Information Retrieval

Lecture 6

Recap of the last lecture

- Parametric and field searches
  - Zones in documents
- Scoring documents: zone weighting
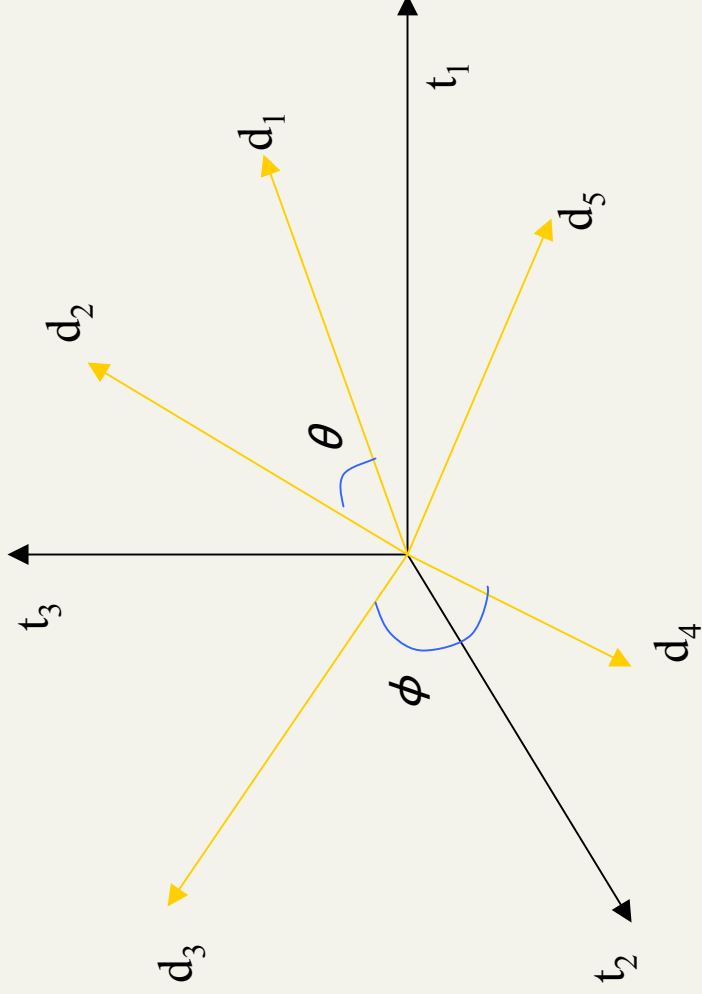 - Index support for scoring
- *tf×idf* and vector spaces

This lecture

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

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

Query as vector:

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

- Developed in the SMART system (Salton, c. 1970).

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.

First cut

- Distance between d_1 and d_2 is the length of the vector $|d_1 - d_2|$.
  - Euclidean distance
- Why is this not a great idea?
- We still haven't dealt with the issue of length normalization
  - Long 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.
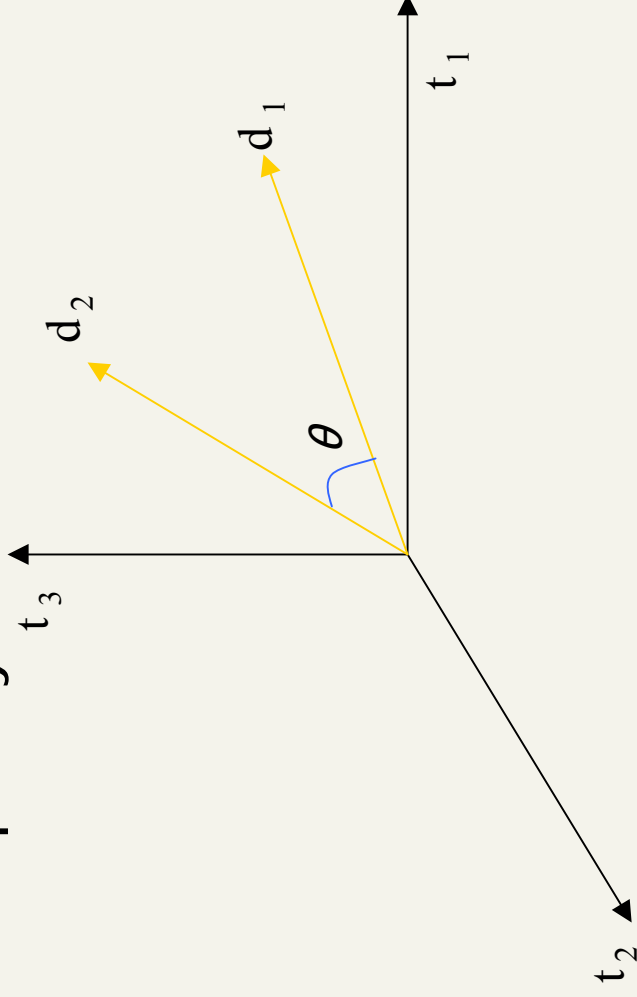
Cosine similarity

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

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

Normalization

Cosine similarity

- Define the length of a document vector by

$$\text{Length } |\vec{d}| = \sqrt{\sum_{i=1}^{n} d_i^2}$$

- 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:

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

- Longer documents don't get more weight

Normalized vectors

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

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

Cosine similarity exercises

- *Exercise: Rank the following by decreasing cosine similarity:*
  - 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

- Euclidean distance between vectors:

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

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

Example

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

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

	SaS	PaP	WH
affection	0.996	0.993	0.847
jealous	0.087	0.120	0.466
gossip	0.017	0.000	0.254

- cos(SAS, PAP) = .996 x .993 + .087 x .120 + .017 x 0.0 = 0.999
- cos(SAS, WH) = .996 x .847 + .087 x .466 + .017 x .254 = 0.929

Digression: spamming indices

- This was all invented before the days when people were in the business of spamming web search engines:
  - 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

Digression: ranking in ML

- Our problem is:
  - Given document collection D and query q, return a ranking of D according to relevance to q.
- Such ranking problems have been much less studied in machine learning than classification/regression problems
- But much more interest recently, e.g.,
  - W.W. Cohen, R.E. Schapire, and Y. Singer. Learning to order things. *Journal of Artificial Intelligence Research*, 10:243–270, 1999.
  - And subsequent research

Digression: ranking in ML

- Many "WWW" applications are ranking (or *ordinal regression*) problems:
  - Text information retrieval
  - Image similarity search (QBIC)
  - Book/movie recommendations
    - Collaborative filtering
  - Meta-search engines

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

- 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

Vectors and phrases

- Phrases don't fit naturally into the vector space world:
  - *"tangerine trees" "marmalade skies"*
  - Positional indexes don't capture tf/idf information for *"tangerine trees"*
- Biword indexes (lecture 2) treat certain phrases as terms
  - For these, can pre-compute tf/idf.
- A hack: cannot expect end-user formulating queries to know what phrases are indexed

Vectors and Boolean queries

- Vectors and Boolean queries really don't work together very well

- In the space of terms, vector proximity selects by <u>spheres</u>: 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
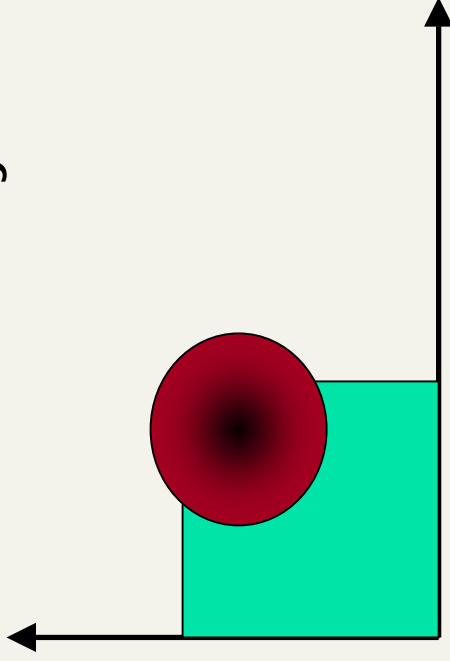
Vectors and wild cards

- 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 *tf*s and *idf*s to deal with.
- Net – not a good idea.

Vector spaces and other operators

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

Exercises

- How would you augment the inverted index built in lectures 1–3 to support cosine ranking computations?

- Walk through the steps of serving a query.

- *The math of the vector space model is quite straightforward, but being able to do cosine ranking efficiently at runtime is nontrivial*

Efficient cosine ranking

- Find the k docs in the corpus "nearest" to the query ⟹ 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?

Efficient cosine ranking

- What an IR system does is in effect solve the k-nearest neighbor problem for each query

- In general 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.
- Accumulate component-wise sum

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

- More on speeding up a single cosine later on
- If you're indexing 5 billion documents (web search) an array of accumulators is infeasible

Ideas?

Encoding document frequencies

aargh	2
abacus	8
acacia	35

1,2	7,3	83,1	87,2	...
1,1	5,1	13,1	17,1	...
7,1	8,2	40,1	97,3	...
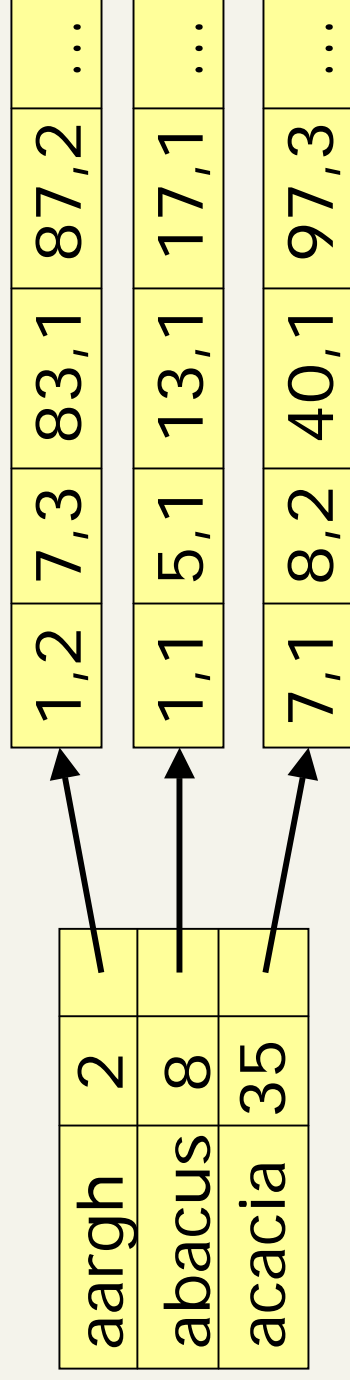
- Add $tf_{d,t}$ to postings lists
  - Almost always as frequency – scale at runtime
  - Unary code is very effective here — **Why?**
  - γ code (Lecture 1) is an even better choice
  - Overall, requires little additional space

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 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 >
  values of children

- Takes *2n* operations to construct, then each
  of *k* log *n* "winners" read off in 2log *n* steps.

- For *n*=1M, *k*=100, this is about 10% of the
  cost of sorting.
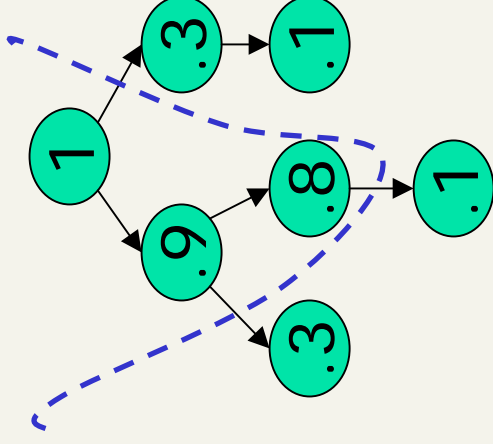
Bottleneck

- Still need to first compute cosines from query to each of n docs → several seconds for n = 1M.

- Can select from only non-zero cosines

  - Need union of postings lists accumulators (<<1M): on the query *aargh abacus* would only do accumulators 1,5,7,13,17,83,87 (below).

aargh	2			
abacus	8			
acacia	35			

1,2	7,3	83,1	87,2	...
1,1	5,1	13,1	17,1	...
7,1	8,2	40,1	97,3	...
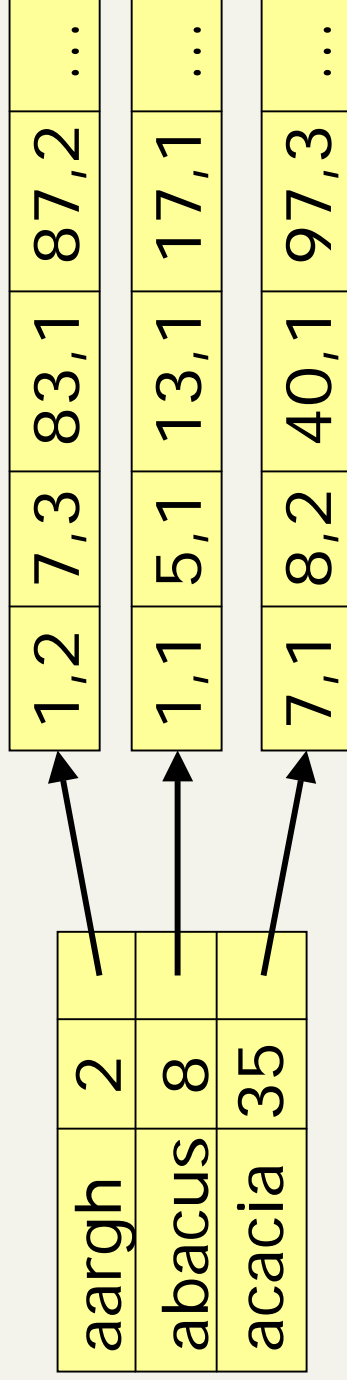
Removing bottlenecks

- Can further limit to documents with non-zero cosines on rare (high idf) words
- Enforce conjunctive search (a la Google): non-zero cosines on *all* words in query
  - Get # accumulators down to {min of postings lists sizes}
- But 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 this?

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

Term-wise candidates

- <u>Preprocess</u>: 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.
- <u>Search</u>:
  - 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 top k.

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

Exercises

- Fill in the details of the calculation:
  - Which docs go into the preferred list for a term?
- Devise a small example where this method gives an incorrect ranking.

Cluster pruning

- First run a pre-processing phase:
  - pick √n *docs* at random: call these *leaders*
  - For each other doc, pre-compute nearest leader
    - Docs attached to a leader: its *followers*;
    - <u>Likely</u>: each leader has ~ √n followers.

- Process a query as follows:
  - Given query *Q*, find its nearest *leader L*.
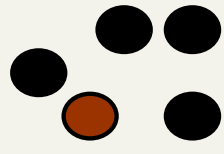 - Seek *k* nearest docs from among *L*'s followers.

Visualization
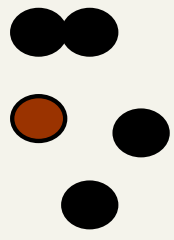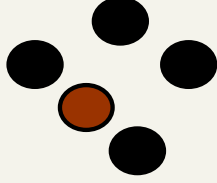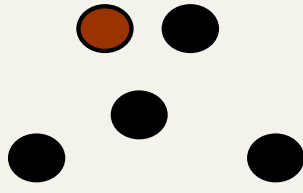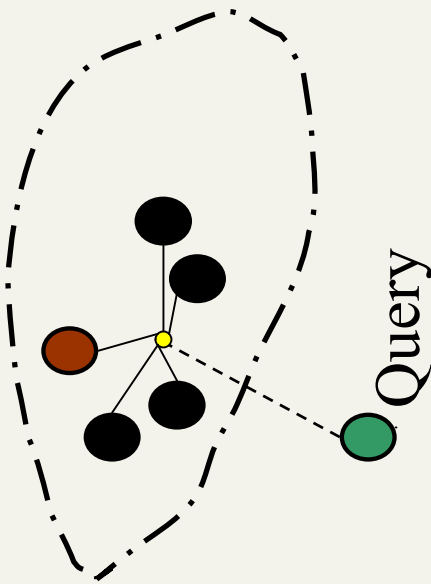
Query

Follower

Leader

Why use random sampling

- Fast
- Leaders reflect data distribution

General variants

- Have each follower attached to $a=3$ (say) nearest leaders.

- From query, find $b=4$ (say) nearest leaders and their followers.

- Can recur on leader/follower construction.

Exercises

- To find the nearest leader in step 1, how many cosine computations do we do?
  - Why did we have √n in the first place?
- What is the effect of the constants a,b on the previous slide?
- Devise an example where this is *likely to* fail
  – i.e., we miss one of the k nearest docs.
   - *Likely* under random sampling.

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:
  - "Latent semantic indexing".
  - Random projection.

Random projection onto
k<<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, ... x_{i-1}.
- Project each document vector into the subspace spanned by $\{x_1, x_2, ..., x_k\}$.

E.g., from 3 to 2 dimensions
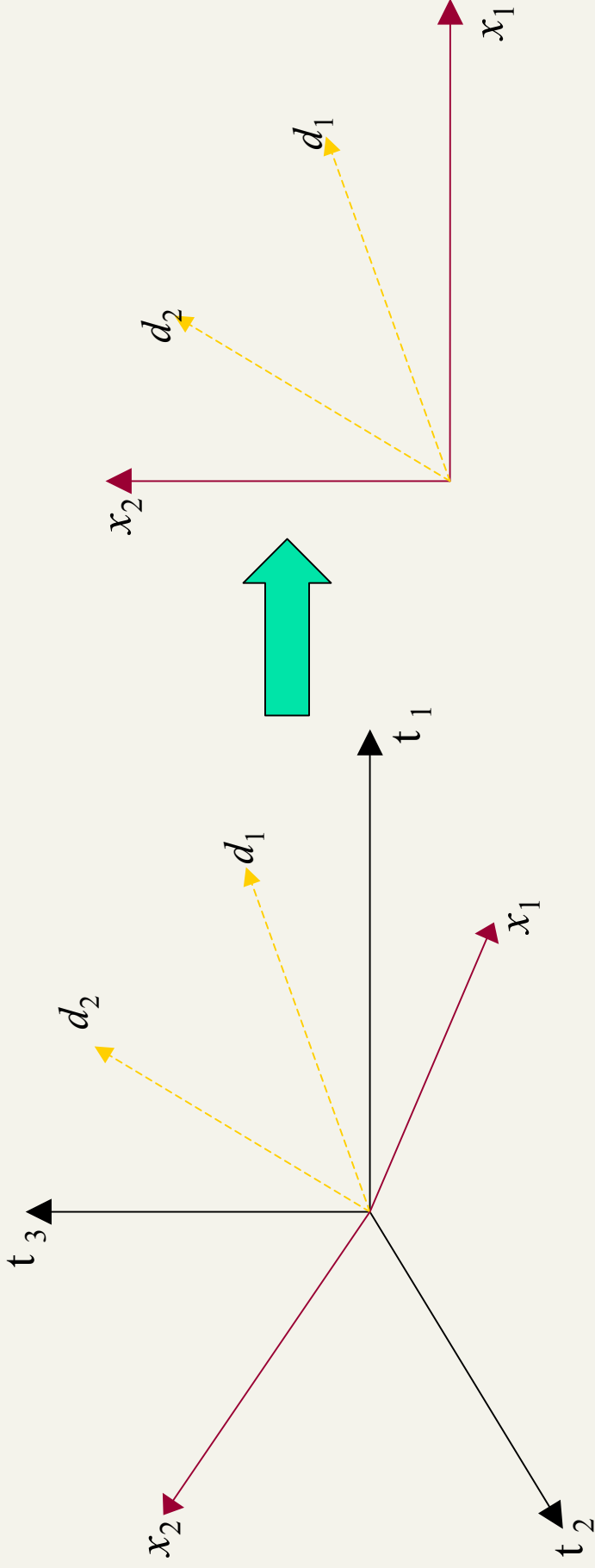
x_1 is a random direction in (t_1, t_2, t_3) space.
x_2 is chosen randomly but orthogonal to x_1.

Guarantee

- With high probability, relative distances are (approximately) preserved by projection.

- Pointer to precise theorem in Resources.

- (Using random projection is a newer idea: it's somewhat surprising, but it works very well.)

Computing the random projection

- Projecting n vectors from m dimensions down to k dimensions:
  - Start with $m \times n$ matrix of terms × 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 $k << m$ 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?*

Why?

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*

Notions from linear algebra

- Matrix, vector
- Matrix transpose and product
- Rank
- Eigenvalues and eigenvectors.

Resources

- MG Ch. 4.4–4.6; MIR 2.5, 2.7.2; FSNLP 15.4
  - Random projection theorem
  - Faster random projection
- http://lsi.argreenhouse.com/lsi/LSIpapers.html
- http://lsa.colorado.edu/
- http://www.cs.utk.edu/~lsi/