

Part 5: Scoring, Term Weighting and the Vector Space Model



Francesco Ricci

Most of these slides comes from the
course:

Information Retrieval and Web Search,
Christopher Manning and Prabhakar
Raghavan



Content

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

Boolean 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: "*standard user dlink 650*" → 200,000 hits
- ❑ Query 2: "*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 – the **query language** and the **retrieval model** - but in practice, ranked retrieval models have normally been associated with free text queries.

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 k (≈ 10) results
 - We don't overwhelm the user
 - ***Premise: the ranking algorithm works***

Do you really agree with that?

Scoring as the basis of ranked retrieval

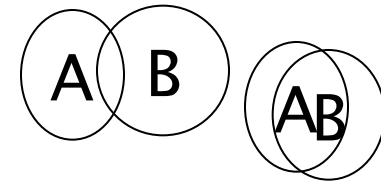
- We wish to return in order the documents most likely to be useful to the searcher
- How can we rank-order the documents in the collection with respect to a query?
- Assign a score – say in $[0, 1]$ – to each document
- This score measures how well document and query “match”.

Query-document matching scores

- 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:
 - The score should be 0
 - *Why? Can we do better?*
- **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

- A commonly used measure of overlap of two sets A and B
- $\text{jaccard}(A,B) = |A \cap B| / |A \cup B|$
- $\text{jaccard}(A,A) = 1$
- $\text{jaccard}(A,B) = 0$ if $A \cap B = 0$
- A and B don't have to be the same size
- Always assigns a number between 0 and 1
- We saw that in the context of k -gram overlap between two words.



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*

- $\text{jaccard}(Q,D) = |Q \cap D| / |Q \cup D|$

- $\text{jaccard}(\text{Query}, \text{Document1}) = 1/6$
- $\text{jaccard}(\text{Query}, \text{Document2}) = 1/5$

Issues with Jaccard for scoring

- It doesn't consider *term frequency* (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 also a more sophisticated way of **normalizing for length**
- Later in this lecture, we'll use $|A \cap B| / \sqrt{|A \cup B|}$
- . . . instead of $|A \cap B| / |A \cup B|$ (Jaccard) for length normalization.

Recall (Part 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

Bag of words model

- ❑ Vector representation doesn't consider the ordering of words in a document
- ❑ *"John is quicker than Mary"* and *"Mary is quicker than John"* 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
- ❑ We will look at "recovering" positional information later in this course
- ❑ For now: bag of words model.

Term frequency tf

- The term frequency $tf_{t,d}$ 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.

NB: *frequency = count* - in IR

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 t in both q and d :

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

- The score is 0 if none of the query terms is present in the document
- If $q' \subseteq q$ then $\text{score}(d, q') \leq \text{score}(d, q)$ – is this a problem?

Normal vs. Sublinear tf scaling

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

- The above formula defined the **sublinear tf-scaling**
- The simplest approach (**normal**) is to use the number of occurrences of the term in the document (frequency)
- But as discussed earlier sublinear tf should be better.

Properties of the Logarithms

- $y = \log_a x$ iff $x = a^y$
- $\log_a 1 = 0$
- $\log_a a = 1$
- $\log_a (xy) = \log_a x + \log_a y$
- $\log_a (x/y) = \log_a x - \log_a y$
- $\log_a (x^b) = b \log_a x$
- $\log_b x = \log_a x / \log_a b$
- $\log x$ is typically $\log_{10} x$
- $\ln x$ is typically $\log_e x$ ($e = 2.7182\dots$
Napier or Euler number) – Natural logarithm.

Document frequency

- Rare terms – in the whole collection - are 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*.

Document frequency, *cont'd*

- Generally frequent terms are **less informative** than rare terms
- Consider a **query** term that is frequent in the collection (e.g., *high*, *increase*, *line*)
- A document containing such a term is more likely to be relevant than a document that doesn't
- **But** consider a query containing two terms – e.g.: *high arachnocentric*
- For a **frequent** term in a document, s.a., *high*, we want a positive weight but **lower** than for terms that are **rare** in the collection, s.a., *arachnocentric*
- We will use **document frequency** (df) to capture this.

idf weight

- df_t is the document frequency of t : the number of documents that contain t
 - df_t is an inverse measure of the informativeness of t (*the smaller the better*)
 - $df_t \leq N$
- We define the idf (**inverse document frequency**) of t by

$$idf_t = \log(N/df_t)$$

Is a function of t only – does not depend on the document

- We use $\log(N/df_t)$ instead of N/df_t to “dampen” the effect of idf.

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

$$idf_t = \log (N/df_t)$$

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

Effect of idf on ranking

- Can idf have an effect on ranking for one-term queries? E.g. like:
 - iPhone
- idf has **no effect** on ranking **one term queries** - since there is one idf value for each term in a collection
 - 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**.

Collection vs. Document frequency

- The collection frequency of t is the number of occurrences of t in the collection, counting multiple occurrences
- Example:

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

- Which word is a better search term (and should get a higher weight)?

tf-idf weighting

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

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

- 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
- 1. **Increases** with the **number of occurrences within a document**
- 2. **Increases** with the **rarity of the term in the collection.**

Final ranking of documents for a query

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

We will see some other options for computing the score ...

Binary → count → weight matrix

		Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
dimensions	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:
 - 400,000 in RCV1
 - 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

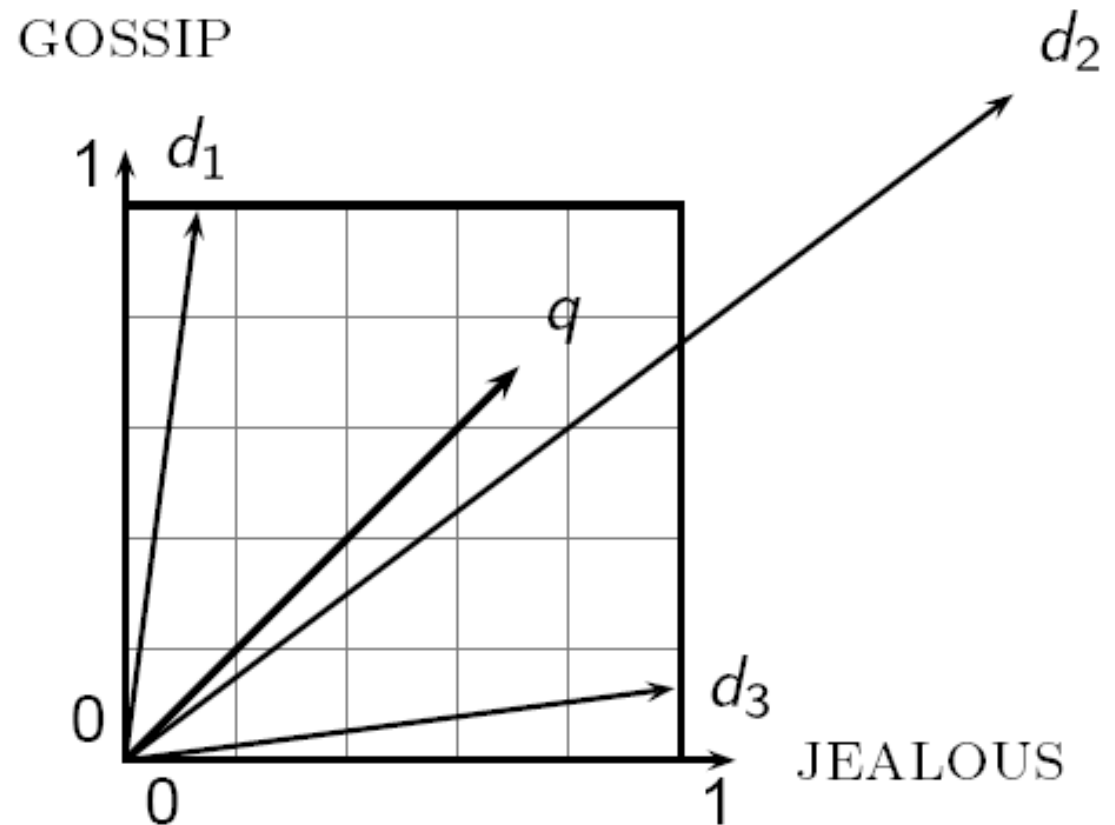
- Key idea 1: Do the same for queries: represent them as vectors in the space
- Key idea 2: Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors
- proximity \approx inverse of distance
- **Recall: We do this because we want to get away from the you're-either-in-or-out Boolean model**
- Instead: rank more relevant documents higher than less relevant documents.

Formalizing vector space proximity

- First cut: distance between two points
 - (= 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 q and d_2 is large even though the distribution of terms in the query q and the distribution of terms in the document d_2 are very similar.



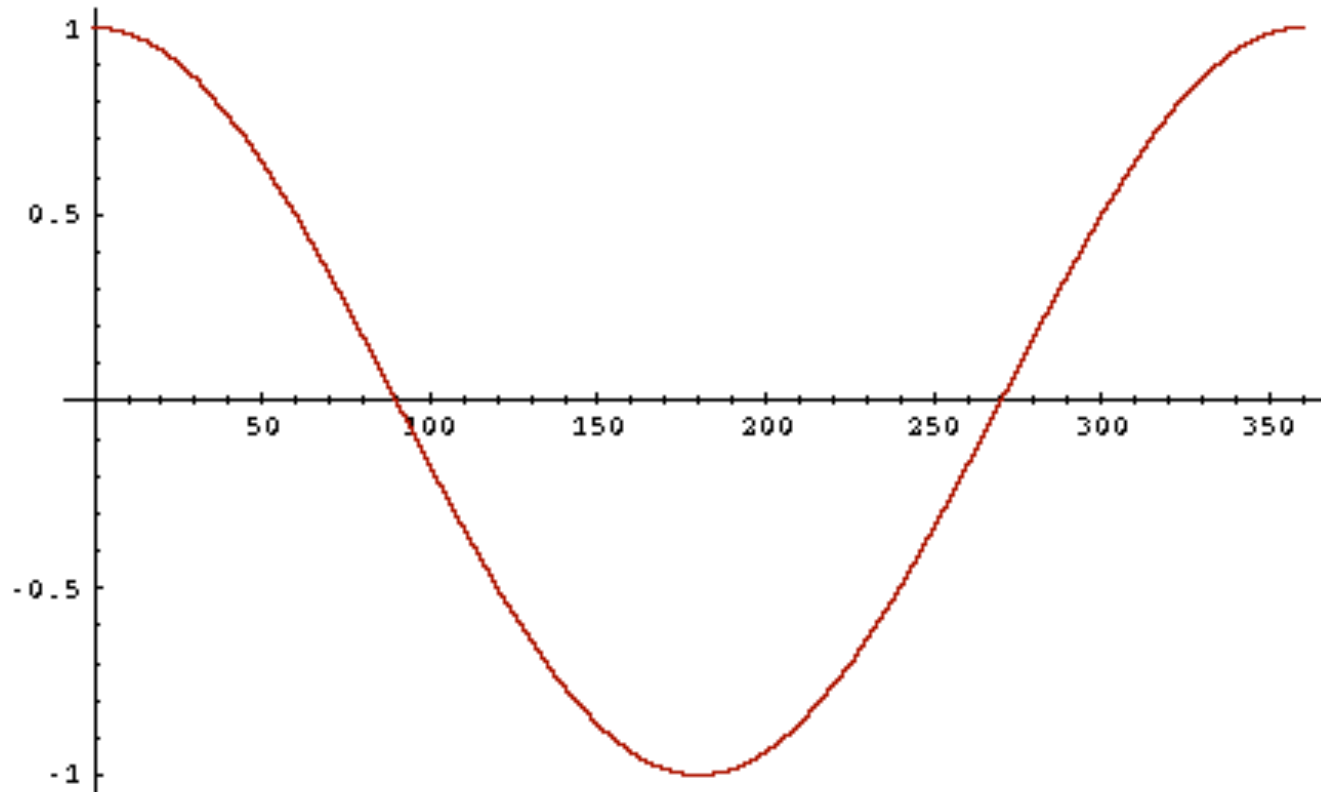
Use angle instead of distance

- 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
- If the $tf_{t,d}$ representation is used then 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 cosine (query,document)
- Cosine is a monotonically decreasing function for the interval $[0^\circ, 180^\circ]$

From angles to cosines



- But how – *and why* – 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)

$$\cos(\vec{q}, \vec{d}) = \frac{\overbrace{\vec{q} \cdot \vec{d}}^{\text{Dot product}}}{|\vec{q}| |\vec{d}|} = \frac{\overbrace{\vec{q}'}^{\text{Unit vectors}} \cdot \vec{d}}{|\vec{q}| |\vec{d}|} = \frac{\sum_{i=1}^{|\mathcal{V}|} q_i d_i}{\sqrt{\sum_{i=1}^{|\mathcal{V}|} q_i^2} \sqrt{\sum_{i=1}^{|\mathcal{V}|} d_i^2}}$$

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

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

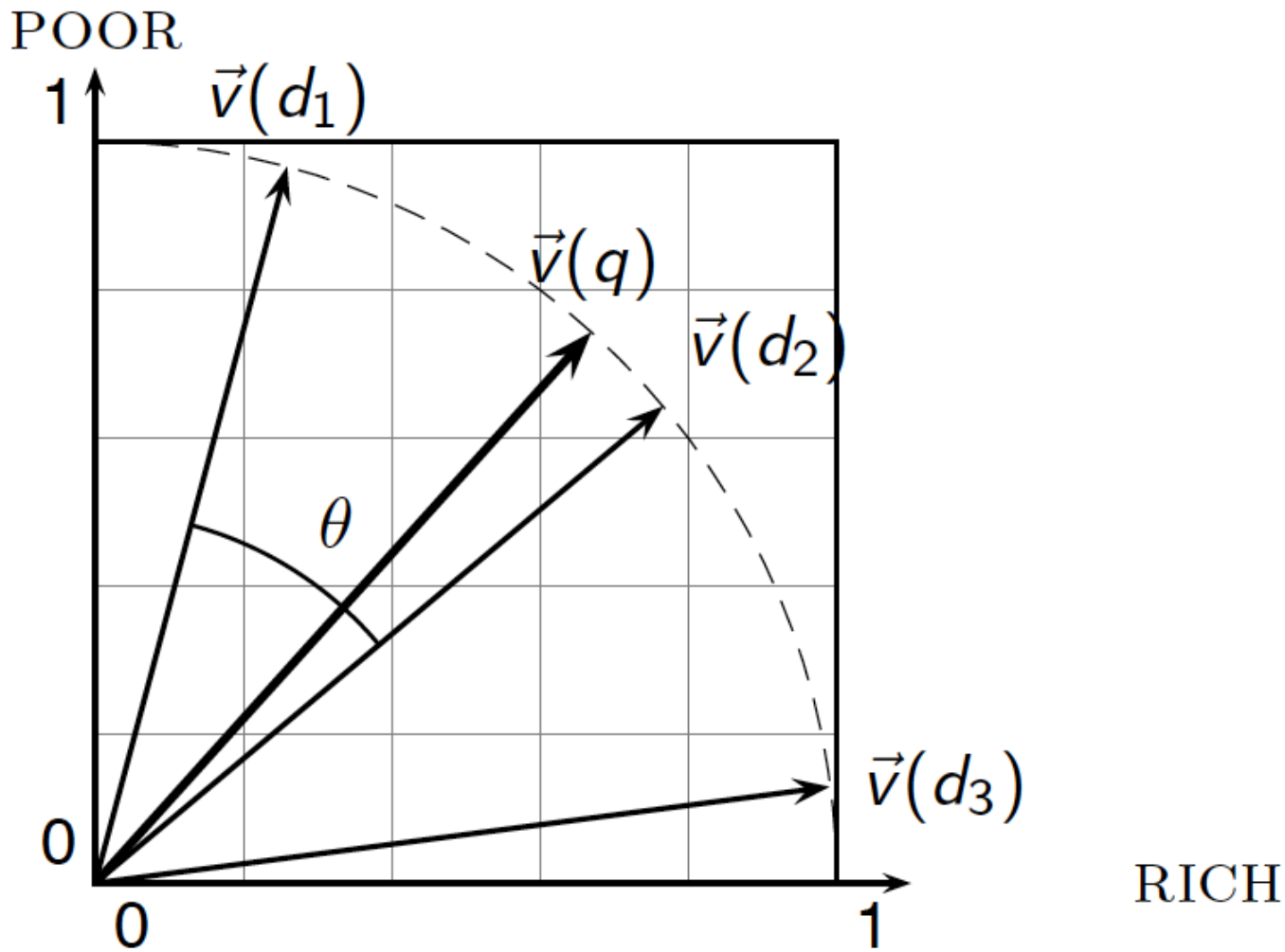
Cosine for length-normalized vectors

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

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

for q, d length-normalized.

Cosine similarity illustrated



Cosine similarity amongst 3 documents

How similar are
the novels

SaS: *Sense and
Sensibility*

PaP: *Pride and
Prejudice*, and

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, 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

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

$$0.789 \times 0.832 + 0.515 \times 0.555 + 0.335 \times 0.0 + 0.0 \times 0.0$$

$$\approx 0.94$$

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

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

Why do we have $\cos(\text{SaS}, \text{PaP}) > \cos(\text{SaS}, \text{WH})$?

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[]
```

of documents

This array contains
the vector lengths of
all the documents

Observations

- ❑ The inverted index – used for Boolean queries – is still important for retrieving top scoring documents
- ❑ It is not feasible to scan all the documents to compute the top K documents (more similar) for a given input query
- ❑ Queries are “normally” short documents



Request Entity Too Large

Your client issued a request that was too large.

Try with
a long
query!

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

Columns headed 'n' are acronyms for weight schemes.

Why is the base of the log in idf immaterial?

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 *ddd.qqq*, 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 ...

A bad idea?



tf-idf example: Inc.Itc

ddd.qqq

Document: *car insurance auto insurance*

Query: *best car insurance*

Term	Query						Document				Prod
	tf-raw	tf-wt	df	idf	wt	n'lize	tf-raw	tf-wt	wt	n'lize	
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

Exercise: what is N , the number of docs?

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

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

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 K (e.g., $K = 10$) to the user.



Reading Material

- ▣ Sections: 6.2, 6.3, 6.4 (excluded 6.4.4)