Recap of last lecture

- Collection and vocabulary statistics: Heaps' and Zipf's laws
- Dictionary compression for Boolean indexes
  - Dictionary string, blocks, front coding
- Postings compression: Gap encoding, prefix-unique codes
  - Variable-Byte and Gamma codes

collection (text, xml markup etc)	3,600.0	MB
collection (text)	960.0	
Term-doc incidence matrix	40,000.0	
postings, uncompressed (32-bit words)	400.0	
postings, uncompressed (20 bits)	250.0	
postings, variable byte encoded	116.0	
postings, γ–encoded	101.0	

This lecture; IIR Sections 6.2-6.4.3

- Ranked retrieval
- Scoring documents
- Term frequency
- Collection statistics
- Weighting schemes
- Vector space scoring

Ranked retrieval

- So far, all queries have been Boolean.
  - Documents either match or don’t.
- Good for expert users with precise understanding of their needs and the collection.
  - Good if recall of query is > 0 and < 100.
  - 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 for web search.

Example: feast or famine

- Boolean queries often result in either too few (=0) or too many (1000s) results.
- Q1: "*standard user dlink 650*" → 200,000 hits
- Q2: "*standard user 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 a set of documents satisfying a query expression, in ranked retrieval models, the system returns an ordering over the (top) documents in the collection with respect to a query
- Free text queries: Rather than a query language of operators and expressions, the user’s query is just one or more words in a human language
- In principle, there are two separate choices here, but in practice, ranked retrieval models have normally been associated with free text queries and vice versa

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

- When a (Boolean) system produces no results
  - Still show the 10 closest results

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-document matching scores

- Need to assign a score to a query/document pair
- Let’s start with a one-term query
- If the query term does not occur in the document: score should be 0
- The more frequent the query term in the document, the higher the score (should be)
- We will look at a number of alternatives for this.

Take 1: Jaccard coefficient

- Recall from Lecture 3: A commonly used measure of overlap of two sets *A* and *B*
- jaccard$(A,B) = |A \cap B| \;/\; |A \cup B|$
- jaccard$(A,A) = 1$
- jaccard$(A,B) = 0$ if $A \cap B = 0$
- *A* and *B* don't have to be the same size.
- Always assigns a number between 0 and 1.

Jaccard coefficient: Scoring example

- What is the query-document match score that the Jaccard coefficient computes for each of the two documents below?
- Query: *ides of march*
- Document 1: *caesar died in march*
- Document 2: *the long march*

Issues with Jaccard for scoring

- It doesn't consider *term frequency*
  - Docs that talk a lot about a term should score higher
- It doesn't consider *rareness*
  - Rare terms carry more information than stop words
- We need a way of normalizing for length
  - How about?

$$|A \cap B| / \sqrt{|A \cup B|}$$

Recall (Lecture 1): Binary term-document incidence matrix

	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

Each document is represented by a binary vector $\in \{0,1\}^{|V|}$

Term-document count matrices

- Consider number of occurrences of a term in doc:
  - Each document is count vector in $\mathbb{N}^{v}$: a column below

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	1
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

Bag of words model

- Vector representation of a document
  - Each entry represents the frequency of a term
- Vector representation doesn't consider the ordering of words in a document
  - *"John is quicker than Mary"*
  - *"Mary is quicker than John"*
  - Both documents have the same vectors
- This is a step back: No phrase search possible.
  - Possible to combine with positional index.
  - (Not addressed in this course)

Term frequency tf

- The term frequency $\mathrm{tf}_{t,d}$ of term *t* in document *d* is defined as the number of times that *t* occurs in *d*.
- We want to use tf when computing query-document match scores. But how?
- Raw term frequency is not what we want:
  - A document with 10 occurrences of the term is more relevant than a document with 1 occurrence of the term.
  - But not 10 times more relevant.
- Relevance does not increase proportionally with term frequency.

Log-frequency weighting

- The log frequency weight of term t in d is

$$w_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{tf}_{t,d}, & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- $0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 1.3$, $10 \rightarrow 2$, $1000 \rightarrow 4$, etc.
- Score for a document-query pair:
  - sum over terms *t* in both *q* and *d*:

$$= \sum_{t \in q \cap d} (1 + \log \mathrm{tf}_{t,d})$$

- The score is 0 if none of the query terms is present in the document.

Document frequency

- Rare terms more informative than frequent terms
  - Recall stop words
- Consider a term in the query that is rare in the collection (e.g., *arachnocentric*)
- A document containing this term is very likely to be relevant to the query *arachnocentric*
- → We want a high weight for rare terms like *arachnocentric*.

idf weight

- df_t (document frequency):
  - the number of documents that contain term *t*
  - df_t is an inverse measure of the informativeness of *t*
  - $\mathrm{df}_t \leq N$
- idf_t (inverse document frequency)

$$\mathrm{idf}_t = \log_{10} (N/\mathrm{df}_t)$$

  - Use log to “dampen” the effect.
  - (Don’t worry about base of log.)
- There is one df_t and idf_t value per term in corpus.
  - Not query or document specific!

idf example, suppose N = 1 million

term	df_t	idf_t
calpurnia	1	6
animal	100	4
sunday	1,000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0

$$\mathrm{idf}_t = \log_{10} \left(N/\mathrm{df}_t \right)$$

Effect of idf on ranking

- No effect for queries with a single term
- Queries with more than one term:
  - E.g. capricious person
  - idf weighting makes occurrences of capricious count for much more than occurrences of person.

tf.idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$w_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \times \log_{10}(N / \mathrm{df}_t)$$

- Best known weighting scheme in IR
  - Alternative names: tf-idf, tf x idf
- Increases with #occurrences within a document
- Increases with rarity of the term in the collection

Binary → count → weight matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	5.25	3.18	0	0	0	0.35
Brutus	1.21	6.1	0	1	0	0
Caesar	8.59	2.54	0	1.51	0.25	0
Calpurnia	0	1.54	0	0	0	0
Cleopatra	2.85	0	0	0	0	0
mercy	1.51	0	1.9	0.12	5.25	0.88
worser	1.37	0	0.11	4.15	0.25	1.95

Each document is now represented by a real-valued vector of tf.idf weights $\in \mathbb{R}^{|V|}$

Final ranking of documents for a query

$$\mathrm{Score}(q,d) = \sum_{t \in q \cap d} \mathrm{tf.idf}_{t,d}$$

Documents as vectors

- So we have a |V|-dimensional vector space
- Terms are axes of the space
- Documents are points or vectors in this space
- Very high-dimensional: tens of millions of dimensions (i.e., terms) on the web
- These are sparse vectors: most entries are zero.

Queries as vectors

- <u>Idea 1:</u>
  - Represent queries as vectors in the space
- <u>Idea 2:</u>
  - Rank docs according to <u>*proximity*</u> to the query in space
- proximity = similarity of vectors
- proximity ≈ inverse of distance
- Recall: We do this because we want to get away from the you're-either-in-or-out Boolean model.
- Instead: rank more relevant documents higher than less relevant documents

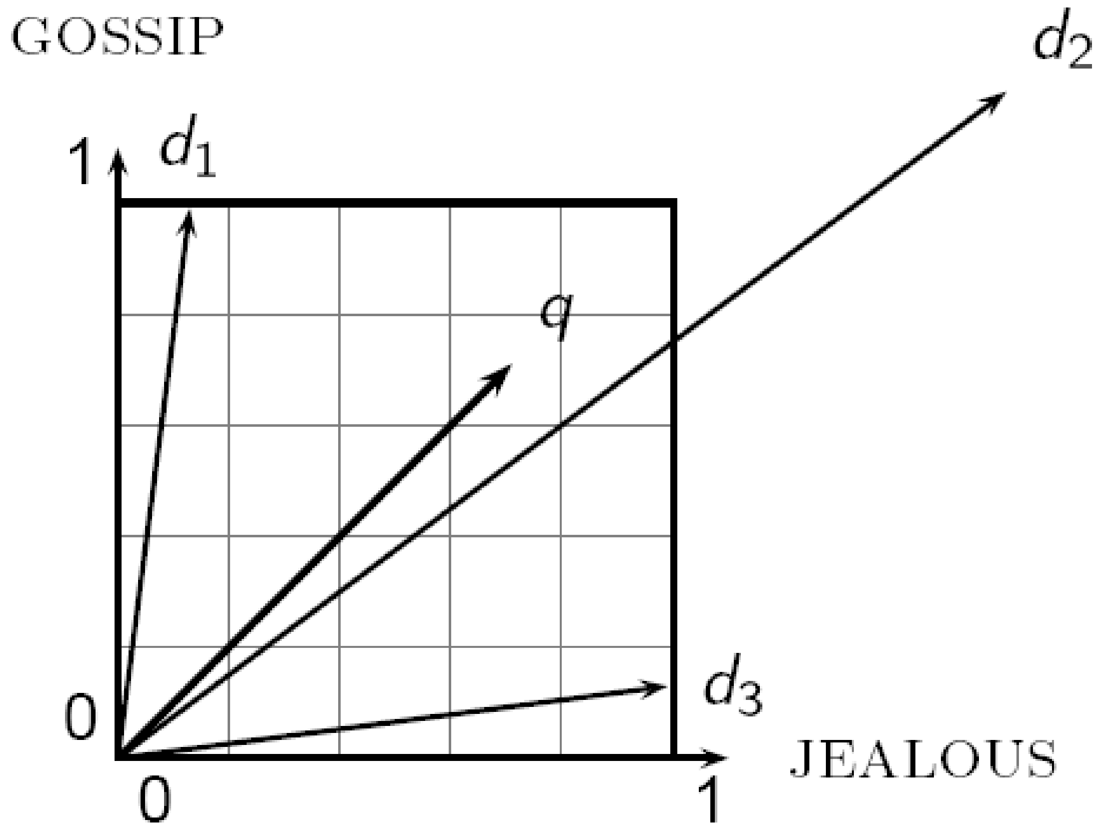
Formalizing vector space proximity

- First cut: distance between two points
- Euclidean distance?
  - Euclidean distance is a bad idea because Euclidean distance is large for vectors of different lengths.

Why Eucledian distance is a bad idea

q and d_2 are similar
but have large distance

q and d_1 are close
but are not similar



Use angle instead of distance

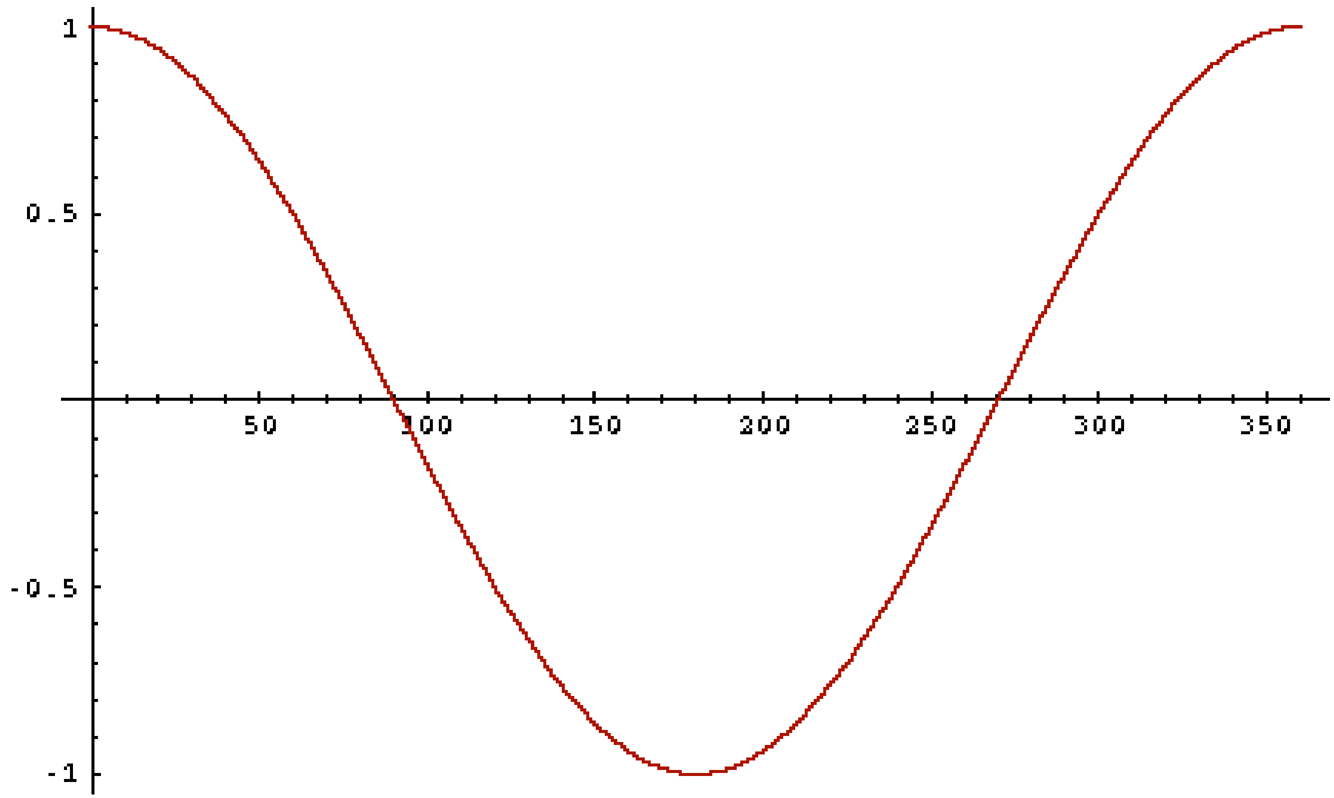
- Thought experiment: take a document *d* and append it to itself. Call this document *d*′.
- "Semantically" d and d′ have the same content
- The Euclidean distance between d and d′ is large
- The angle between d and d′ is 0
  - corresponding to maximal similarity.

- Idea: Rank docs according to angle with query.

From angles to cosines

- The following two notions are equivalent.
  - Rank documents in decreasing order of the angle between query and document
  - Rank documents in increasing order of cosine(query,document)
- Cosine is a monotonically decreasing function for the interval [0^o, 180^o]

From angles to cosines



- But how – *and why* – should we be computing cosines?
- Why are we only interested in the range 0 – 90 degrees?

Length normalization

- A vector can be (length-) normalized by dividing each of its components by its length – for this we use the L_2 norm: $$\left\|\vec{x}\right\|_2 = \sqrt{\sum_i x_i^2}$$
- Dividing a vector by its L_2 norm makes it a unit (length) vector (on surface of unit hypersphere)
- Effect on the two docs d and d′(d′ = d d)
  - they have identical vectors after normalization.
  - long and short docs have comparable weights

cosine(query,document)

Dot product

Unit vectors

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q}\bullet\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\vec{q}}{|\vec{q}|}\bullet\frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\,\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

q_i is the tf.idf weight of term *i* in the query
d_i is the tf.idf weight of term *i* in the document

cos(*q*,*d*) is the cosine similarity of *q* and *d* … or, equivalently, the cosine of the angle between *q* and *d*.

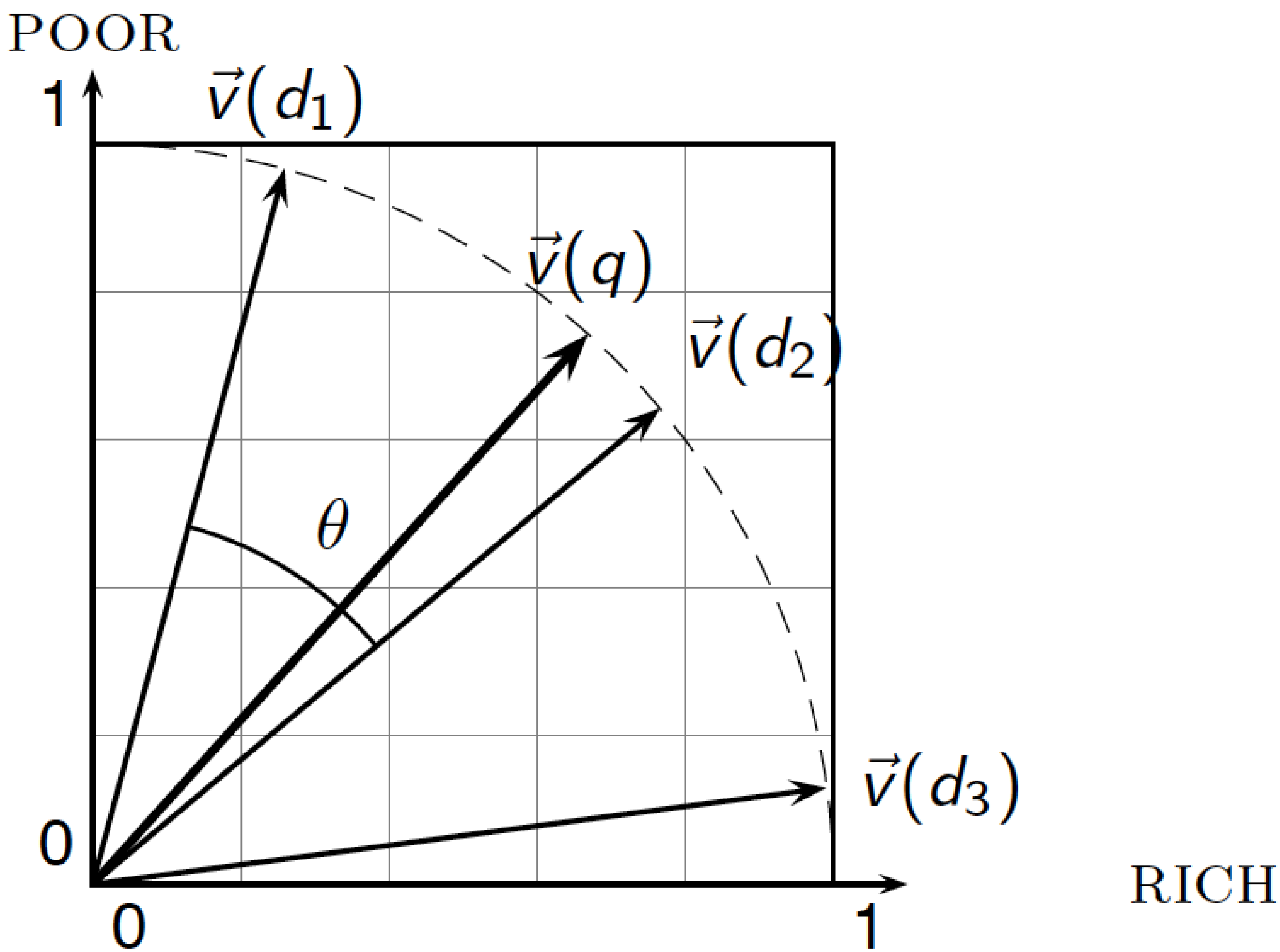
Cosine for length-normalized vectors

- For length-normalized vectors, cosine similarity is simply the dot product (or scalar product):

$$\cos(\vec{q},\vec{d}) = \vec{q} \bullet \vec{d} = \sum_{i=1}^{|V|} q_i d_i$$

for q, d length-normalized.

Cosine similarity illustrated



Looking at extreme cases

- Query = Doc; $q_{rich} = d_{rich}$; $q_{poor} = d_{poor}$
  - $q_r * d_r + q_p * d_p = q_r * q_r + q_p * q_p = 1$
- Query != Doc; $q_{rich} = 1$; $d_{rich} = 0$; $q_{poor} = 0$; $d_{poor} = 1$
  - $q_r * d_r + q_p * d_p = 0$
- (Exact math is a bit more involved, but works!)

Cosine similarity amongst 3 documents

How similar are these novels?

- SaS: *Sense and Sensibility*
- PaP: *Pride and Prejudice*
- WH: *Wuthering Heights*?

term	SaS	PaP	WH
affection	115	58	20
jealous	10	7	11
gossip	2	0	6
wuthering	0	0	38

Term frequencies (counts)

Note: To simplify this example, no idf weighting.

3 documents example contd.

Log frequency weighting

term	SaS	PaP	WH
affection	3.06	2.76	2.30
jealous	2.00	1.85	2.04
gossip	1.30	0	1.78
wuthering	0	0	2.58

After length normalization

term	SaS	PaP	WH
affection	0.789	0.832	0.524
jealous	0.515	0.555	0.465
gossip	0.335	0	0.405
wuthering	0	0	0.588

cos(SaS,PaP) ≈ 0.789 × 0.832 + 0.515 × 0.555 + 0.335 × 0.0 + 0.0 × 0.0 ≈ 0.94
cos(SaS,WH) ≈ 0.79
cos(PaP,WH) ≈ 0.69

Is there some deeper truth here?

Computing cosine scores

CosineScore(q)

float $Scores[N] = 0$
float $Length[N]$
for each query term t
do calculate $\mathrm{w}_{t,q}$ and fetch postings list for t
    for each pair$(d, \mathrm{tf}_{t,d})$ in postings list
    do $Scores[d] += \mathrm{w}_{t,d} \times \mathrm{w}_{t,q}$
Read the array *Length*
for each d
do $Scores[d] = Scores[d] / Length[d]$
return Top K components of $Scores[]$

What about performance?

- Boolean Retrieval is faster
  - docID comparison cheaper than multiplication
  - Skip Lists do not work here! Why?
  - If 10 results needed, can stop early;
    ranked retrieval needs to look at all postings – why?
- Boolean Retrieval needs less space
  - Extra real-value for each posting
  - More difficult to compress
- But, usability aspects win!
  - Just as with phrase search

Tf.idf weighting has many variants

Term frequency		Document frequency		Normalization	
n (natural)	$\mathrm{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(\mathrm{tf}_{t,d})$	t (idf)	$\log \frac{N}{\mathrm{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2 + w_2^2 + \ldots + w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times \mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N - \mathrm{df}_t}{\mathrm{df}_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases} 1 & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1 + \log(\mathrm{tf}_{t,d})}{1 + \log(\mathrm{ave}_{t \in d}(\mathrm{tf}_{t,d}))}$				

Columns headed 'n' are acronyms for weight schemes.

Why is the base of the log in idf immaterial?

High-dimensional Vector Spaces

- The queries “cholera” and “john snow” are far from each other in vector space.
- How can the document “John Snow and Cholera” be close to both of them?
- Our intuitions for 2- and 3-dimensional space don't work in >10,000 dimensions.

Summary – vector space ranking

- Represent query as a weighted tf.idf vector
- Represent each doc as a weighted tf.idf vector
- Compute the cosine similarity score for the query vector and each document vector
- Rank documents with respect to the query by score
- Return the top *K* (e.g., *K* = 10) to the user

Resources for today's lecture

- IIR 6.2 – 6.4.3
- http://www.miislita.com/information-retrieval-tutorial/cosine-similarity-tutorial.html
  - Term weighting and cosine similarity tutorial