for you
doc3	You for me and me for you

	two	tea	me	you
doc1	2	2	0	0
doc2	0	2	1	1
doc3	0	0	2	2

POSTINGS LISTS

- For the example, let's reduce the matrix to binary to reflect word inclusion
- And transpose the matrix, so rows are now terms
  - the rows are called *postings lists* and identify the documents in which the term occurs

	doc1	doc2	doc3
two	1	0	0
tea	1	1	0
me	0	1	1
you	0	1	1

  - queries performed over posting lists

SIMPLE BOOLEAN QUERYING

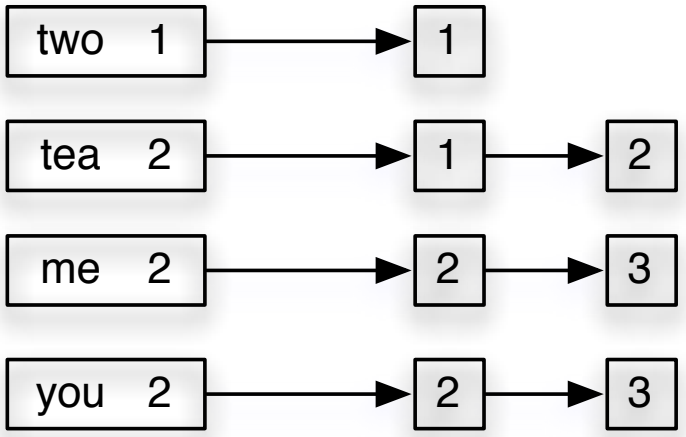
- Consider querying for *tea* **and** *me*
  - *tea* has postings [1,1,0], i.e., occurs in {doc1, doc2}
  - *me* has postings [0,1,1], i.e., occurs in {doc2, doc3}
  - to find the conjunction intersect these results to get [0,1,0] = {doc2}
- Other boolean operations easily supported
  - conjunction = bitwise AND
  - disjunction = bitwise OR
  - negation = bitwise complement

EFFICIENCY OF BINARY TDM

- For realistic sized corpora, posting lists can be very large (e.g., 1M documents) and there can be many of them (e.g., 500k term vocabulary)
- Representing in efficient and compact format essential
- The term-document matrix is extremely sparse
  - most terms occur in only a few documents
  - therefore, use a sparse encoding of posting lists
- Namely the inverted index

INVERTED INDEX

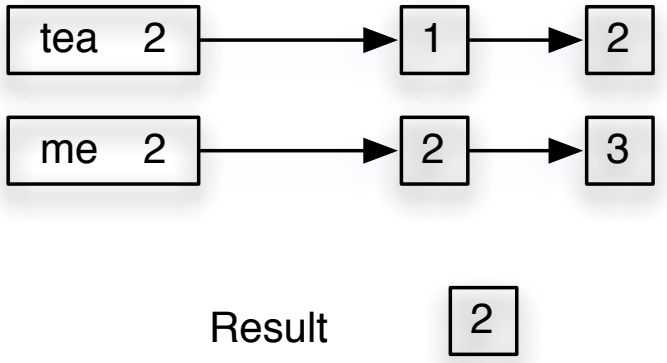
- Store only the indexes of the 1s in each posting list



- Known as an *inverted index*
- Lists are sorted by *document identifier* and stored using a variable length sequence, e.g., linked list

QUERYING AN INVERTED INDEX

- Queries now performed over posting lists



- Conjunction queries need common elements from the sorted sequences
- Can be done iteratively in O(x + y) time, where x and y are the lengths of the posting lists
- Generally for more than two part queries, the order can effect runtime

INVERTED INDEX CONSTRUCTION

Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

⟹

term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

⟹

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Step 1: tokenize and collect term occurences.
Step 2: sort by term.
Step 3: collate into posting lists and doc. freq.
Figure from Manning et al, IIR.

STORAGE COSTS

- Storage requirements
  - terms and counts, |T| entries
  - pointers to postings, |T| entries
  - list of docIDs, |T| x s entries
- Majority of terms have few occurrences, *s* is small (Zipf's law)
- Worst case for stop words, s ≈ |D|
- Long posting lists lead to poor query runtime

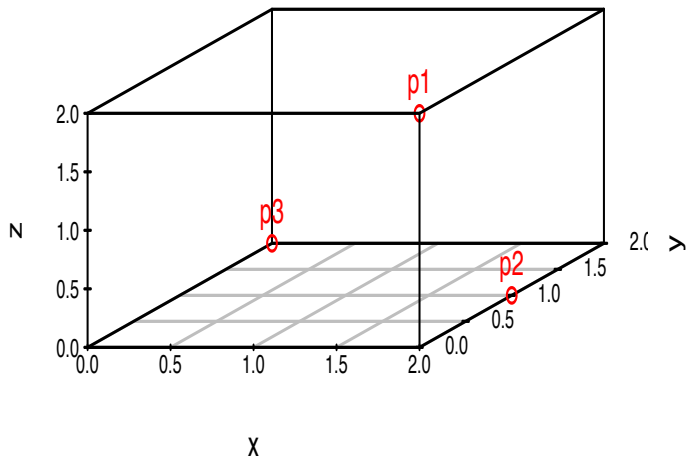
RANKED RETRIEVAL: USING FREQUENCIES

- Binary retrieval is great for some problems but often want *ranked* retrieval outputs
  - based on how *'close'* each document is to the query
- How to measure closeness?
- Let's start by revisiting the TDM

GEOMETRICAL VIEW OF THE TDM

- The TDM not just a useful document representation
  - also suggests a useful way of modelling documents
  - consider documents as *points (vectors)* in a *multi-dimensional term space*
- E.g., points in 3d

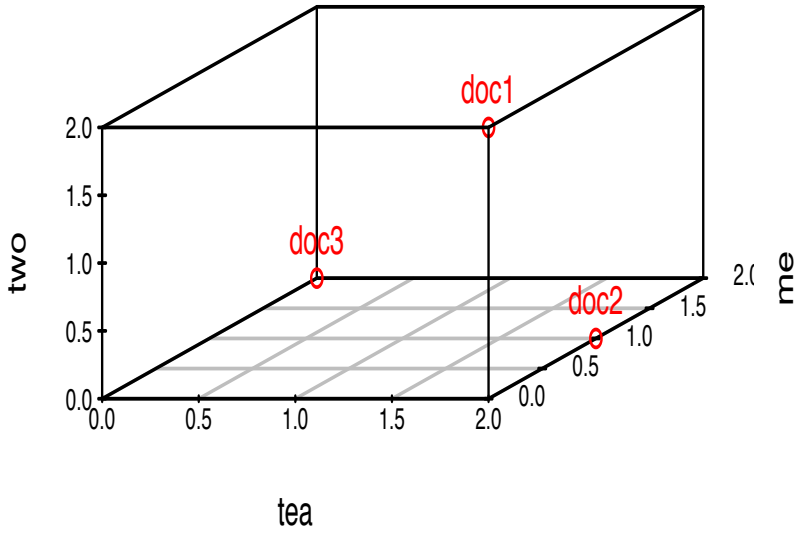
Point	x	y	z
p1	2	0	2
p2	2	1	0
p3	0	2	0



DOCUMENTS IN TERM SPACE

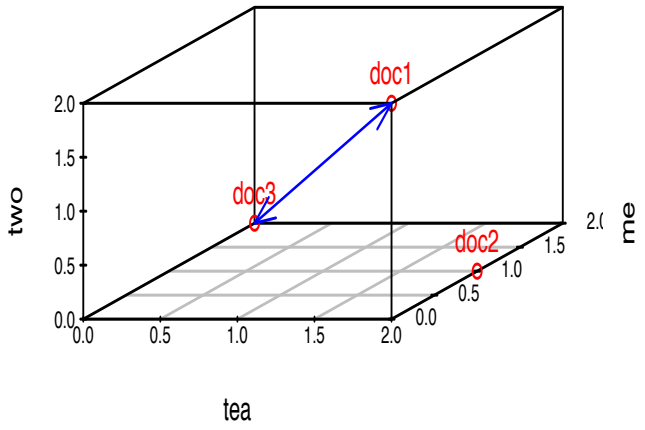
- Each of the |T| terms becomes a dimension
- Documents are points, with the value for each dimension determined by the score $s_{d,t}$
  - this might be the frequency of term t in document d

Point	tea	me	two
doc1	2	0	2
doc2	2	1	0
doc3	0	2	0

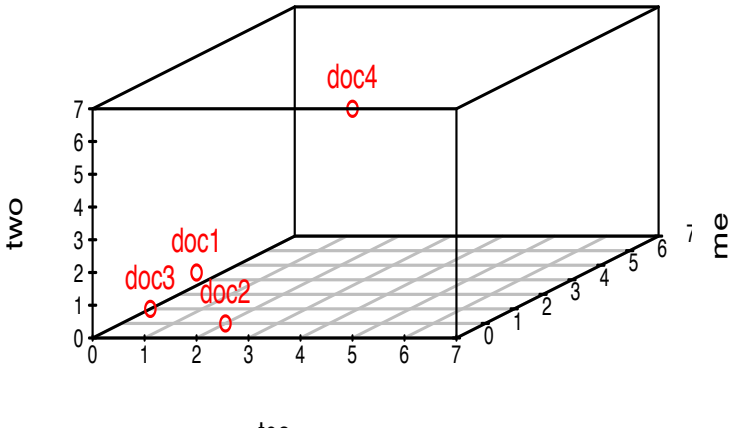


HOW TO MEASURE SIMILARITY

- We can now attempt to measure similarity between documents
  - Euclidean distance?
  - consider what happens for a long document, with many words



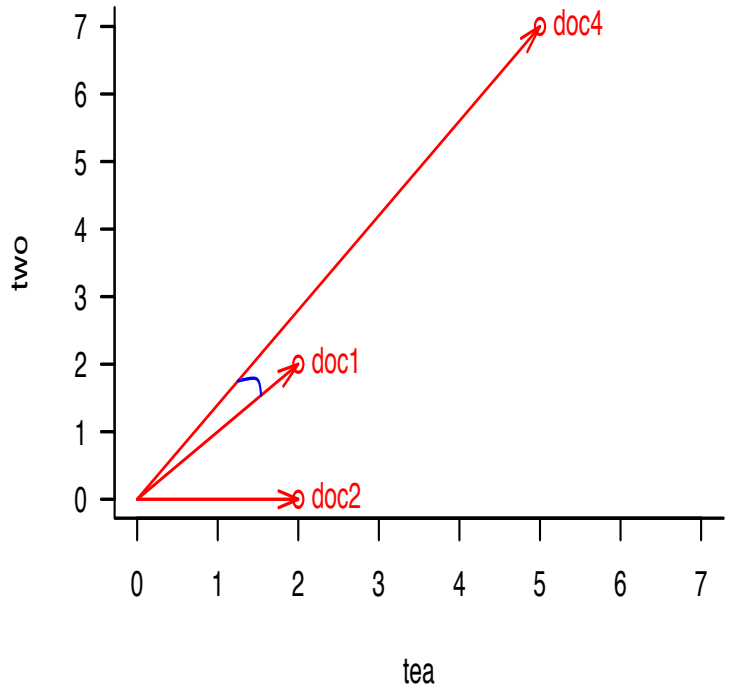
Point	tea	me	two
doc1	2	0	2
doc2	2	1	0
doc3	0	2	0
doc4	5	0	7



ANGULAR SIMILARITY

- To account for length bias consider documents as *vectors*
  - and measure the angle between them

Point	tea	two
doc1	2	2
doc2	2	0
doc4	5	7



COSINE DISTANCE

- Given two vectors, **a** and **b**, the cosine between them is

$$\cos(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a}.\mathbf{b}}{|\mathbf{a}| \times |\mathbf{b}|}$$

- where . is the vector dot product, i.e., $\mathbf{a}.\mathbf{b} = \sum_{i=1}^{|T|} a_i b_i$
- and $|\mathbf{a}|$ denotes the vector magnitude $|\mathbf{a}| = \sqrt{\sum_{i=1}^{|T|} a_i^2}$

SPEEDING UP COSINE DISTANCE

- All documents pre-normalised to unit length
  (or normalisation factors precomputed and stored)
- Cosine then just requires a dot product **a.b**
  - but vectors (like posting lists) are sparse, with many entries equal to zero
  - only need to consider non-zero entries in both **a** and **b**
  - use inverted index to reflect sparsity, and allow efficient storage and querying
- This is known as *the vector space model*

TF*IDF

- Scores in TDM typically combine two elements, $tf_{d,t} \times idf_t$
  - The *term frequency* or TF, based on the occurrence count for the term in a document
  - The *inverse document frequency* or IDF, based on how rare the word is, and therefore how informative an occurrence will be
  - IDF typically formulated as $\text{idf}_t = \log \frac{N}{\text{df}_t}$ where N is the number of documents and df_t the number of documents containing t
- Consider extrema for *idf*, rare words vs. stop-words

TWEAKING TF*IDF

- Many variant formulations for term weighting
  - raw term frequency, $f_{t,d}$, versus $log\ (1 + f_{t,d})$
  - various *idf* definitions, or leave out
  - whether to include document length normalisation
- Choices are mostly heuristic
  - Zobel and Moffat, "*Exploring the Similarity Space*" (1998) identify $(8 \times 9 \times 2 \times 6 \times 14) = 12096$ possible different combinations of choices
- Typically try several options to evaluate which is most effective

SUMMARY

- Concept of the term-document matrix and inverted index
- Efficient data structures and query methods for boolean queries
- Introduced the vector space model, similarity and scoring methods
- Reading
  - Chapter 1, “Boolean Retrieval” of Manning, Raghavan, and Schutze, Introduction to Information Retrieval
  - Chapter 6, “Scoring, term weighting & the vector space model” of Manning, Raghavan, and Schutze, Introduction to Information Retrieval