

Multimedia Content Management

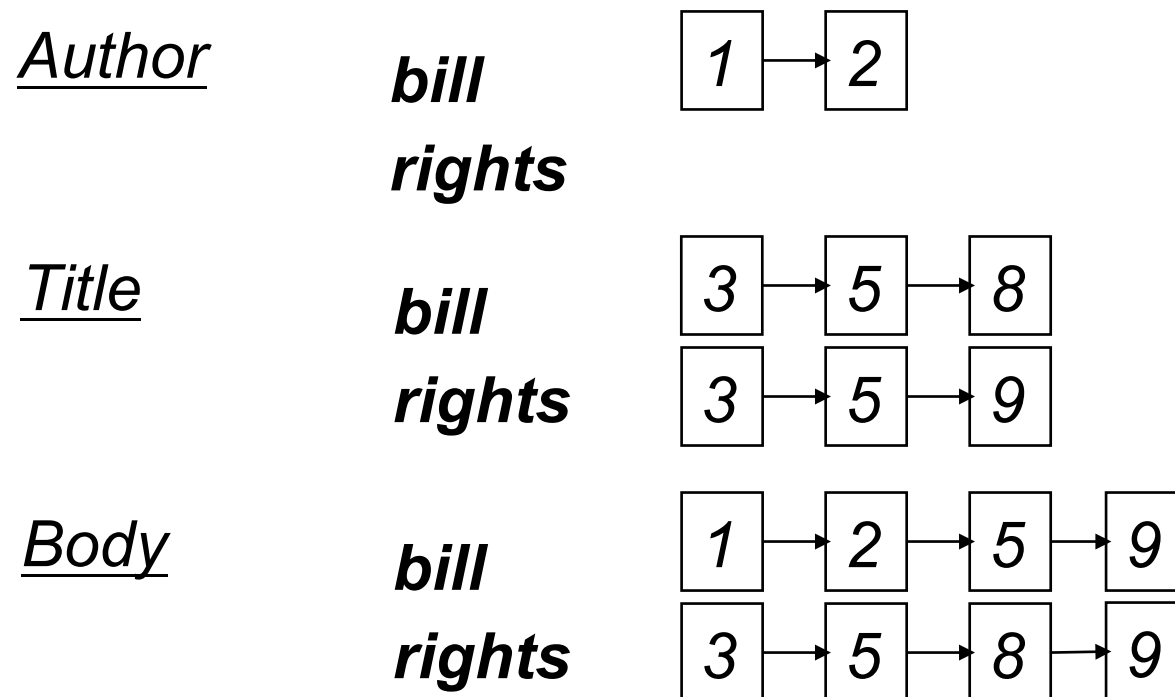
Ralf Moeller
Hamburg Univ. of Technology

Recap of the last lecture

- Parametric and field searches
 - ◆ Zones in documents
- Can apply text queries to images due to interpretation results
- Scoring documents: zone weighting
 - ◆ Index support for scoring
- *tf×idf* and vector spaces

Indexes: “Postings lists”

- On the query **bill OR rights** suppose that we retrieve the following docs from the various zone indexes:



Recap: tf x idf (or tf.idf)

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

$$w_{i,d} = tf_{i,d} \times \log(n / df_i)$$

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

n = total number of documents

df_i = the number of documents that contain term i

- *Instead of tf, sometimes wf is used:*

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

This lecture

- Vector space scoring
- Efficiency considerations
 - ◆ Nearest neighbors and approximations

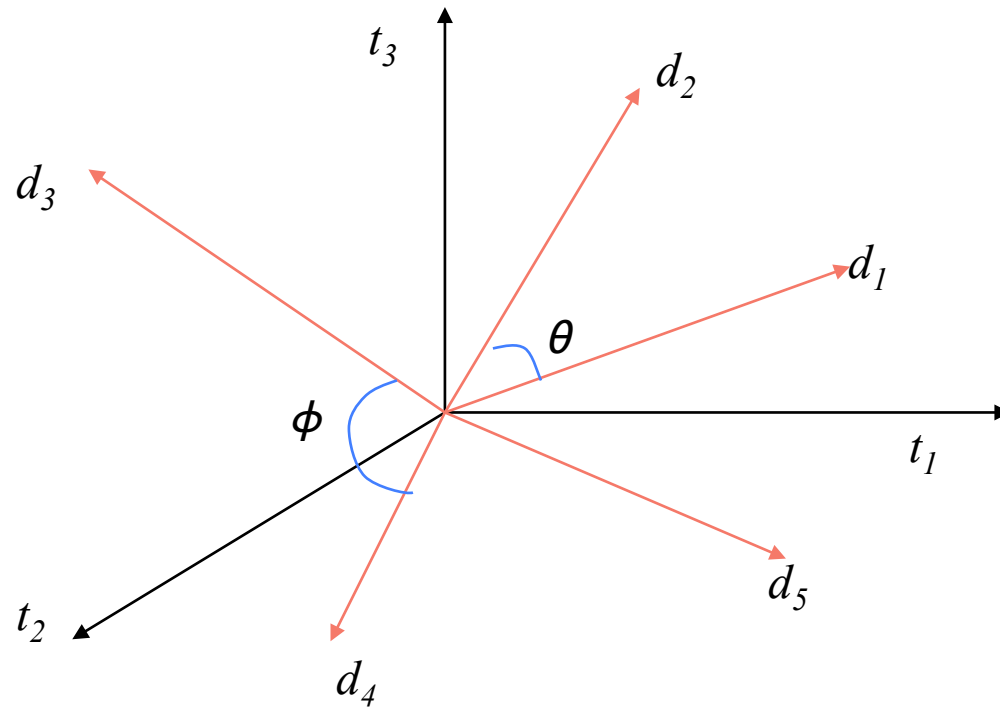
Documents as vectors

- At the end of the last lecture we said:
- Each doc d 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 50,000+ 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.

Desiderata for proximity

- If d_1 is near d_2 , then d_2 is near d_1 .
- If d_1 near d_2 , and d_2 near d_3 , then d_1 is not far from d_3 .
- No doc is closer to d than d itself.
- Triangle inequality

First cut

- Idea: Distance between d_1 and d_2 is the length of the vector $|d_1 - d_2|$.

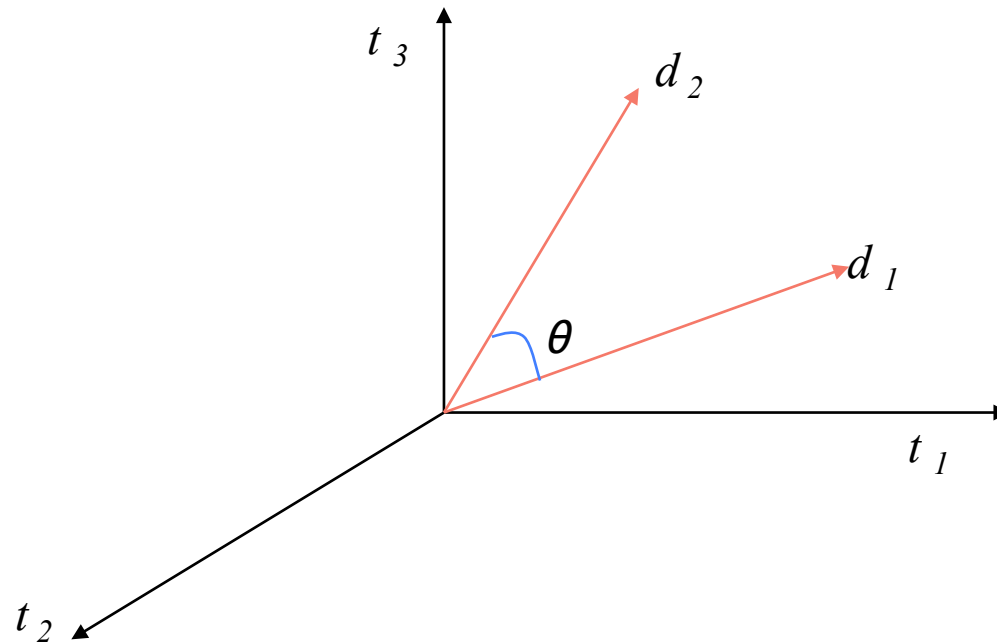
- ♦ Euclidean distance:

$$|d_j - d_k| = \sqrt{\sum_{i=1}^n (d_{i,j} - d_{i,k})^2}$$

- Why is this not a great idea?
- We still haven't dealt with the issue of length normalization
 - ♦ Short documents would be more similar to each other by virtue of length, not topic
- However, we can implicitly normalize by looking at *angles* instead

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

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

- This maps vectors onto the unit sphere:
- Then, $|\vec{d}_j| = \sqrt{\sum_{i=1}^n w_{i,j}^2} = 1$
- Longer documents don't get more weight

Cosine similarity

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

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



Normalization

Normalized vectors

- For normalized vectors, the cosine is simply the dot product:

$$\cos(\vec{d}_j, \vec{d}_k) = \vec{d}_j \cdot \vec{d}_k$$

Example

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

	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

- $\cos(\text{SAS}, \text{PAP}) = .996 \times .993 + .087 \times .120 + .017 \times 0.0 = 0.999$
- $\cos(\text{SAS}, \text{WH}) = .996 \times .847 + .087 \times .466 + .017 \times .254 = 0.889$

Cosine similarity exercises

- *Exercise: Rank the following by decreasing cosine similarity. Assume tf-idf weighting:*
 - ◆ Two docs that have only frequent words (***the, a, an, of***) in common.
 - ◆ Two docs that have no words in common.
 - ◆ Two docs that have many rare words in common (***wingspan, tailfin***).

Exercise

- Show that, for normalized vectors, Euclidean distance gives the same proximity ordering as the cosine measure

Queries in the vector space model

Central idea: the query as a vector:

- We regard the query as short document
- We return the documents ranked by the closeness of their vectors to the query, also represented as a vector.

$$\text{sim}(d_j, d_q) = \frac{\vec{d}_j \cdot \vec{d}_q}{|\vec{d}_j| |\vec{d}_q|} = \frac{\sum_{i=1}^n w_{i,j} w_{i,q}}{\sqrt{\sum_{i=1}^n w_{i,j}^2} \sqrt{\sum_{i=1}^n w_{i,q}^2}}$$

- Note that d_q is very sparse!

Summary: What's the point of using vector spaces?

- A well-formed algebraic space for retrieval
- Key: A user's query can be viewed as a (very) short document.
- Query becomes a vector in the same space as the docs.
- Can measure each doc's proximity to it.
- Natural measure of scores/ranking – no longer Boolean.
 - ♦ Queries are expressed as bags of words

Digression: spamming indices

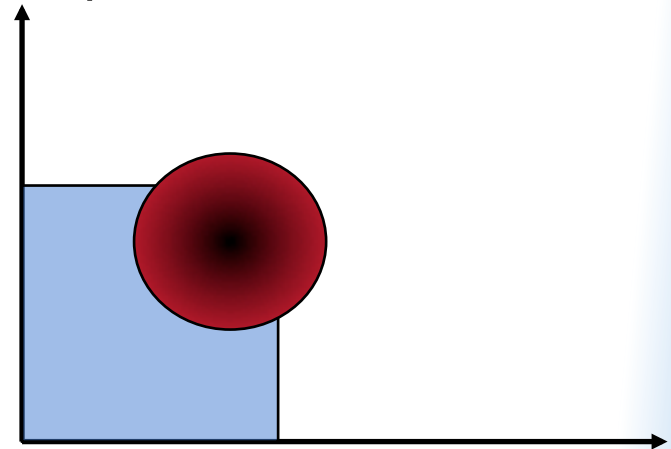
- This was all invented before the days when people were in the business of spamming web search engines. Consider:
 - ♦ Indexing a sensible passive document collection vs.
 - ♦ An active document collection, where people (and indeed, service companies) are shaping documents in order to maximize scores
- Vector space similarity may not be as useful in this context.

Interaction: vectors and phrases

- Scoring phrases doesn't fit naturally into the vector space world:
 - ♦ *“tangerine trees” “marmalade skies”*
 - ♦ Positional indexes don't calculate or store tf.idf information for *“tangerine trees”*
- Biword indexes treat certain phrases as terms
 - ♦ For these, we can pre-compute tf.idf.
 - ♦ Theoretical problem of correlated dimensions
- Problem: we cannot expect end-user formulating queries to know what phrases are indexed
- We can use a positional index to boost or ensure phrase occurrence

Vectors and Boolean queries

- Vectors and Boolean queries really don't work together very well
- In the space of terms, vector proximity selects by spheres: e.g., all docs having cosine similarity ≥ 0.5 to the query
- Boolean queries on the other hand, select by (hyper-)rectangles and their unions/intersections
- Round peg – square hole



Vectors and wild cards

- How about the query *tan* marm**?
 - ♦ Can we view this as a bag of words?
 - ♦ Thought: expand each wild-card into the matching set of dictionary terms.
- Danger – unlike the Boolean case, we now have *tfs* and *idfs* to deal with.
- Net – not a good idea.

Vector spaces and other operators

- Vector space queries are apt for no-syntax, bag-of-words queries
 - ◆ Clean metaphor for similar-document queries
- Not a good combination with Boolean, wild-card, positional query operators
- But ...

Query language vs. scoring

- May allow user a certain query language, say
 - ◆ Free text basic queries
 - ◆ Phrase, wildcard etc. in Advanced Queries.
- For scoring (oblivious to user) may use all of the above, e.g. for a free text query
 - ◆ Highest-ranked hits have query as a phrase
 - ◆ Next, docs that have all query terms near each other
 - ◆ Then, docs that have some query terms, or all of them spread out, with $tf \times idf$ weights for scoring

Efficient cosine ranking

- Find the k docs in the corpus “nearest” to the query $\Rightarrow k$ largest query-doc cosines.
- Efficient ranking:
 - ◆ Computing a single cosine efficiently.
 - ◆ Choosing the k largest cosine values efficiently.
 - Can we do this without computing all n cosines?
 - n = number of documents in collection

Efficient cosine ranking

- What we're doing in effect: solving the k -nearest neighbor problem for a query vector
- In general, we do not know how to do this efficiently for high-dimensional spaces
- But it is solvable for short queries, and standard indexes are optimized to do this

Computing a single cosine

- For every term i , with each doc j , store term frequency tf_{ij} .
 - ♦ Some tradeoffs on whether to store term count, term weight, or weighted by idf_i .
- At query time, use an array of accumulators A_j to accumulate component-wise sum

$$sim(\vec{d}_j, \vec{d}_q) = \sum_{i=1}^m w_{i,j} \times w_{i,q}$$

- If you're indexing 5 billion documents (web search) an array of accumulators is infeasible

← Ideas?

Indexing for retrieving document frequencies

<i>aargh</i>	10		1,2	7,3	83,1	87,2	...
<i>abacus</i>	8		1,1	5,1	13,1	17,1	...
<i>acacia</i>	35		7,1	8,2	40,1	97,3	...

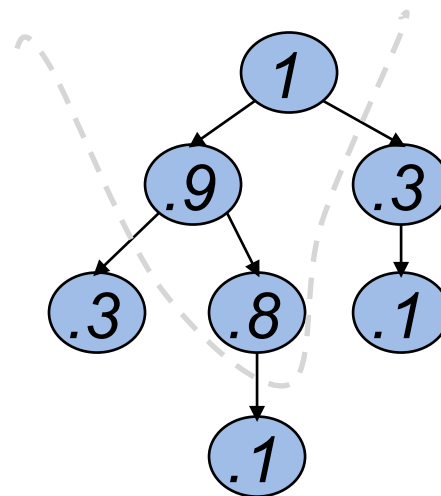
- Add $tf_{t,d}$ to postings lists
 - ◆ Overall, requires little additional space

Computing the k largest cosines: selection vs. sorting

- Typically we want to retrieve the top k docs (in the cosine ranking for the query)
 - ◆ not to totally order all docs in the corpus
- Can we pick off docs with k highest cosines?

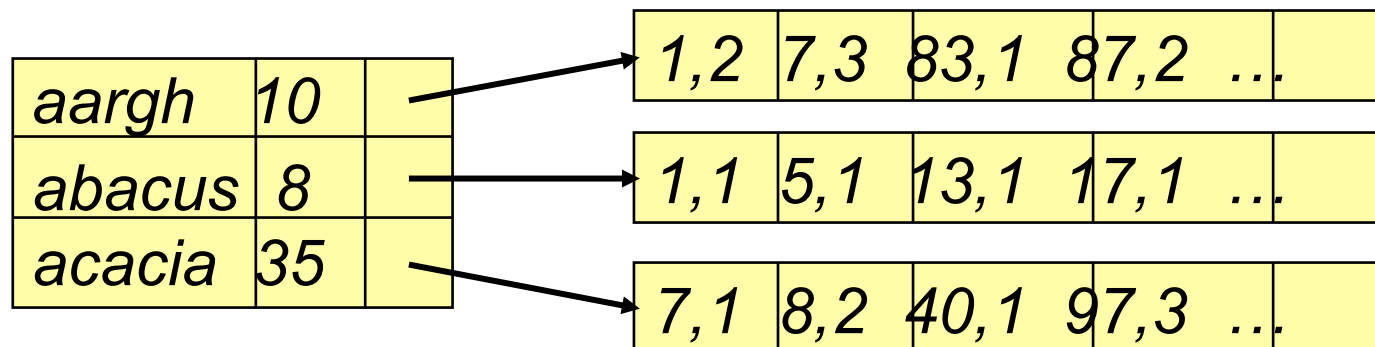
Use heap for selecting top k

- Binary tree in which each node's value $>$ the values of children
- Takes $2n$ operations to construct, then each of k “winners” read off in $2\log n$ steps.
- For $n=1\text{M}$, $k=100$, this is about 10% of the cost of sorting.



Bottleneck

- Still need to first compute cosines from query to each of n docs \rightarrow several seconds for $n = 1M$.
- Can select from only non-zero cosines
 - ♦ Need union of postings lists accumulators ($\ll 1M$)



Removing bottlenecks

- Can further limit to documents with non-zero cosines on rare (high idf) words
- Or enforce conjunctive search (a la Google): non-zero cosines on *all* words in query
 - ♦ Get # accumulators down to {min of postings lists sizes}
- But in general still potentially expensive
 - ♦ Sometimes have to fall back to (expensive) soft-conjunctive search:
 - ♦ If no docs match a 4-term query, look for 3-term subsets, etc.

Can we avoid all this computation ?

- Yes, but may occasionally get an answer wrong
 - ♦ a doc *not* in the top k may creep into the answer.

Limiting the accumulators: Best m candidates

- Preprocess: Pre-compute, for each term, its m nearest docs.
 - ◆ (Treat each term as a 1-term query.)
 - ◆ lots of preprocessing.
 - ◆ Result: “preferred list” for each term.
- Search:
 - ◆ For a t -term query, take the union of their t preferred lists – call this set S , where $|S| \leq mt$.
 - ◆ Compute cosines from the query to only the docs in S , and choose the top k .

Need to pick $m > k$ to work well empirically.

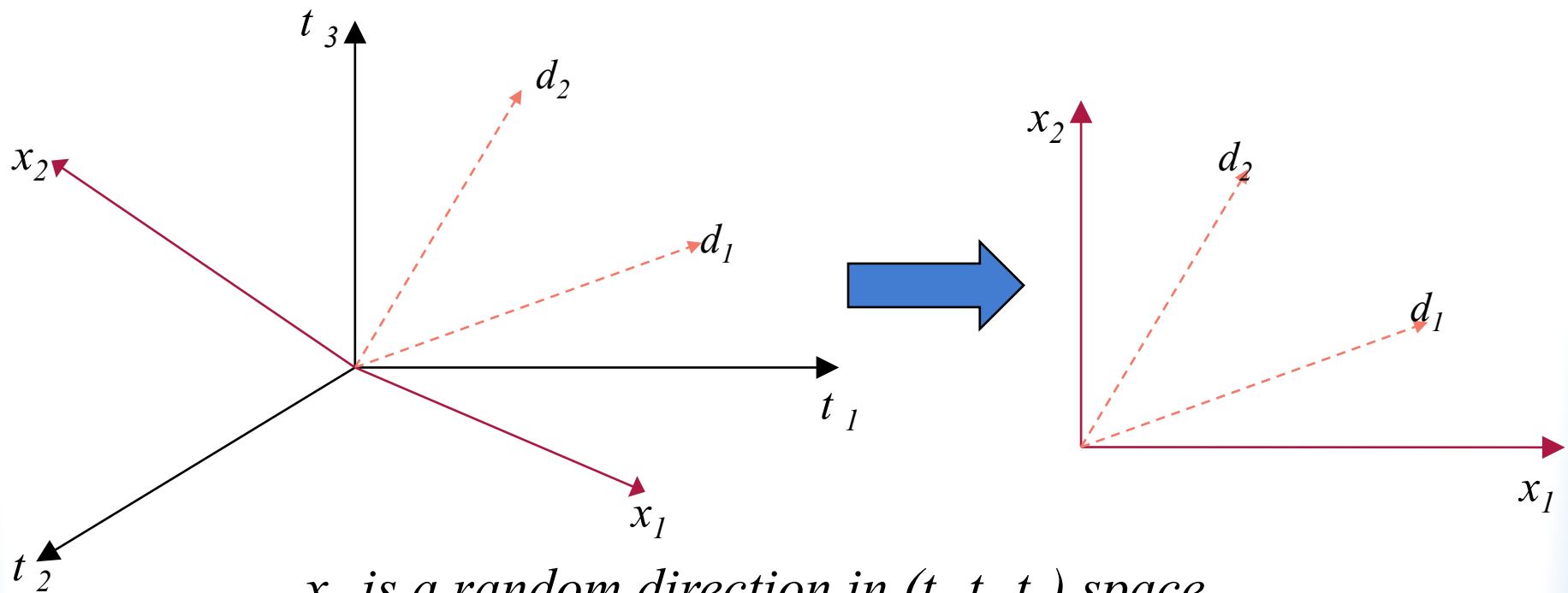
Dimensionality reduction

- What if we could take our vectors and “pack” them into fewer dimensions (say 50,000→100) while preserving distances?
- (Well, almost.)
 - ♦ Speeds up cosine computations.
- Two methods:
 - ♦ Random projection.
 - ♦ “Latent semantic indexing”.

Random projection onto $k \ll m$ axes

- Choose a random direction x_1 in the vector space.
- For $i = 2$ to k ,
 - ◆ Choose a random direction x_i that is orthogonal to x_1, x_2, \dots, x_{i-1} .
- Project each document vector into the subspace spanned by $\{x_1, x_2, \dots, x_k\}$.

E.g., from 3 to 2 dimensions



*x_1 is a random direction in (t_1, t_2, t_3) space.
 x_2 is chosen randomly but orthogonal to x_1 .*

Dot product of x_1 and x_2 is zero.

Guarantee

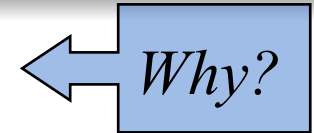
- With high probability, relative distances are (approximately) preserved by projection.
- Pointer to precise theorem in Resources.

Computing the random projection

- Projecting n vectors from m dimensions down to k dimensions:
 - ♦ Start with $m \times n$ matrix of terms \times docs, A .
 - ♦ Find random $k \times m$ orthogonal projection matrix R .
 - ♦ Compute matrix product $W = R \times A$.
- j^{th} column of W is the vector corresponding to doc j , but now in $k \ll m$ dimensions.

Cost of computation

- This takes a total of kmn multiplications.
- Expensive – see Resources for ways to do essentially the same thing, quicker.
- *Question: by projecting from 50,000 dimensions down to 100, are we really going to make each cosine computation faster?*



Latent semantic indexing (LSI)

- Another technique for dimension reduction
- Random projection was data-*independent*
- LSI on the other hand is data-*dependent*
 - ◆ Eliminate redundant axes
 - ◆ Pull together “related” axes – hopefully
 - *car* and *automobile*
- More on LSI when studying clustering, in the next lecture.