

Information Retrieval 101

Hal Daumé III

School of Computing
University of Utah

me@hal3.name



Query

- Which plays of Shakespeare contain the words ***Brutus*** ***AND Caesar*** but ***NOT Calpurnia***?
- Could grep all of Shakespeare's plays for ***Brutus*** and ***Caesar***, then strip out lines containing ***Calpurnia***?
 - Slow (for large corpora)
 - ***NOT Calpurnia*** is non-trivial
 - Other operations (e.g., find the phrase ***Romans and countrymen***) not feasible

Term-document incidence

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

1 if **play** contains
word, 0 otherwise

Incidence vectors

- So we have a 0/1 vector for each term.
- To answer query: take the vectors for ***Brutus, Caesar*** and ***Calpurnia*** (complemented) ➤ bitwise *AND*.
- $110100 \text{ AND } 110111 \text{ AND } 101111 = 100100$.

Answers to query

➤ Antony and Cleopatra, Act III, Scene ii

- *Agrippa* [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus,
- When Antony found Julius **Caesar** dead,
- He cried almost to roaring; and he wept
- When at Philippi he found **Brutus** slain.

➤ Hamlet, Act III, Scene ii

- *Lord Polonius*: I did enact Julius **Caesar** I was killed i' the
- Capitol; **Brutus** killed me.

Bigger document collections

- Consider $N = 1$ million documents, each with about 1K terms.
- Avg 6 bytes/term incl spaces/punctuation
 - 6GB of data in the documents.
- Say there are $M = 500K$ *distinct* terms among these.

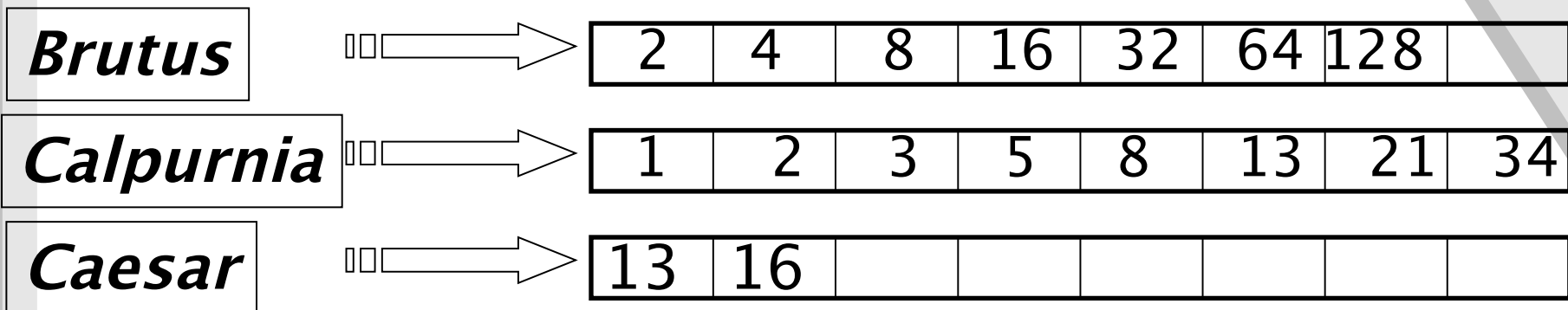
Can't build the matrix

- 500K x 1M matrix has half-a-trillion 0's and 1's.
- But it has no more than one billion 1's.
 - matrix is extremely sparse.
- What's a better representation?
 - We only record the 1 positions.



Inverted index

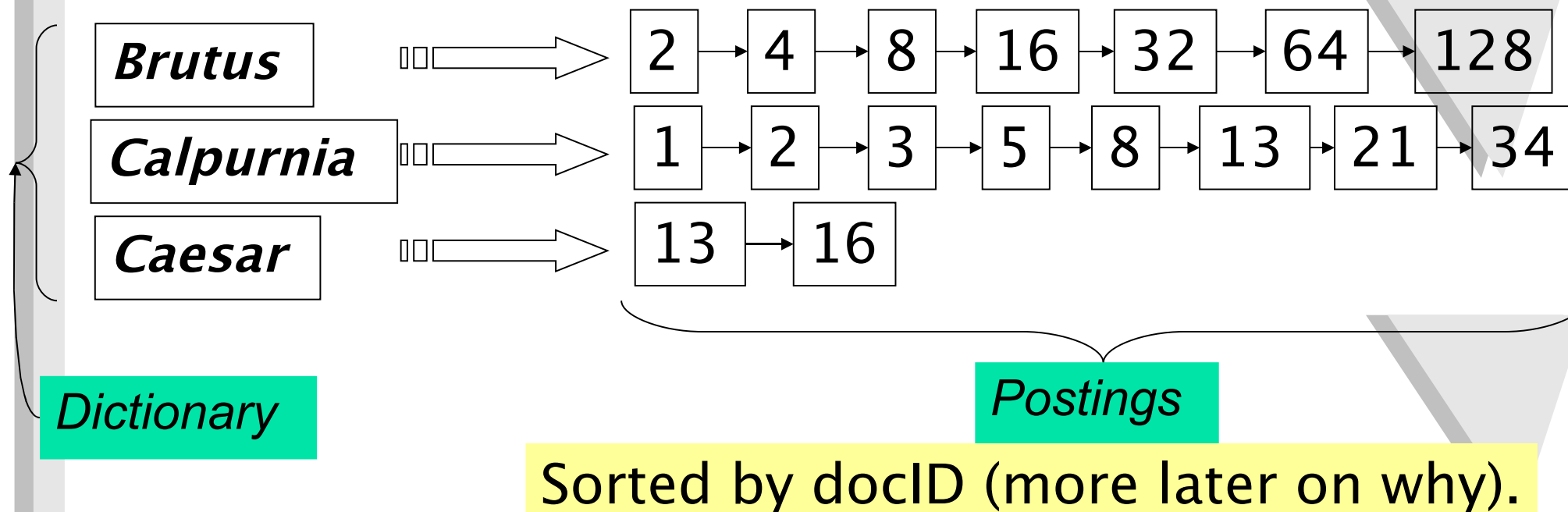
- For each term T : store a list of all documents that contain T .
- Do we use an array or a list for this?



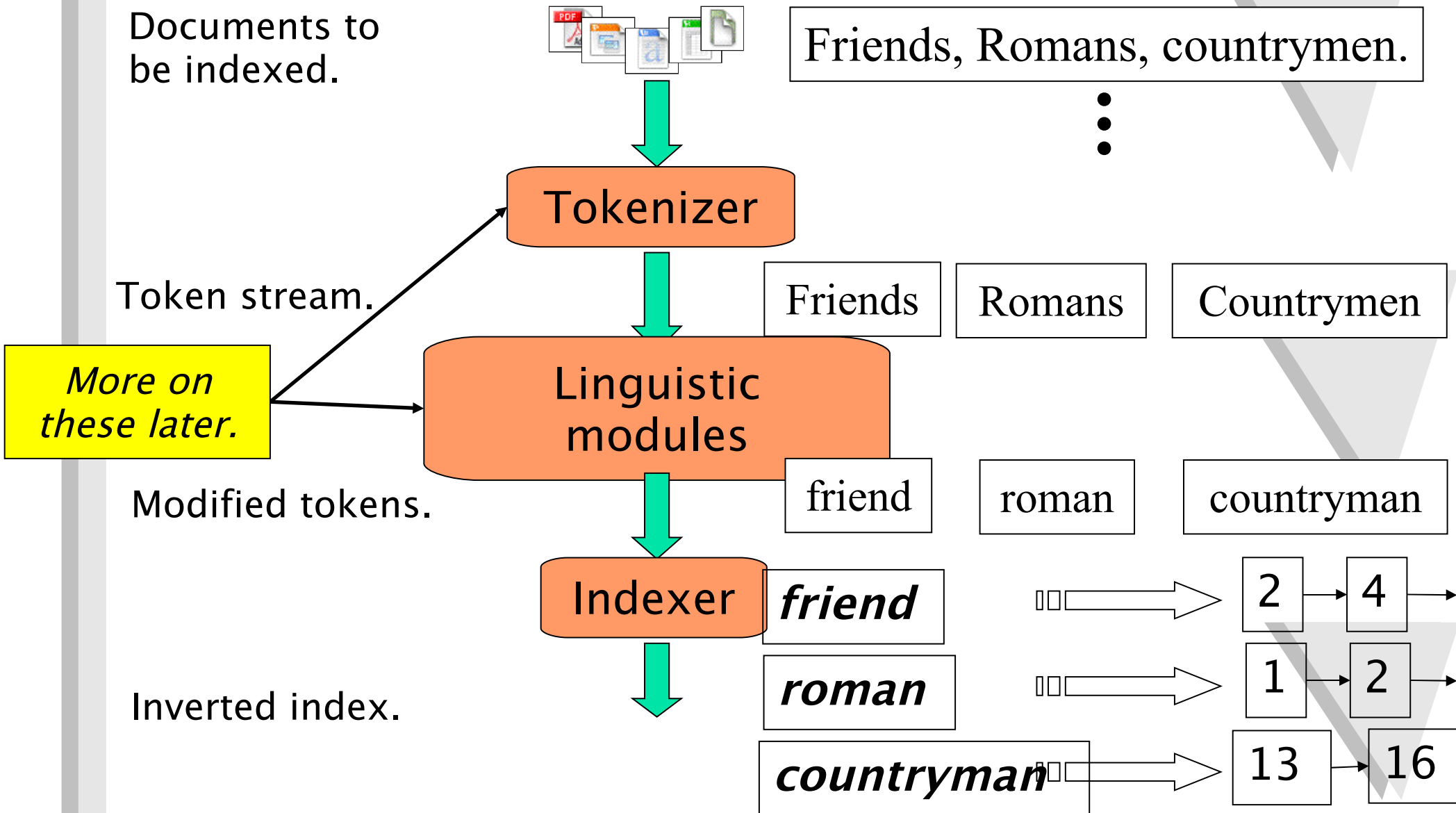
What happens if the word **Caesar** is added to document 14?

Inverted index

- Linked lists generally preferred to arrays
 - Dynamic space allocation
 - Insertion of terms into documents easy
 - Space overhead of pointers

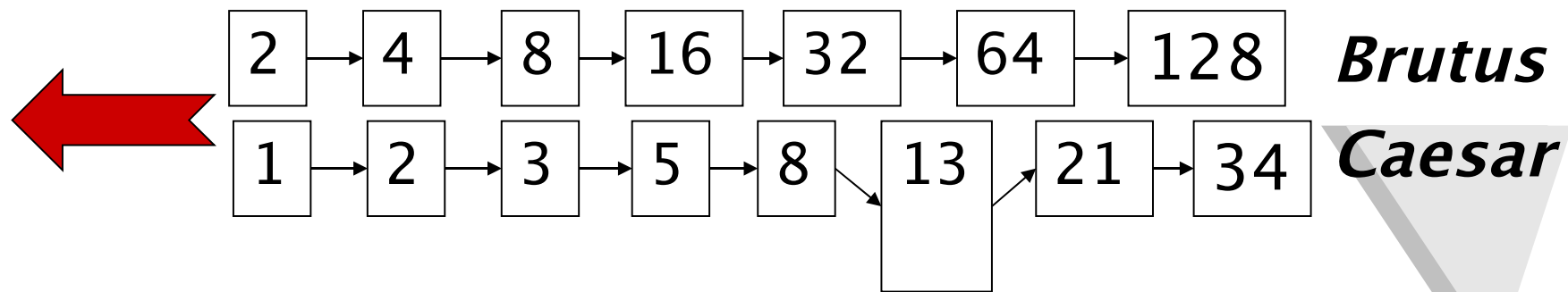


Inverted index construction



Query processing

- Consider processing the query:
Brutus AND Caesar
- Locate ***Brutus*** in the Dictionary;
 - Retrieve its postings.
- Locate ***Caesar*** in the Dictionary;
 - Retrieve its postings.
- “Merge” the two postings:



Basic postings intersection

```
INTERSECT( $p_1, p_2$ )
1   $answer \leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $docID(p_1) = docID(p_2)$ 
4      then  $\text{ADD}(answer, docID(p_1))$ 
5           $p_1 \leftarrow next(p_1)$ 
6           $p_2 \leftarrow next(p_2)$ 
7      else if  $docID(p_1) < docID(p_2)$ 
8          then  $p_1 \leftarrow next(p_1)$ 
9          else  $p_2 \leftarrow next(p_2)$ 
10 return  $answer$ 
```

► **Figure 1.7** Algorithm for the intersection of two postings lists p_1 and p_2 .

Stemming

- Reduce terms to their “roots” before indexing
 - language dependent
 - e.g., *automate(s)*, *automatic*, *automation* all reduced to *automat*.

for example compressed and compression are both accepted as equivalent to compress.

for exampl compres and compres are both accept as equival to compres.

Porter's algorithm

- Commonest algorithm for stemming English
- Conventions + 5 phases of reductions
 - phases applied sequentially
 - each phase consists of a set of commands
 - sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*

Typical rules in Porter

- *sses* → *ss*
- *ies* → *i*
- *ational* → *ate*
- *tional* → *tion*

Other stemmers

- Other stemmers exist, e.g., Lovins stemmer
<http://www.comp.lancs.ac.uk/computing/research/stemming/general/lovins.htm>
- Single-pass, longest suffix removal (about 250 rules)
- Motivated by Linguistics as well as IR
- Full morphological analysis - modest benefits for retrieval

Stemming variants

Sample text: Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

Lovins stemmer: such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

Porter stemmer: such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

Paice stemmer: such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

► **Figure 2.7** A comparison of three stemming algorithms on a sample text.

Term frequency vectors

- Consider the number of occurrences of a term t in a document d , denoted $tf_{t,d}$
 - Document is a vector: a column below
 - Bag of words model

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

Scores from term frequencies

- Given a free-text query q , define

$$\text{Score}(q,d) = \sum_{t \in q} \text{tf}_{t,d}$$

Simply add up the term frequencies of all query terms in the document

This assigns a score to each document; now rank-order documents by this score.

Bag of words view of a doc

- Thus the doc
 - *John is quicker than Mary.*
- is indistinguishable from the doc
 - *Mary is quicker than John.*

Adding frequencies

- Consider query *ides of march*
 - *Julius Caesar* has 5 occurrences of *ides*
 - No other play has *ides*
 - *march* occurs in over a dozen
 - All the plays contain *of*
- By this scoring measure, the top-scoring play is likely to be the one with the most *of*s

Digression: terminology

- WARNING: In a lot of IR literature, “frequency” is used to mean “count”
- Thus *term frequency* in IR literature is used to mean *number of occurrences* in a doc
- Not divided by document length (which would actually make it a frequency)
- **We will conform to this misnomer**
- In saying term frequency we mean the number of occurrences of a term in a document.

Term frequency $tf_{t,d}$

- Long docs are favored because they're more likely to contain query terms
- Can fix this to some extent by normalizing for document length
- But is raw $tf_{t,d}$ the right measure?

Weighting term frequency: *tf*

- What is the relative importance of
 - 0 vs. 1 occurrence of a term in a doc
 - 1 vs. 2 occurrences
 - 2 vs. 3 occurrences ...
- Unclear: while it seems that more is better, a lot isn't proportionally better than a few
 - Can just use raw *tf*
 - Another option commonly used in practice:

$$wf_{t,d} = 0 \text{ if } tf_{t,d} = 0, 1 + \log tf_{t,d} \text{ otherwise}$$

Weighting should depend on the term overall

- Which of these tells you more about a doc?
 - 10 occurrences of *hernia*?
 - 10 occurrences of *the*?
- Would like to attenuate the weights of *common terms*
 - But what is “common”?
- Suggestion: look at collection frequency (*cf*)
 - The total number of occurrences of the term in the entire collection of documents

Document frequency

- But document frequency (df) may be better:
- df = number of docs in the corpus containing the term

Word	cf	df
<i>try</i>	10422	8760
<i>insurance</i>	10440	3997

- Document/collection frequency weighting is only possible in known (static) collection.
- So how do we make use of df ?

tf x idf term weights

- tf x idf measure combines:
 - term frequency (tf)
 - or wf , some measure of term density in a doc
 - inverse document frequency (idf)
 - measure of informativeness of a term: its rarity across the whole corpus
 - could just be raw count of number of documents the term occurs in ($idf_t = 1/df_t$)
 - but by far the most commonly used version is:
- See Papineni, NAACL 2, 2002 for theoretical justification

$$idf_t = \log\left(\frac{N}{df_t}\right)$$

Reuters RCV1 800K docs

- Logarithms are base 10

term	df_t	idf_t
car	18,165	1.65
auto	6723	2.08
insurance	19,241	1.62
best	25,235	1.5

Summary: tf x idf (or tf.idf)

- Assign a tf.idf weight to each term i in each document d

$$w_{t,d} = tf_{t,d} \times \log(N / df_t)$$

*What is the wt
of a term that
occurs in all
of the docs?*

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

N = total number of documents

df_t = the number of documents that contain term t

- Increases with the number of occurrences *within* a doc
- Increases with the rarity of the term *across* the whole corpus

Real-valued term vectors

- Still Bag of words model
- Each is a vector in \mathbb{R}^M
 - Here log-scaled *tf.idf*

Note can be > 1!

	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

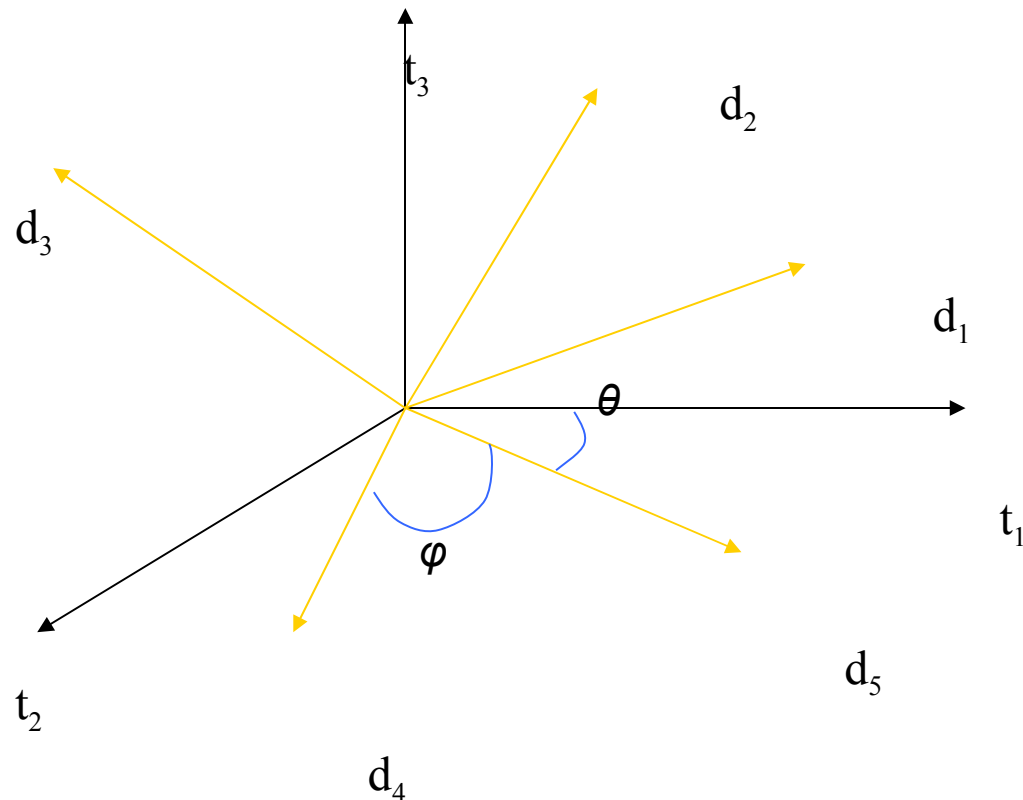
Documents as vectors

- Each doc j can now be viewed as a vector of $tf \times idf$ values, one component for each term
- So we have a vector space
 - terms are axes
 - docs live in this space
 - even with stemming, may have 20,000+ dimensions

Why turn docs into vectors?

- First application: Query-by-example
 - Given a doc D , find others “like” it.
 - Now that D is a vector, find vectors (docs) “near” it.

Intuition



Postulate: Documents that are “close together” in the vector space talk about the same things.

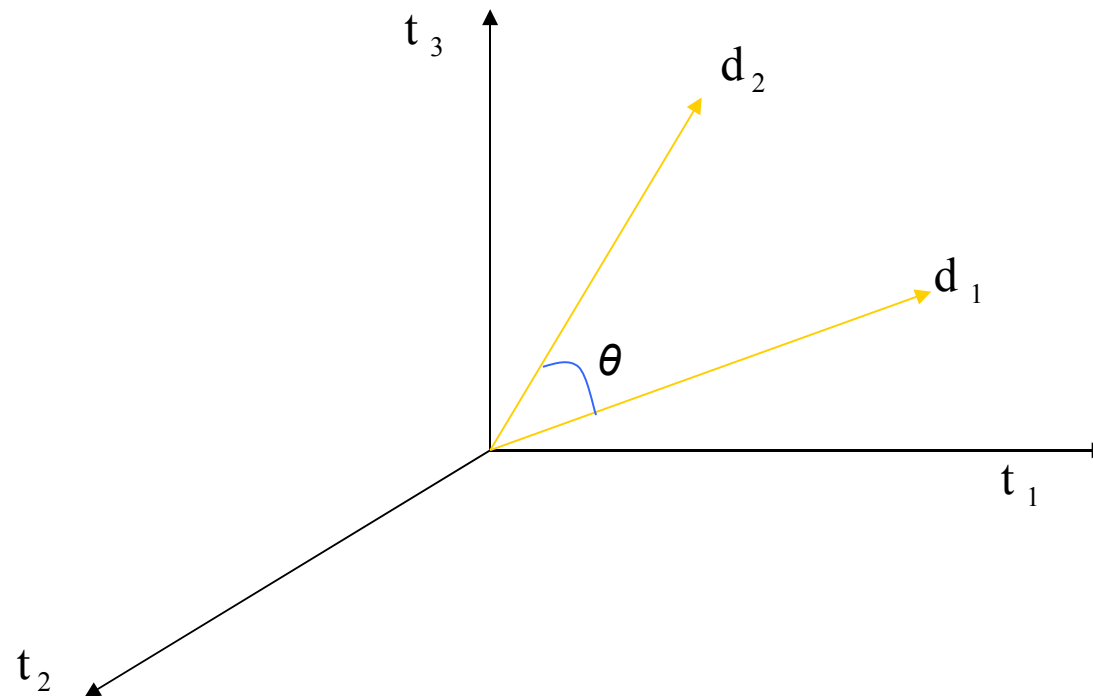
The vector space model

Freetext query as vector:

- We regard freetext query as short document
- We return the documents ranked by the closeness of their vectors to the query vector.

Cosine similarity

- Distance between vectors d_1 and d_2 captured by the cosine of the angle x between them.
- Note – this is *similarity*, not distance
- No triangle inequality for similarity.



Cosine similarity

- A vector can be *normalized* (given a length of 1) by dividing each of its components by its length – here we use the L_2 norm
- This maps vectors onto the unit sphere:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_i x_i^2}$$

- Then, $|d_j| = \sqrt{\sum_{i=1}^M w_{i,j}} = 1$

- Longer documents don't get more weight

Cosine similarity

$$\text{sim}(d_j, d_k) = \frac{d_j \cdot d_k}{\|d_j\| \|d_k\|} = \frac{\sum_{i=1}^M w_{i,j} w_{i,k}}{\sqrt{\sum_{i=1}^M w_{i,j}^2} \sqrt{\sum_{i=1}^M w_{i,k}^2}}$$

- Cosine of angle between two vectors
- The denominator involves the lengths of the vectors.



Normalization

Example

- Docs: Austen's *Sense and Sensibility*, *Pride and Prejudice*; Bronte's *Wuthering Heights*

	SaS	PaP	WH
<i>affection</i>	115	58	20
<i>jealous</i>	10	7	11
<i>gossip</i>	2	0	6

	SaS	PaP	WH
<i>affection</i>	0.996	0.993	0.847
<i>jealous</i>	0.087	0.120	0.466
<i>gossip</i>	0.017	0.000	0.254

- $\cos(\text{SAS}, \text{PAP}) = .996 \times .993 + .087 \times .120 + .017 \times 0.0 = 0.999$
- $\cos(\text{SAS}, \text{WH}) = .996 \times .847 + .087 \times .466 + .017 \times .254 = 0.929$

Basic cosine score computation

COSINESCORE(q)

```

1  float  $Scores[N] = 0$ 
2  Initialize  $Length[N]$ 
3  for each query term  $t$ 
4  do
5      calculate  $w_{t,q}$  and fetch inverted list for  $t$ 
6      for each pair( $d, tf_{t,d}$ ) in inverted list
7      do
8          add  $w_{t,q} \times tf_{t,d}$  to  $Scores[d]$ 
9          Read the array  $Length[d]$ 
10     for each  $d$ 
11     do Divide  $Scores[d]$  by  $Length[d]$ 
12     return Top  $K$  components of  $Scores[]$ 

```