

# Introduction to **Information Retrieval**

CS276: Information Retrieval and Web Search

Pandu Nayak and Prabhakar Raghavan

Lecture 6: Scoring, Term Weighting and the  
Vector Space Model

# This lecture; IIR Sections 6.2-6.4.3

---

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

# Ranked retrieval

---

- Thus far, our queries have all been Boolean.
  - Documents either match or don't.
- Good for expert users with precise understanding of their needs and the collection.
  - 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 of web search.

# Problem with Boolean search: feast or famine

---

- Boolean queries often result in either too few (=0) or too many (1000s) results.
- Query 1: “ ” → 200,000 hits
- Query 2: “ ”: 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**, the system returns an ordering over the (top) documents in the collection for 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 has 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
  - Indeed, the size of the result set is not an issue
  - We just show the top ( $\approx 10$ ) results
  - We don't overwhelm the user
- Premise: the ranking algorithm works

# 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”.

# Take 1: Jaccard coefficient

---

- A common measure of overlap of two sets  $A$  and  $B$
- $$\text{jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$
- $$\text{jaccard}(A, A) = 1$$
- $$\text{jaccard}(A, B) = 0 \text{ if } A \cap B = \emptyset$$
- $A$  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:
- Document 1:
- Document 2:

# Issues with Jaccard for scoring

---

- It doesn't consider (how many times a term occurs in a document)
- Rare terms in a collection are more informative than frequent terms. Jaccard doesn't consider this information
- We need a more sophisticated way of normalizing for length

# Query-document matching scores

---

- We need a way of assigning 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.

# Recall (Lecture 2): 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 the number of occurrences of a term in a document:
  - Each document is a **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	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

# model

- Vector representation doesn't consider the ordering of words in a document
- **and**  
**have the same vectors**
- This is called the bag of words model.
- In a sense, this is a step back: The positional index was able to distinguish these two documents.

# Term frequency tf

---

- The term frequency  $tf$  of term in document is defined as the number of times that occurs in .
  - Note: Frequency means count in IR
- 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

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

- $0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 1.3, 10 \rightarrow 2, 1000 \rightarrow 4$ , etc.
- Score for a document-query pair: sum over terms in both  $q$  and  $d$  :
- $\text{score} = \sum_{t \in q \cap d} (1 + \log \text{tf}_{t,d})$
- The score is 0 if none of the query terms is present in the document.



# Rare terms are more informative

---

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

.

# Collection vs. Document frequency

- Collection frequency of  $t$  is the number of occurrences of  $t$  in the collection
- Document frequency of  $t$  is the number of documents in which  $t$  occurs

- Example:

Word	Collection frequency	Document frequency
<i>insurance</i>	10440	3997
<i>try</i>	10422	8760

- Which word is for better search (gets higher weight)

# idf weight

---

- $df$  is the document frequency of : the number of documents that contain
  - $df$  is an inverse measure of the informativeness of
  - $df \leq$
- We define the idf (inverse document frequency) of by
$$idf_t = \log_{10} (N/df_t)$$
  - We use  $\log ( /df )$  instead of  $/df$  to “dampen” the effect of idf.

idf example, suppose = 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

$$idf_t = \log_{10} (N/df_t)$$

There is one idf value for each term  $t$  in a collection.

# Effect of idf on ranking

---

- Does idf have an effect on ranking for one-term queries, like
  - iPhone
- idf has no effect on ranking one term queries
  - idf affects the ranking of documents for queries with at least two terms
- For the query capricious person, idf weighting makes occurrences of capricious count for much more in the final document ranking 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.

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

- **Best known weighting scheme in information retrieval**
  - Note: the “-” in tf-idf is a hyphen, not a minus sign!
  - **Alternative names: tf.idf, tf x idf**
- Increases with the number of occurrences within a document
- **Increases with the rarity of the term in the collection**

# Score for a document given a query

---

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

- There are many variants
  - How “tf” is computed (with/without logs)
  - Whether the terms in the query are also weighted
  - ...

# Binary $\rightarrow$ count $\rightarrow$ 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  $\in \mathbb{R}^{|V|}$



# 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 when you apply this to a web search engine
- These are very sparse vectors - most entries are zero.

# Queries as vectors

---

- [Key idea 1:](#) Do the same for queries: represent them as vectors in the space
- [Key idea 2:](#) Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors
- proximity  $\approx$  inverse of distance

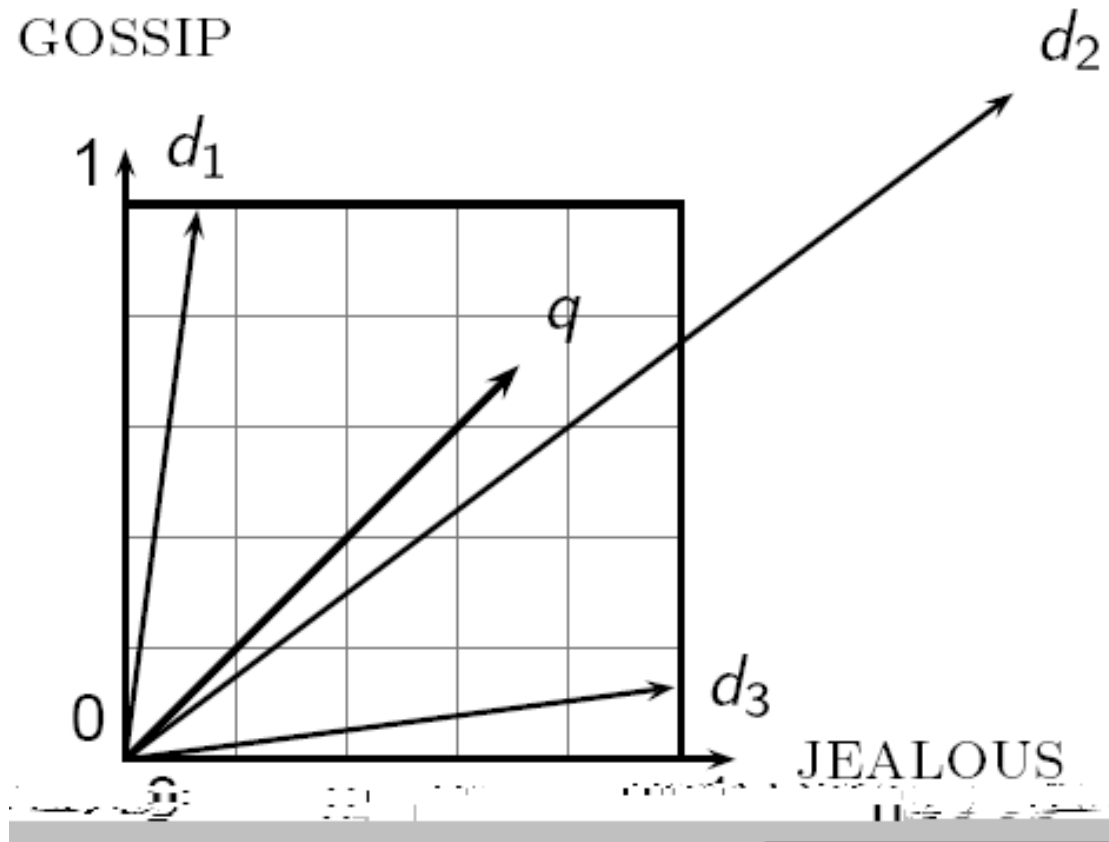
# Formalizing vector space proximity

---

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

# Why distance is a bad idea

The Euclidean distance between  $\vec{b}$  and  $\vec{q}$  is large even though the distribution of terms in the query  $\vec{q}$  and the distribution of terms in the document  $\vec{b}$  are very 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 the two documents can be quite large
- The angle between the two documents is 0, corresponding to maximal similarity.
- Key idea: Rank documents according to angle with query.

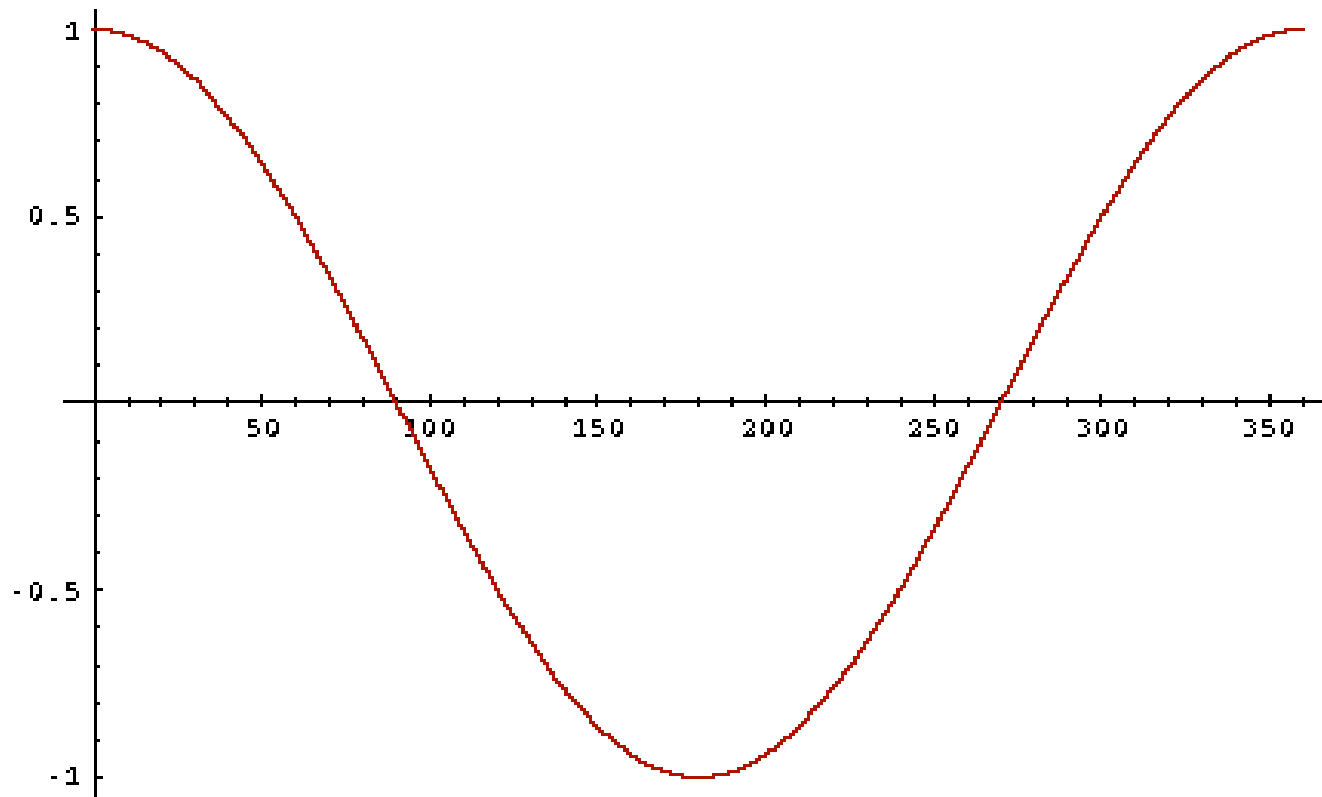
# From angles to cosines

---

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

# From angles to cosines

---



- But how should we be computing cosines?

# 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:

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

- Dividing a vector by its  $L_2$  norm makes it a unit (length) vector (on surface of unit hypersphere)
- Effect on the two documents  $d$  and  $d'$  ( $d$  appended to itself) from earlier slide: they have identical vectors after length-normalization.
  - Long and short documents now 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 weight of term  $i$  in the query

$d_i$  is the weight of term  $i$  in the document

$\cos(\vec{q}, \vec{d})$  is the cosine similarity of  $\vec{q}$  and  $\vec{d}$  ... or,  
equivalently, the cosine of the angle between  $\vec{q}$  and  $\vec{d}$ .

# Cosine for length-normalized vectors

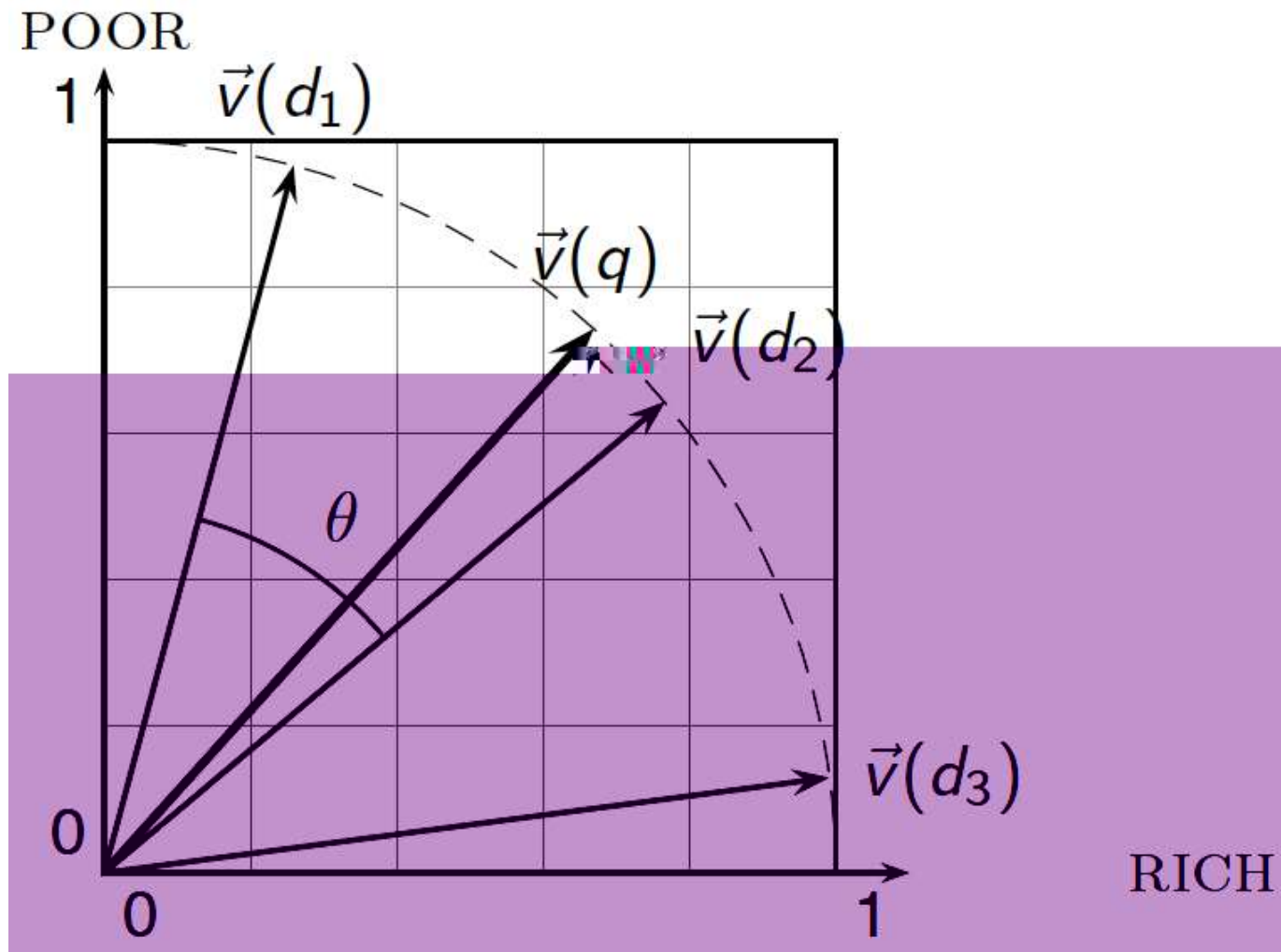
---

- 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



# Cosine similarity amongst 3 documents

How similar are  
the novels

SaS:

PaP:

, and

WH:

?

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, we don't do 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

$$\cos(\text{SaS}, \text{PaP}) \approx 0.94$$

$$\cos(\text{SaS}, \text{WH}) \approx 0.79$$

$$\cos(\text{PaP}, \text{WH}) \approx 0.69$$

# Computing cosine scores

---

COSINESCORE( $q$ )

```
1  float Scores[ $N$ ] = 0
2  float 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[ $d$ ] + =  $w_{t,d} \times w_{t,q}$ 
7  Read the array Length
8  for each  $d$ 
9  do Scores[ $d$ ] = Scores[ $d$ ] / Length[ $d$ ]
10 return Top  $K$  components of Scores[]
```

# Computing cosine scores

---

- Previous algorithm scores term-at-a-time (TAAT)
- Algorithm can be adapted to scoring document-at-a-time (DAAT)
- Storing  $\cos(\mathbf{t}, \mathbf{d})$  in each posting could be expensive
  - ...because we'd have to store a floating point number
  - For tf-idf scoring, it suffices to store  $\log(\text{tf-idf})$  in the posting and  $\log(\text{doc\_len})$  in the head of the postings list
- Extracting the top K items can be done with a priority queue (e.g., a heap)

# tf-idf weighting has many variants

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)	$1/u$
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)	$\frac{1 + \log(tf_{t,d})}{1 + \log(\text{ave}_{t \in d}(tf_{t,d}))}$				



# Weighting may differ in queries vs documents

---

- Many search engines allow for different weightings for queries vs. documents
- **SMART Notation:** denotes the combination in use in an engine, with the notation using the acronyms from the previous table
- A very standard weighting scheme is: Inc.ltc
- Document: logarithmic tf (**l as first character**), no idf and cosine normalization
- **Query: logarithmic tf (l in leftmost column), idf (t in second column), cosine normalization ...**

# tf-idf example: Inc.Itc

Document: *car insurance auto insurance*

Query: *best car insurance*

Term	Query						Document				Prod
	tf-raw	tf-wt	df	idf	wt	e	tf-raw	tf-wt	wt	e	
auto	0	0	5000	2.3	0	0	1	1	1	0.52	0
best	1	1	50000	1.3	1.3	0.34	0	0	0	0	0
car	1	1	10000	2.0	2.0	0.52	1	1	1	0.52	0.27
insurance	1	1	1000	3.0	3.0	0.78	2	1.3	1.3	0.68	0.53

$$\text{Doc length} = \sqrt{1^2 + 0^2 + 1^2 + 1.3^2} \approx 1.92$$

$$\text{Score} = 0 + 0 + 0.27 + 0.53 = 0.8$$

# Summary – vector space ranking

---

- Represent the query as a weighted tf-idf vector
- Represent each document 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 (e.g., = 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 for SEO folk!