

# Non-Boolean models of retrieval: Agenda

- Review of Boolean model and TF/IDF
  - Simple extensions thereof
- Vector model
- Language Model-based retrieval
- Matrix decomposition methods

# Non-Boolean models of retrieval: Agenda

- Review of Boolean model and TF/IDF
  - Simple extensions thereof
- Vector model
- Language Models and IR
- Matrix decomposition methods

# Simple Boolean model:

Inverted Index mapping terms to docs:

Brutus	→	1	2	4	11	31	45	173	174	
Caesar	→	1	2	4	5	6	16	57	132	...
Calpurnia	→	2	31	54	101					

Results obtained by intersecting/disjoining/etc.  
posting lists:

Brutus	→	1	→	2	→	4	→	11	→	31	→	45	→	173	→	174
Calpurnia	→	2	→	31	→	54	→	101								
Intersection	⇒	2	→	31												

Query: Brutus *AND* Calpurnia

One big limitation: no obvious way to rank results.

*Intuitive solution: an article that uses the query terms many times is probably “more relevant” than one that only uses them once...*

*... but not all query terms are equally informative.*

Term Frequency (TF): How often does a term occur in the document?

Document Frequency (DF): In how many documents does the term occur?

Inverse Document Frequency (IDF):  $\text{idf}_t = \log \frac{N}{\text{df}_t}$

*Rare terms have high IDF, common terms have very low IDF.*

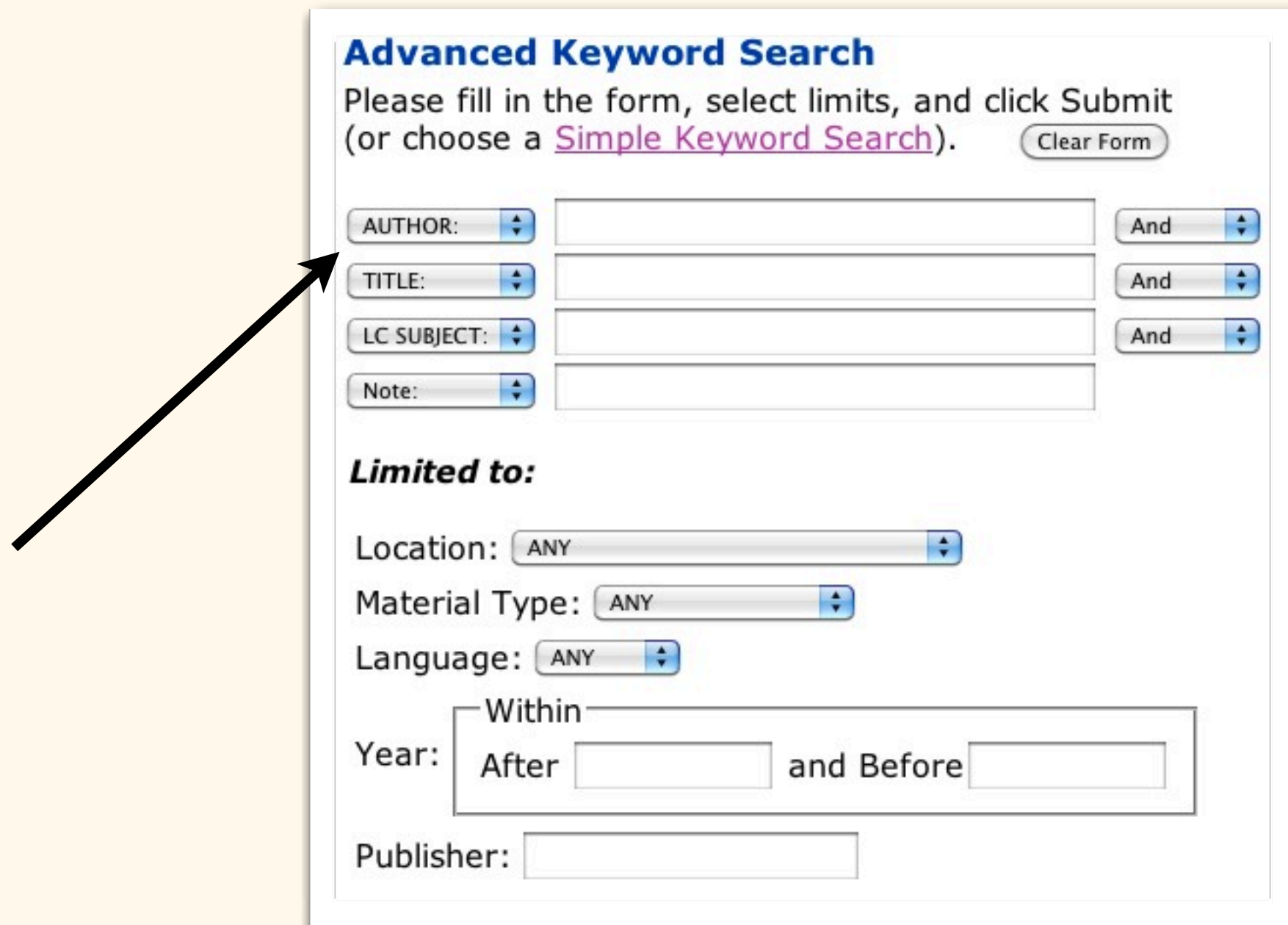
By combining these, we get a scoring that is:

1. Highest when  $t$  occurs frequently within a small group of documents...
2. Lower when term occurs less frequently within a document, or occurs in many documents...
3. Lowest when term is extremely common

$$\text{tf-idf}_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$$

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

# Extension to the simple model: zone scoring



**Advanced Keyword Search**  
Please fill in the form, select limits, and click Submit (or choose a [Simple Keyword Search](#)).

AUTHOR:	<input type="text"/>	And
TITLE:	<input type="text"/>	And
LC SUBJECT:	<input type="text"/>	And
Note:	<input type="text"/>	

**Limited to:**

Location:

Material Type:

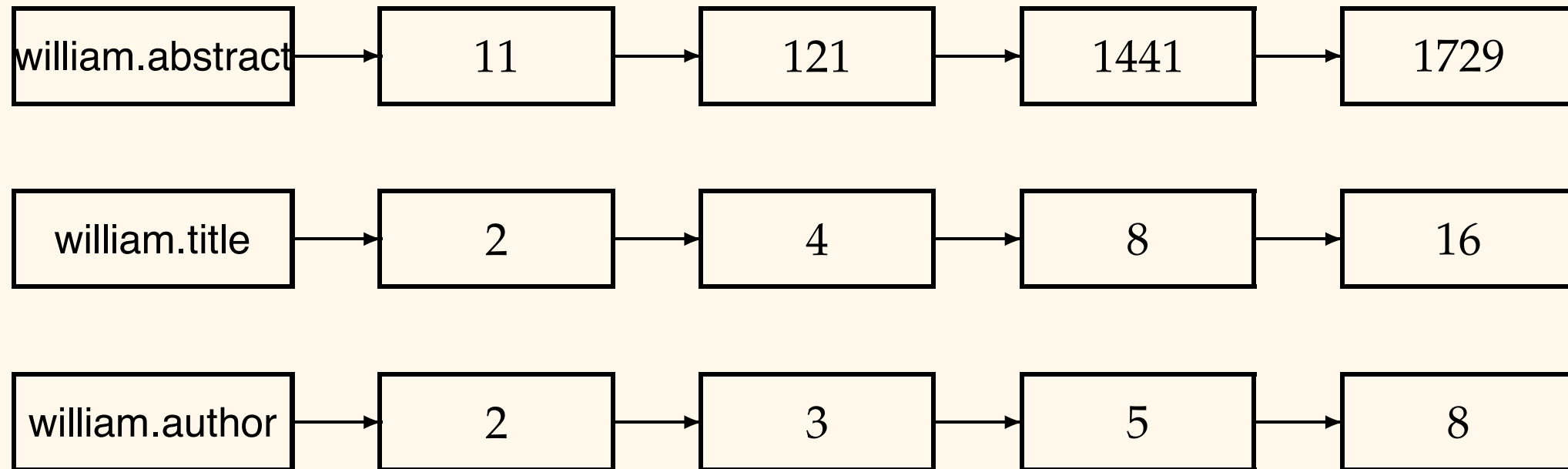
Language:

Year:  After  and Before

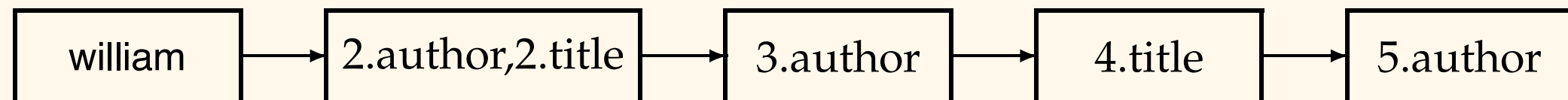
Publisher:

We can define “fields” or “zones” containing different parts of the documents’ structure.

# Extension to the simple model: zone scoring



Zones can be implemented in the main index as “virtual” terms...



... or as extensions to the dictionary data structures.

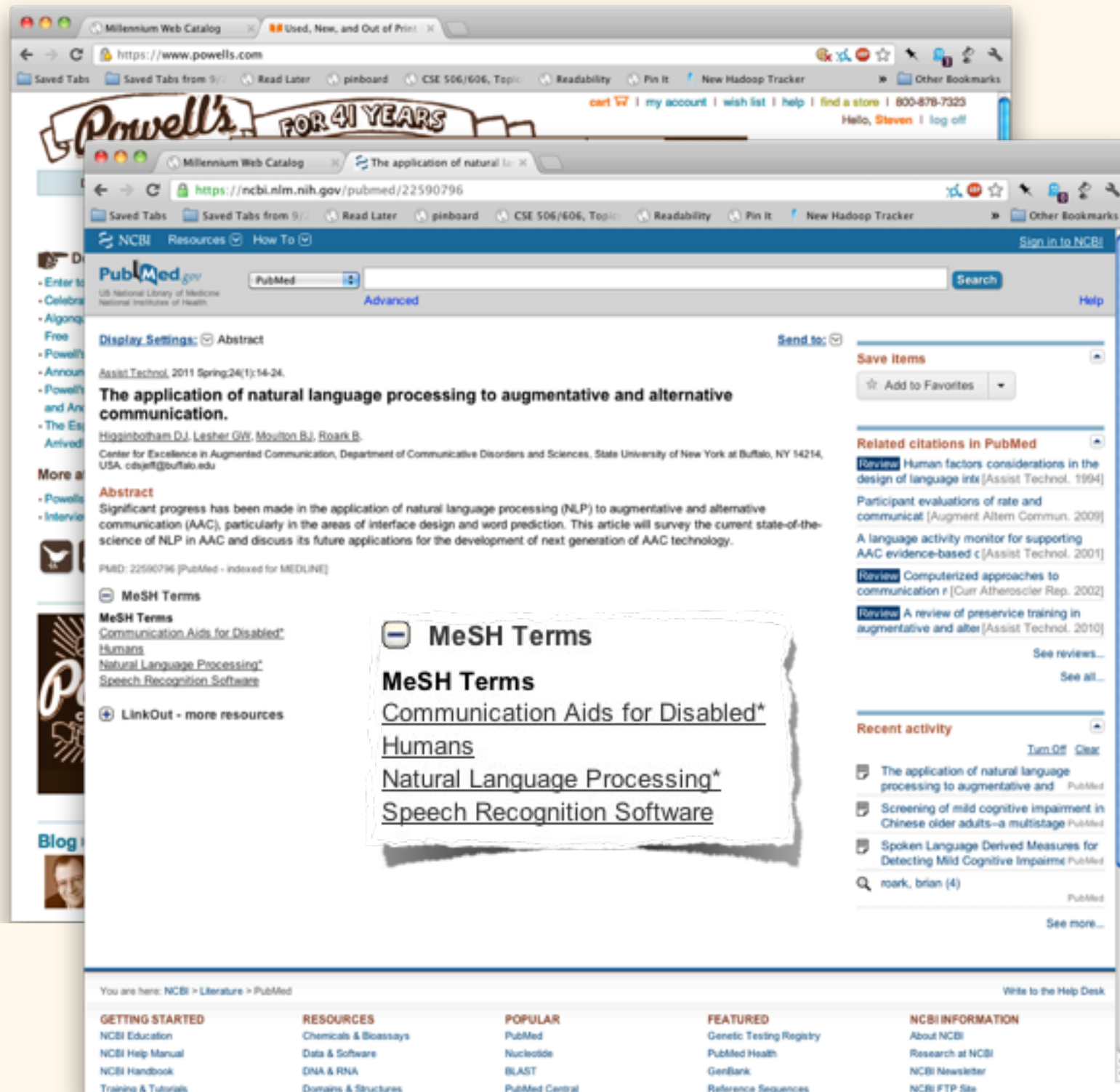


# Not all zones/fields are equally important:



Titles might be more important than descriptions.

# Not all zones/fields are equally important:



Index terms  
might get high  
priority.

*Weighted-zone scoring* lets us represent this.

Basic idea: define  $z$  zones, each of which have a weight  $g$ .

To score a document, then, we compute a linear sum of the weighted scores for each zone.

$$\sum_{i=1}^z g_i s_i$$

*Weighted-zone scoring* lets us represent this.

Basic idea: define  $z$  zones, each of which have a weight  $g$ .

To score a document, then, we compute a linear sum of the weighted scores for each zone.

$s_i$  can be whatever we want: a Boolean flag indicating term presence/absence, etc.

Similar to original intersection algorithm for a simple AND query...

```
ZONE SCORE( $q_1, q_2$ )
1  float  $scores[N] = [0]$ 
2  constant  $g[\ell]$ 
3   $p_1 \leftarrow postings(q_1)$ 
4   $p_2 \leftarrow postings(q_2)$ 
5  //  $scores[]$  is an array with a score entry for each document, initialized to zero.
6  //  $p_1$  and  $p_2$  are initialized to point to the beginning of their respective postings.
7  // Assume  $g[]$  is initialized to the respective zone weights.
8  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
9  do if  $docID(p_1) = docID(p_2)$ 
10     then  $scores[docID(p_1)] \leftarrow \text{WEIGHTEDZONE}(p_1, p_2, g)$ 
11          $p_1 \leftarrow next(p_1)$ 
12          $p_2 \leftarrow next(p_2)$ 
13     else if  $docID(p_1) < docID(p_2)$ 
14         then  $p_1 \leftarrow next(p_1)$ 
15     else  $p_2 \leftarrow next(p_2)$ 
16 return  $scores$ 
```

... except we're building up a list of scores rather than documents.

# How to decide on zone weights?

$$\sum_{i=1}^z g_i s_i$$

Experts or users can set them...

... or we can use machine learning to discover optimal weights.

Note: this requires expensive manual relevance judgment!



# Non-Boolean models of retrieval: Agenda

- Review of Boolean model and TF/IDF
  - Simple extensions thereof
- Vector model
- Language Models and IR
- Matrix decomposition methods

We can model a document as a *vector*:

Each component corresponds to a term in the index dictionary...

... and the magnitude of each component corresponds to some *score* (presence/absence, tf, tf/idf, etc.).



This lets us represent our collection of documents in a common coordinate space...

... which in turn gives us a whole new toolbox for working with them.

# How might we compare the “similarity” of two documents using this model?

One approach might measure the magnitude of the vector difference between the two docs.

$$\text{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)| |\vec{V}(d_2)|}$$



Normalize for vector (document) length

This is also referred to as the *cosine similarity* between  $d_1$  and  $d_2$ .

$$\text{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)| |\vec{V}(d_2)|}$$

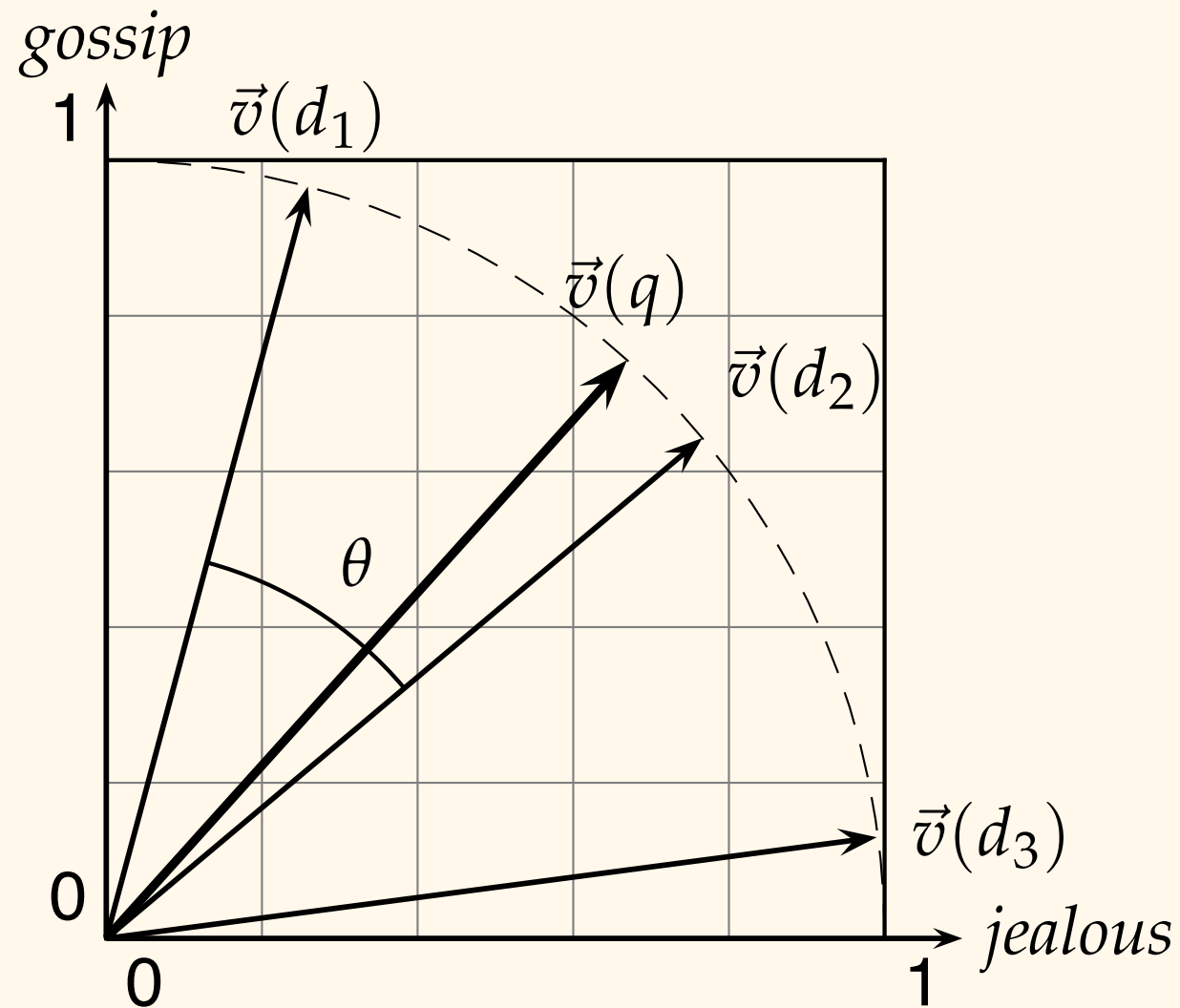


term	SaS	PaP	WH
affection	115	58	20
jealous	10	7	11
gossip	2	0	6

term	SaS	PaP	WH
affection	0.996	0.993	0.847
jealous	0.087	0.120	0.466
gossip	0.017	0	0.254

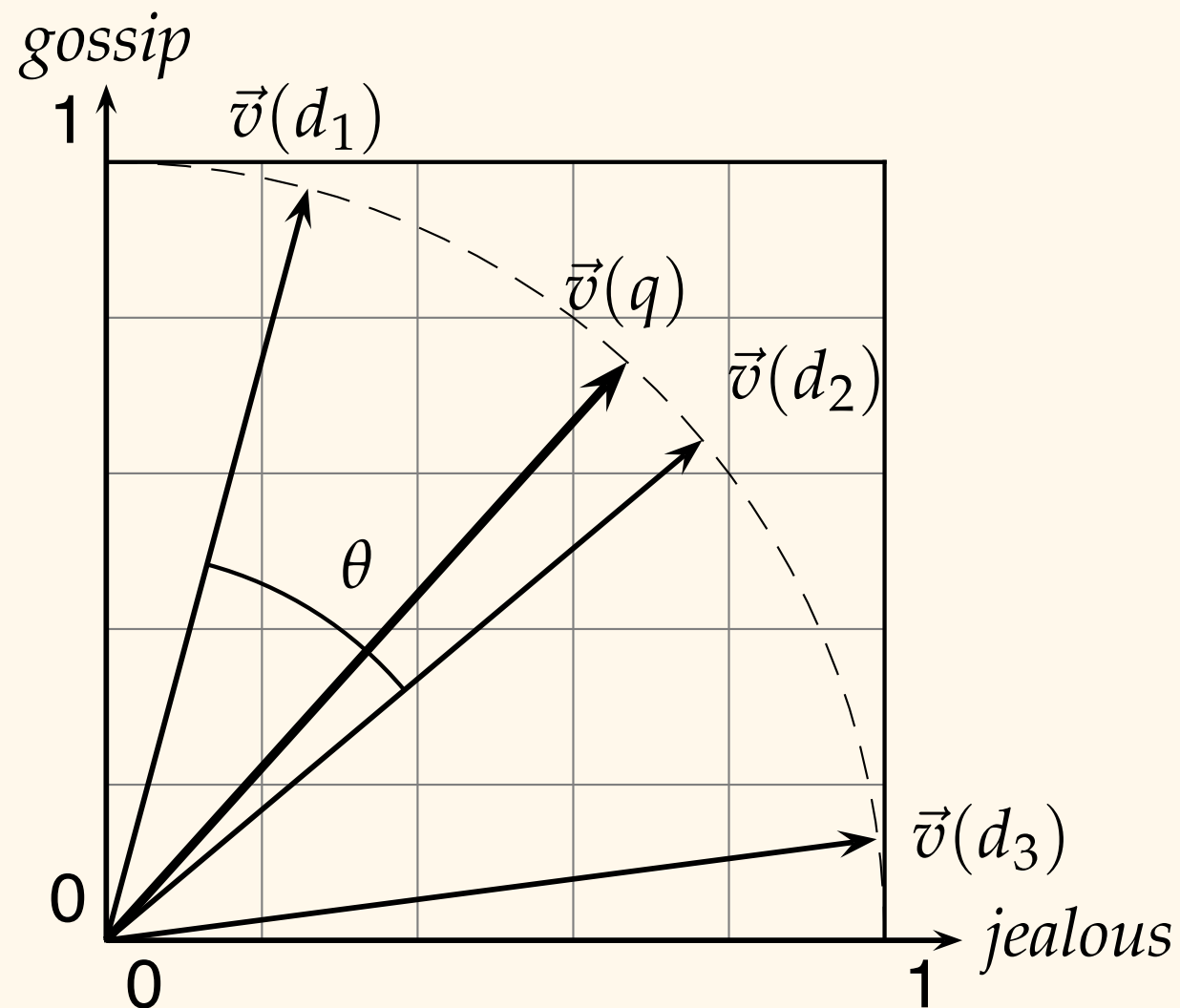


How might we compare the “similarity” of two documents using this model?



$$\vec{v}(d_1) \cdot \vec{v}(d_2) = \cos(\theta)$$

Now, finding the  $n$  most similar documents to a given document is simple:



Compute pairwise similarity scores, take the top  $n$ .

This vector model lends itself nicely to the construction of a *term-document* matrix.

	$t_1$	$t_2$	$t_3$	$t_4$	$\dots$	$t_n$
$d_1$						
$d_2$						
$d_3$						
$d_4$						
$\dots$						
$d_m$						

$m$  documents (rows),  $n$  terms (cols)

This lends itself nicely to the construction of a *term-document* matrix.

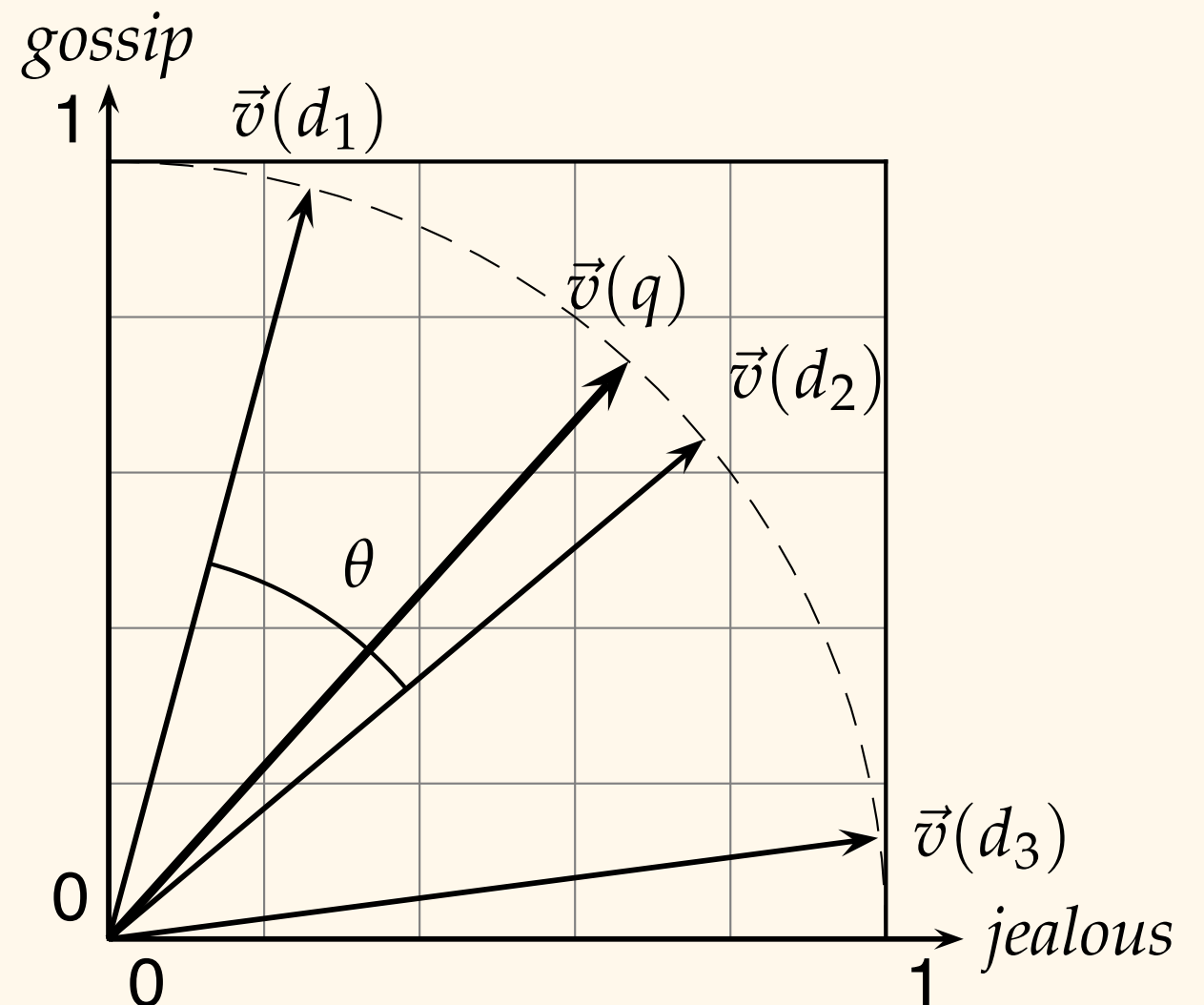
	$t_1$	$t_2$	$t_3$	$t_4$	$\dots$	$t_n$
$d_1$						
$d_2$						
$d_3$						
$d_4$						
$\dots$						
$d_m$						

We'll come back to this later...

Another nice thing about this model: queries as vectors.

Since we're treating documents as "bags-of-words", we can pretend that queries are just "short documents" ...

$$\text{score}(q, d) = \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)| |\vec{V}(d)|}$$

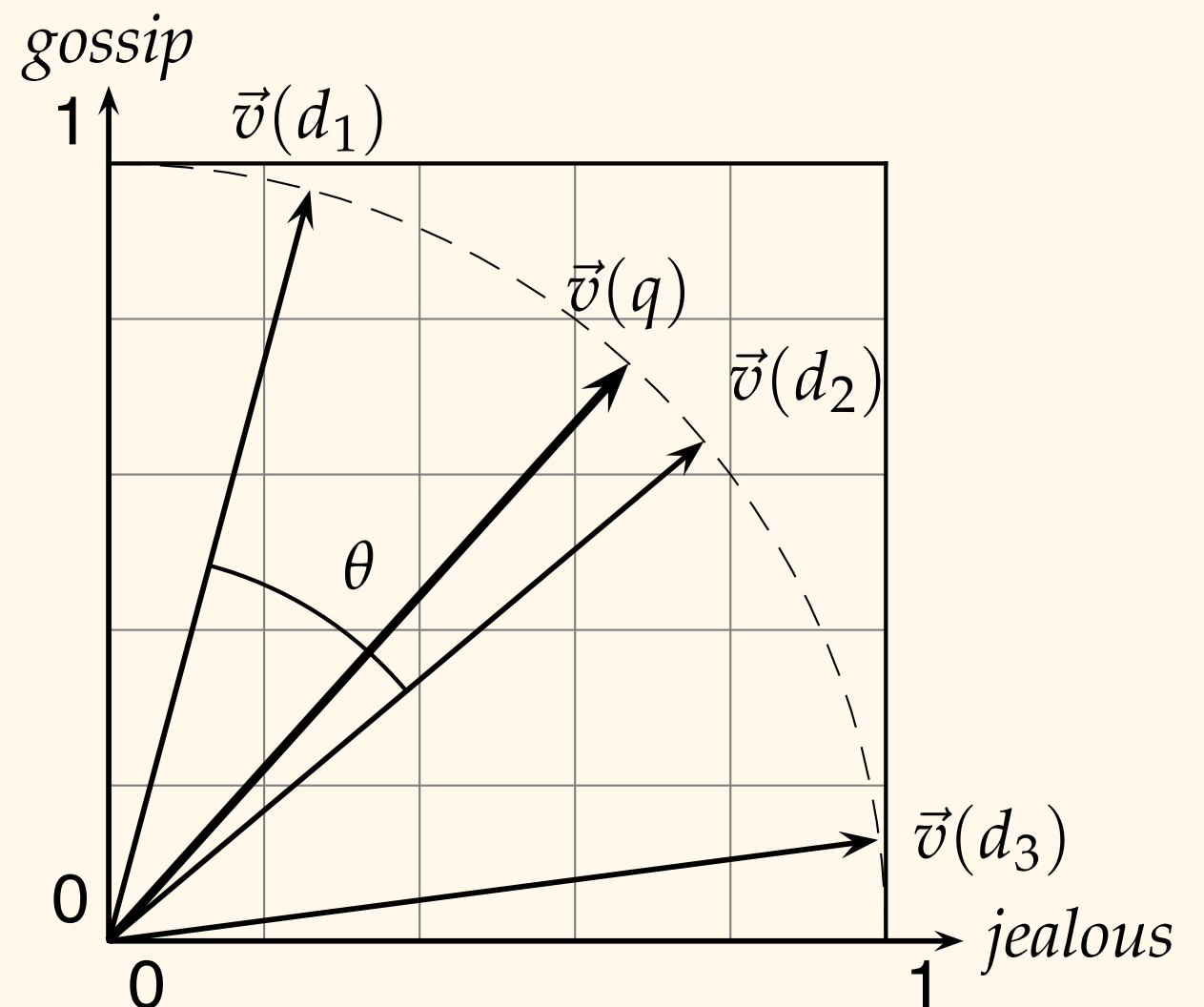




Another nice thing about this model: queries as vectors.

Of course, in reality, the number of dimensions will be *much* higher.

$$\text{score}(q, d) = \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)| |\vec{V}(d)|}$$



Which leads to the main drawback of the vector model:

$$\text{score}(q, d) = \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)| |\vec{V}(d)|}$$

Calculating multi-thousand-dimension dot-products over millions of documents is expensive.

# One solution: *term-at-a-time* scoring.

```
COSINESCORE( $q$ )
1  float  $Scores[N] = 0$ 
2  Initialize  $Length[N]$ 
3  for each query term  $t$ 
4  do calculate  $w_{t,q}$  and fetch postings list for  $t$ 
5      for each pair( $d, tf_{t,d}$ ) in postings list
6      do  $Scores[d] += wf_{t,d} \times w_{t,q}$ 
7  Read the array  $Length[d]$ 
8  for each  $d$ 
9  do  $Scores[d] = Scores[d] / Length[d]$ 
10 return Top  $K$  components of  $Scores[]$ 
```

Because each query term's contribution is independent, we can calculate each term separately.

# There are many variations on tf-idf.

*Sublinear tf scaling:*

If a term occurs 20 times in a document, is it truly 20 times more relevant than a single occurrence?

$$\text{wf}_{t,d} = \begin{cases} 1 + \log \text{tf}_{t,d} & \text{if } \text{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

# There are many variations on tf-idf.

*Maximum tf normalization:*

Longer documents have higher average TF, but are not necessarily more relevant.

This is especially important in collections with widely-varying document lengths.

$$\text{ntf}_{t,d} = a + (1 - a) \frac{\text{tf}_{t,d}}{\text{tf}_{\max}(d)}$$

smoothing term



# There are many variations on tf-idf.

Term frequency		Document frequency		Normalization	
n (natural)	$tf_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(tf_{t,d})$	t (idf)	$\log \frac{N}{df_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2 + w_2^2 + \dots + w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times tf_{t,d}}{\max_t(tf_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N - df_t}{df_t}\}$	u (pivoted unique)	$1 / u$ (Section 6.4.4)
b (boolean)	$\begin{cases} 1 & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1 / CharLength^\alpha, \alpha < 1$
L (log ave)	$\frac{1 + \log(tf_{t,d})}{1 + \log(\text{ave}_{t \in d}(tf_{t,d}))}$				

“SMART notation”

We can use different combinations of weighting schemes on the document vectors than on the query vectors.

# There are many variations on tf-idf.

One common scheme: *Inc./tc*

Term frequency		Document frequency		Normalization	
n (natural)	$tf_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(tf_{t,d})$	t (idf)	$\log \frac{N}{df_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2 + w_2^2 + \dots + w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times tf_{t,d}}{\max_t(tf_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N - df_t}{df_t}\}$	u (pivoted unique)	$1 / u$ (Section 6.4.4)
b (boolean)	$\begin{cases} 1 & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1 / CharLength^\alpha, \alpha < 1$
L (log ave)	$\frac{1 + \log(tf_{t,d})}{1 + \log(\text{ave}_{t \in d}(tf_{t,d}))}$				

“SMART notation”

There are also numerous possible optimizations of the vector space model.

Index elimination:

Only consider postings with idf over some pre-set threshold.

Champion lists:

Pre-compute sets of high-scoring documents for each term in the dictionary; only do cosine computation for postings in a champion list.

Cluster pruning:

Cluster document vectors; select random subset from each cluster, and use those as the starting points for calculations.



# Non-Boolean models of retrieval: Agenda

- Review of Boolean model and TF/IDF
  - Simple extensions thereof
- Vector model
- Language Models and IR
- Matrix decomposition methods

# The basic idea:

Treat each document as a representative text sampled from a “language”...

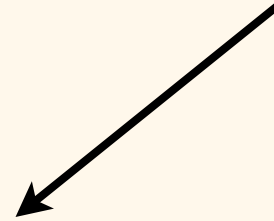
... build a probabilistic language model for each document...

... and use that to find the documents whose languages would be most likely to produce the query.

More formally, we want to rank documents  $d$  against query  $q$  by calculating:  $P(d|q)$

# How to do that?

Documents assumed to be  
equally probable...



$$P(d|q) = P(q|d) \frac{P(d)}{P(q)}$$



... ditto for queries...

... so what we are really doing is modeling the probability  
of the query  $q$  being generated by document  $d$ 's model.

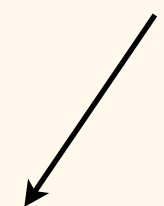
We can use MLE to estimate  $\hat{P}(d|q)$ :

For the unigram case:

$$\hat{P}(q|M_d) = \prod_{t \in q} \hat{P}_{\text{mle}}(t|M_d) = \prod_{t \in q} \frac{\text{tf}_{t,d}}{L_d}$$

What about when a query term isn't present in the document?

*Model from entire collection*

$$\hat{P}(t|d) = \lambda \hat{P}_{\text{mle}}(t|M_d) + (1 - \lambda) \hat{P}_{\text{mle}}(t|M_c)$$


Linear interpolation smoothing is a common solution.

The smoothing isn't just to make the math work.

$$\hat{P}(t|d) = \lambda \hat{P}_{\text{mle}}(t|M_d) + (1 - \lambda) \hat{P}_{\text{mle}}(t|M_c)$$

It helps use the entire collection to inform an individual document's score...

... and this effect can be affected by tuning the smoothing parameter.

# A Language Modeling Approach to Information Retrieval

Jay M. Ponte and W. Bruce Croft  
Computer Science Department  
University of Massachusetts, Amherst  
{ponte, croft}@cs.umass.edu

**Abstract** Models of document indexing and document retrieval have been extensively studied. The integration of these two classes of models has been the goal of several researchers but it is a very difficult problem. We argue that much of the reason for this is the lack of an adequate indexing model. This suggests that perhaps a better indexing model would help solve the problem. However, we feel that making unwarranted parametric assumptions will not lead to better retrieval performance. Furthermore, making prior assumptions about the similarity of documents is not warranted either. Instead, we propose an approach to retrieval based on probabilistic language modeling. We estimate models for each document individually. Our approach to modeling is non-parametric and integrates document indexing and document retrieval into a single model. One advantage of our approach is that collection statistics which are used heuristically in many other retrieval models are an integral part of our model. We have implemented our model and tested it empirically. Our approach significantly outperforms standard *tf.idf* weighting on two different collections and query sets.

## 1 Introduction

Over the past three decades, probabilistic models of document retrieval have been studied extensively. In general, these approaches can be characterized as methods of estimating the probability of relevance of documents to user queries. One component of a probabilistic retrieval model is the indexing model, i.e., a model of the assignment of indexing terms to documents. We argue that the current indexing models have not led to improved retrieval results. We believe this is due to two unwarranted assumptions made by these models. We have taken a different approach based on non-parametric estimation that allows us to relax these assumptions. We have implemented our approach and empirical results on two different collections and query sets are significantly better than the standard *tf.idf* method of retrieval. Now we take a brief look at some existing models of document indexing.

We begin our discussion of indexing models with the 2-Poisson model, due to Bookstein and Swanson [1] and

also to Harter [7]. By analogy to manual indexing, the task was to assign a subset of words contained in a document (the 'specialty words') as indexing terms. The probability model was intended to indicate the useful indexing terms by means of the differences in their rate of occurrence in documents 'elite' for a given term, i.e., a document that would satisfy a user posing that single term as a query, vs. those without the property of eliteness.

The success of the 2-Poisson model has been somewhat limited but it should be noted that Robertson's *tf*, which has been quite successful, was intended to behave similarly to the 2-Poisson model [12].

Other researchers have proposed a mixture model of more than two Poisson distributions in order to better fit the observed data. Margulis proposed the *n*-Poisson model and tested the idea empirically [10]. The conclusion of this study was that a mixture of *n*-Poisson distributions provides a very close fit to the data. In a certain sense, this is not surprising. For large values of *n* one can fit a very complex distribution arbitrarily closely by a mixture of *n* parametric models if one has enough data to estimate the parameters [18]. However, what is somewhat surprising is the closeness of fit for relatively small values of *n* reported by Margulis [10].

Nevertheless, the *n*-Poisson model has not brought about increased retrieval effectiveness in spite of the close fit to the data. In any event, the semantics of the underlying distributions are less obvious in the *n*-Poisson case as compared to the 2-Poisson case where they model the concept of eliteness.

Apart from the adequacy of the available indexing models, estimating the parameters of these models is a difficult problem. Researchers have looked at this problem from a variety of perspectives and we will discuss several of these of these approaches in section 2. In addition, as previously mentioned, many of the current indexing models make assumptions about the data that we feel are unwarranted.

- The parametric assumption.
- Documents are members of pre-defined classes.

In our approach we relax these two assumptions. Rather than making parametric assumptions, as is done in the 2-Poisson model it is assumed that terms follow a mixture of two Poisson distributions, as Silverman said, "the data will be allowed to speak for themselves [16]." We feel that it is unnecessary to construct a parametric model of the data when we have the actual data. Instead, we rely on non-parametric methods.

Regarding the second assumption, the 2-Poisson model was originally based on the idea of 'eliteness' [7]. It was assumed that a document elite for a given term would

## Precision

Rec.	tf-idf	LM	%chg	
0.0	0.7439	0.7590	+2.0	
0.1	0.4521	0.4910	+8.6	
0.2	0.3514	0.4045	+15.1	*
0.3	0.2761	0.3342	+21.0	*
0.4	0.2093	0.2572	+22.9	*
0.5	0.1558	0.2061	+32.3	*
0.6	0.1024	0.1405	+37.1	*
0.7	0.0451	0.0760	+68.7	*
0.8	0.0160	0.0432	+169.6	*
0.9	0.0033	0.0063	+89.3	
1.0	0.0028	0.0050	+76.9	
Ave	0.1868	0.2233	+19.55	*

Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or fee. SIGIR'98, Melbourne, Australia © 1998 ACM 1-58113-015-5 8/98 \$5.00.

# Issues with the LM approach:

1. Smoothing parameters can be finicky and hard to tune properly
2. Difficult to expand beyond unigram models
3. No obvious way to incorporate relevance feedback

More recent work has focused on addressing issues #2 and #3.

# Document-likelihood model:

Directly estimate  $P(d|q)$  by building a LM from the query.

1. Less text to work with, so smaller (and less accurate) model...
2. However, relevance feedback is easy: incorporate words from relevant documents into query model.

Of course, other variations are possible (see text).



# Non-Boolean models of retrieval: Agenda

- Review of Boolean model and TF/IDF
  - Simple extensions thereof
- Vector model
- Language Models and IR
- Matrix decomposition methods

# Quick linear algebra review:

$$S = \begin{pmatrix} 30 & 0 & 0 \\ 0 & 20 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Rank: 3

Eigenvalues: values of lambda such that  $C\vec{x} = \lambda\vec{x}$

$S$  has three:  $\lambda_1 = 30, \lambda_2 = 20, \lambda_3 = 1$

$$\vec{x}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \vec{x}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \vec{x}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

# Matrix decomposition:

$$S = \begin{pmatrix} 30 & 0 & 0 \\ 0 & 20 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

There exists an *eigendecomposition*  $S = U\Lambda U^{-1}$

Where the columns of  $U$  are the eigenvectors of  $S$ , and  $\Lambda$  is a diagonal matrix whose entries are the eigenvalues of  $S$  *in decreasing order*.

This works for square matrices...

	$t_1$	$t_2$	$t_3$	$t_4$	$\dots$	$t_n$
$d_1$						
$d_2$						
$d_3$						
$d_4$						
$\dots$						
$d_m$						

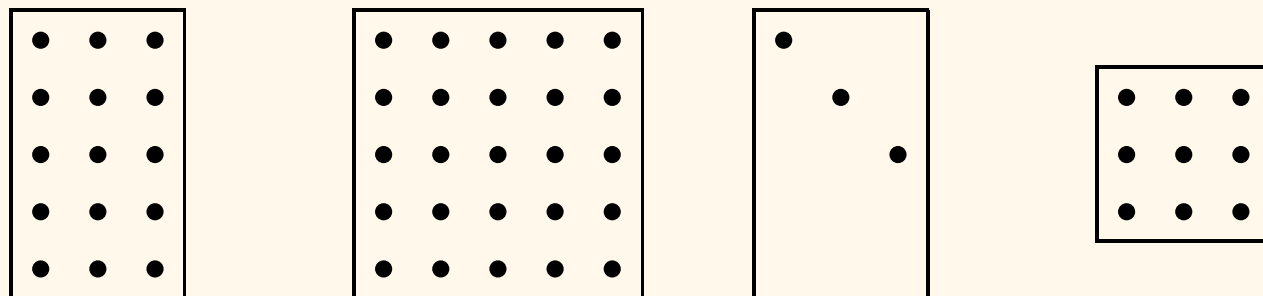
# Singular Value Decomposition:

For an  $M$  by  $N$  matrix  $C$ , we can decompose as follows:

$$C = U \Sigma V^T$$

Where the eigenvalues  $\lambda_1 \dots \lambda_r$  of  $CC^T$  are the same as the eigenvalues of  $C^T C$ ...

and  $\Sigma$  is a diagonal-ish matrix containing the *singular values* of  $\sqrt{\lambda_i}$



$C = U \Sigma V^T$

We can use the SVD to compute a *low-rank approximation* of  $C$ :

1. Compute SVD;
2. Compute from  $\Sigma$  the matrix  $\Sigma_k$  by zero-ing out  $\Sigma$ 's  $r - k$  smallest singular values.
3. Compute  $C_k = U \Sigma_k V^T$

$$C_k = U \Sigma_k V^T$$

*Latent Semantic Indexing* uses the new matrices to perform dimensionality reduction.

1. Take query vector...
2. Map to low-dimensional space:  $\vec{q}_k = \Sigma_k^{-1} U_k^T \vec{q}$
3. Compute cosine similarity in low-d space.

This often results in good and interesting results... but at a very high computational cost.

