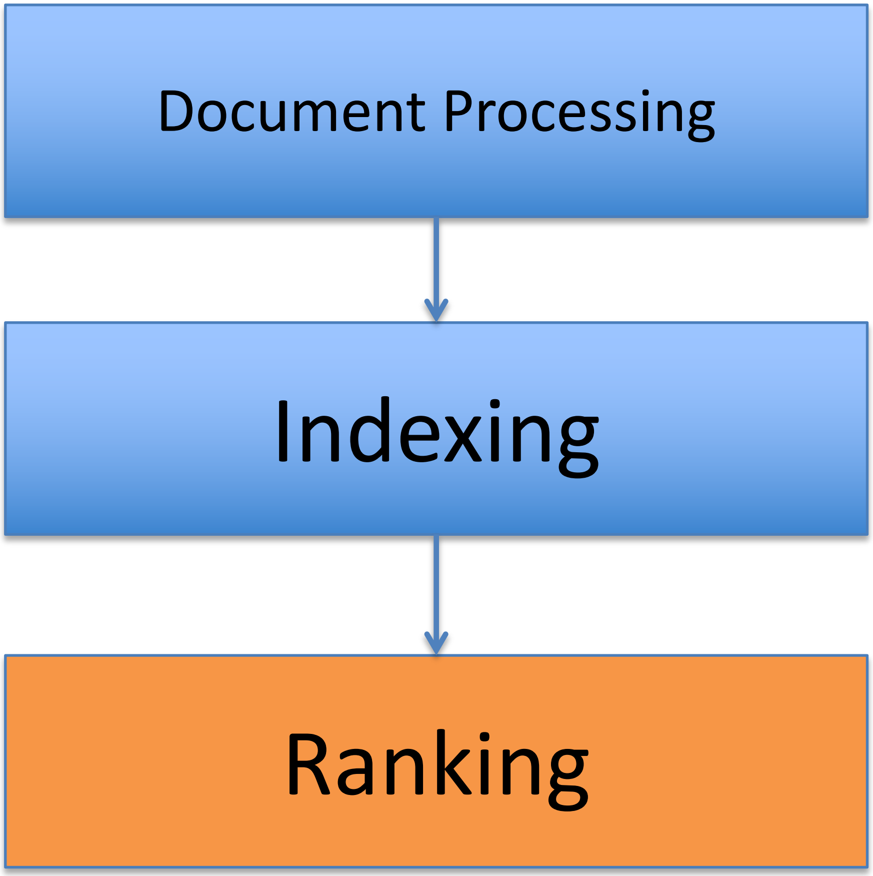
Basic ranking Models

Boolean and Vector Space Models


Document Processing
Indexing
Ranking

What is Ranking

- Indexing provides the set of documents which include the keywords of the user’s query
- Often many documents (millions, if the document archive is the full web) are possible “hits” for the user
- Ranking is an essential step to order possibly interesting documents in a why that fits at best the users’ actual information needs
- Ranking methods are where most of the current research institutions in the area of IR devote their effort (including Google)

What is ranking (2)

- Ranking algorithms are made of two components:
  - A **representation function** to represent documents and queries, starting from the set of indexed keywords in each document;
  - A **similarity function** $sim(q_i,d_j)$ to determine the similarity between query q_i and document d_j
  - The similarity function is the basis for establishing an order of relevance of documents w.r.t. the query
- We start with two “basic” ranking methods: the Boolean model and Vector Space model.

Boolean Model

The Boolean Model (1)

- Simple model based on *set theory*
- First model used in “classic” IR systems
- REPRESENTATION: Queries are represented as boolean expressions
  - *E.g., q = a $\wedge$ (b $\vee$ $\neg$c) (where a, b and c are keywords)*
  - *(apple $\wedge$ (computer $\vee$* $\neg$red)

The Boolean Model (2)

- According to *bag of word (BoW)* model, each document is represented by a fixed-lenght vector *dj*, position (*ij*) of the vector is the weight (relevance) of keyword i in document *j* *(i=1,2..|V|* where |V| dimension of vocabulary
- **In boolean model, terms are either present or absent, therefore**:
- $w_{ij} \in \{0,1\}$
- E.g., if the vocabulary is: {*apple, computer, red*} the document d1*: " the apple is red, red, red!"* is represented as*:* ***d1(1,0,1)***

  which is the same as:

  $\boldsymbol{apple \wedge (\neg computer) \wedge red}$

The Boolean Model (3)

The first step is to transform boolean query in Disjunctive Normal Form (DFN) = a disjunction of conjunctive componentes cc

$q = a \wedge (b \vee (\neg c)) = (a \wedge b \wedge c) \vee (a \wedge b \wedge (\neg c)) \vee (a \wedge (\neg b) \wedge (\neg c))$

Next, each conjunctive component of the DNF is transformed into a binary string

$v(qdnf) = (1,1,1)\ \boxed{(1,1,0)}\ (1,0,0)$

Ex., if a=apple, b=computer, c=red ➔ (apple AND (computer OR not(red)) ➔

(apple,computer,red) ∨ (apple, computer) ∨ (apple)

A DFN query *v(qdnf)* is now made of Conjunctive Components *v(qcc)* in the form of binary vectors: they represent the list of <u>all and only</u> possible matching document vectors

Example: v(qcc) = (1,1,0) ➔ equivalent to: (apple ∧ computer ∧ ¬(red))

All documents <u>including</u> apple and computer and <u>not including</u> red are matches for the initial query. Every v(qcc) defines a set of possibly matching documents.

Similarity/Matching function

sim(q,dj) = 1 ***iff*** *vec(dj) = v (qcc)*$_i$ *, v(qcc)*$_i \in$ *v(qdnf)*

0 otherwise

In other terms, matching documents are only those whose boolean vector (ignoring words not in the query) ***is equal to one of the conjunctive components*** *of the query disjunctive normal form*

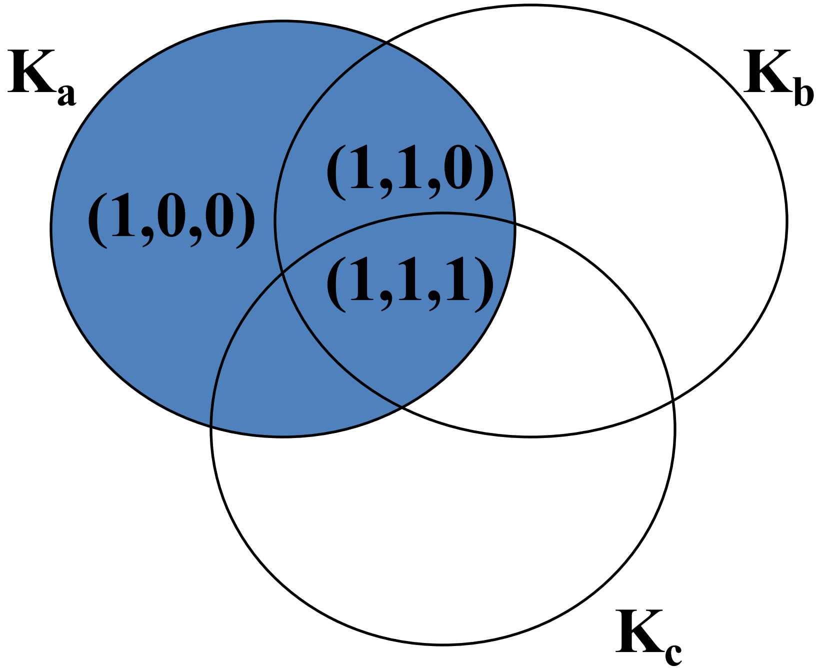
Example

v(qdnf) = (1,1,1) (1,1,0) (1,0,0)

- **For example, matching documents (md) for the above query are:**
  - *md1 = "apple apple blue day" => (1,0,0)*
  - *md2 = "apple computer red] => (1,1,1)*
- **Unmatched documents (ud)**
  - *ud1 = " apple red " => (1,0,1)*
  - *ud2 = "day" => (0,0,0)*

Note that words in documents not included in the query don't need to be represented in document and query vectors, they are «**don't care**» boolean variables!
(in the example, these irrelevant words are *blue day)*

Venn Diagram (K_i = generic keyword)



$$q = k_a \wedge (k_b \vee \neg k_c)$$

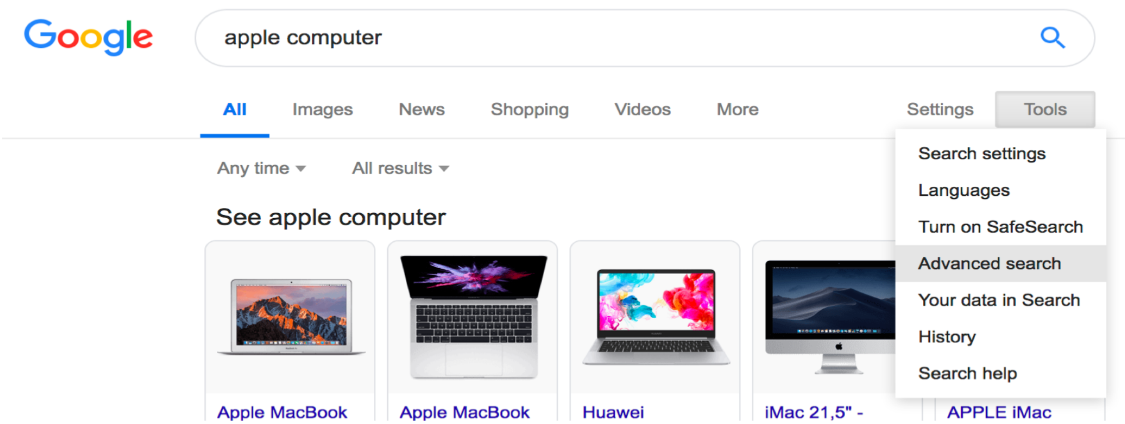
Drawbacks of the Boolean Model

- ❑Expressive power of boolean expressions to capture information needs and document semantics IS *inadequate*
- ❑Retrieval based on binary decision criteria (with no partial match) does not reflect our intuitions behind relevance adequately

- As a result
  - ❑Answer set contains either too few or too many documents in response to a user query
  - ❑No ranking of documents

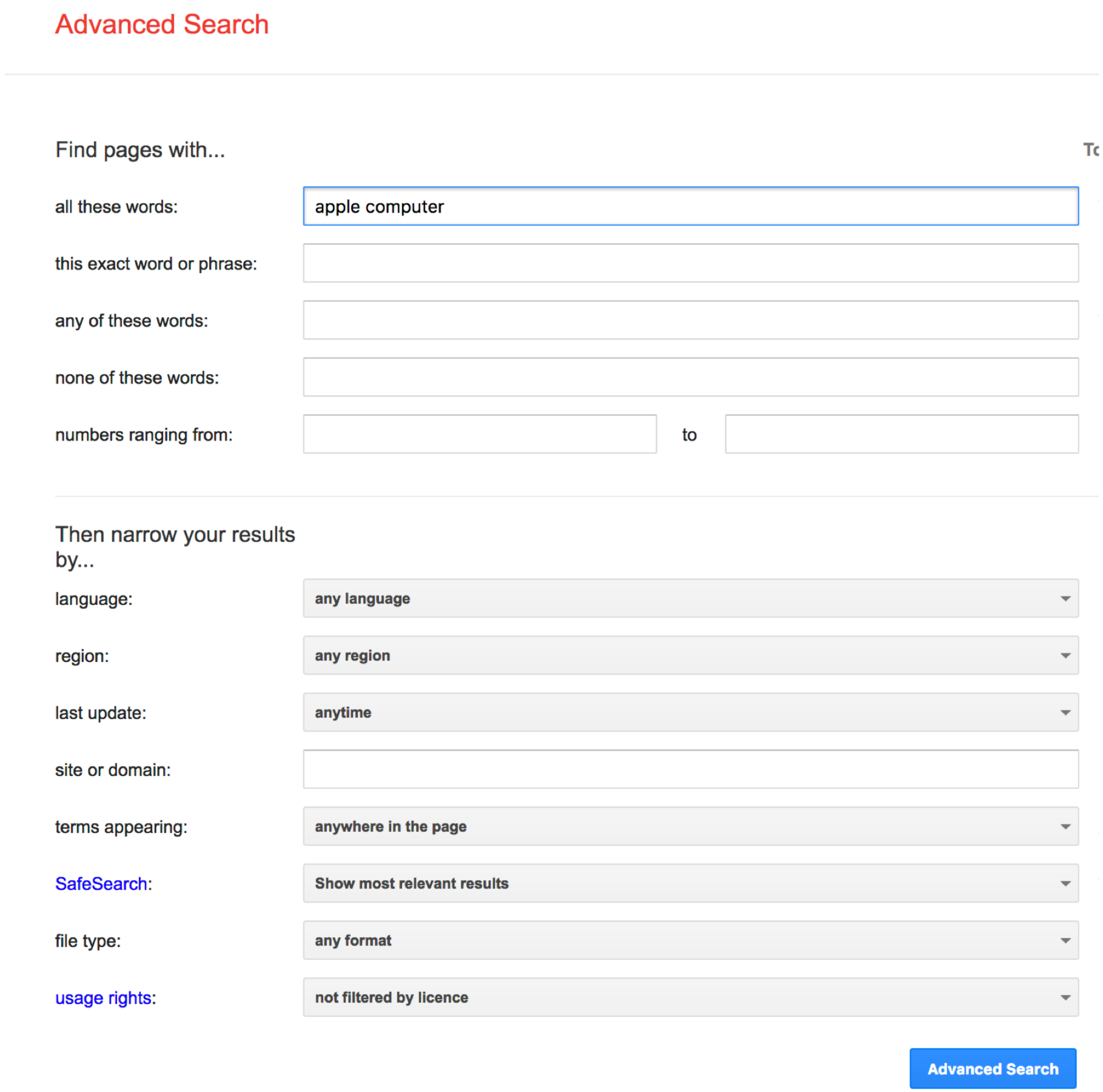
Boolean Search

- Boolean query almost disappeared from web search engines (not used by most users)
- "Advanced search" allows for other types of search



- However still used when users are motivated to search specific information (e.g., legal domains or medical domains)

Advanced search allows for boolean expressions + other types of contraints



Twitter search also exploits boolean and other operators

Search Operators

Operator	Finds tweets...
twitter search	containing both "twitter" and "search". This is the default operator.
"happy hour"	containing the exact phrase "happy hour".
love OR hate	containing either "love" or "hate" (or both).
beer -root	containing "beer" but not "root".
#haiku	containing the hashtag "haiku".
from:alexiskold	sent from person "alexiskold".
to:techcrunch	sent to person "techcrunch".
@mashable	referencing person "mashable".
"happy hour" near:"san francisco"	containing the exact phrase "happy hour" and sent near "san francisco".
near:NYC within:15mi	sent within 15 miles of "NYC".
superhero since:2010-12-27	containing "superhero" and sent since date "2010-12-27" (year-month-day).
ftw until:2010-12-27	containing "ftw" and sent up to date "2010-12-27".
movie -scary :)	containing "movie", but not "scary", and with a positive attitude.
flight :(	containing "flight" and with a negative attitude.
traffic ?	containing "traffic" and asking a question.
hilarious filter:links	containing "hilarious" and linking to URLs.
news source:twitterfeed	containing "news" and entered via TwitterFeed

Operators: OR, AND,NOT

But also:

Sent from (userid), sent to (userid),
Sent from (place) etc.

Vector Model

Ranked retrieval

- Thus far, our queries were Boolean.
  - Documents either match or don’t.
  - Good for expert users with precise understanding of their needs and the collection (e.g., legal search).
  - 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 (e.g., web search).

Problem with Boolean search

- 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 skill to come up with a query that produces a manageable number of hits.

- With **a ranked list of documents**, it does not matter how large the retrieved set is. User will looks only at first results.

Scoring as the basis of ranked retrieval

- We wish to return *in order of relevance* 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: 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.

Vector Space representation model

- **Model**: each document is a bag-of-words (as for boolean model)
- **Representation**: a **N-dimensional vector** (N=|V|, the dimension of the vocabulary (as for boolean)
- **Weighting scheme**: coordinate w_{ij} of vector d_j associated to document d_j is the RELEVANCE of word i in document j (as for boolean)
- How do we measure w_{ij} ? NOT as in boolean model!

Recap on Bag of words vector

- Vector representation doesn't consider the ordering of words in a document
  - *d1: John is quicker than Mary* and *d2: Mary is quicker than John* have the same vectors, since we have a coordinate (or coefficient, or weight) w_i for every word *i* of the vocabulary, and coordinates are ordered alphabetically
  - $d1=d2=(w_{John}, w_{is}, w_{Mary}, w_{quicker}, w_{than})$
- This is called (as we said) the bag of words model.
  - In a sense, this is a step back: the **positional index** (see lectures on indexing) was able to distinguish these two documents, since we know where words are placed.

Weighting schemes for w_i: Binary term-document **matrix**

documents

words	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**

Any column j is a document vector d_j.
Each document is represented by a binary vector $\in \{0,1\}^{|V|}$, w_{ij} is either 0 (word *i* is absent in d_j) or 1 (word *i* appears in d_j)
Number of rows=dimension of vocabulary **|V|**
Number of columns= dimension of the document collection **N**

Vector weighting scheme: Term-document **count matrix**

- This scheme considers the number of occurrences of a term in a document:
  - Each document is a count vector in $\mathbb{N}^v$

	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

Vector weighting scheme: 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 may be more relevant than a document with one occurrence of the term.
  - But not 10 times more relevant!!!
- **Relevance does not increase proportionally with term frequency**.
- One possibility is to **normalize, e.g.**:

$$tf_i^{norm} = tf_i / \max_j (tf_j)$$

Other vector weighting schemes (2): log-frequency weighting

- The log frequency weight of term t in doc d is

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

- 0 → 0, 1 → 1, 2 → 1.3, 10 → 2, 1000 → 4, etc.

When it comes to scoring, is frequency appropriate?

- Score for a document-query pair: sum over terms *t* **in both *q* and *d***:
- Sim(q,d) $= \sum_{t \in q \cap d} (1 + \log \mathrm{tf}_{t,d})$
  - **The score is 0 if none of the query terms is present in the document ($q \cap d = \emptyset$), and grows when the document includes many of the query terms, with a high frequency**
- However, frequency-based ranking (whether normalized or log) IS NOT FULLY APPROPRIATE
- WHY??

Improve document weighting scheme: Inverse Document frequency (1)

- **Rare terms are more informative than frequent terms**
  - Recall stop words! Are they so relevant? (e.g. “the”)
  - Consider instead 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 *“study on arachnocentric people”* (much more than the other query terms *study, people)*
  - → **We want a higher weight for rare terms like *arachnocentric (rare words are good at distinguishing document content)***

Improve document weighting scheme: Inverse Document frequency (2)

- Consider a document including “high” with frequency 5 and “serendipity” with frequency 1. Which one is more relevant to represent a document content?
  - “high” is more frequent, but is likely to be frequent in many other documents! Instead, “serendipity” may better characterize the content of the document.
  - For terms that are frequent in the entire collection, we want lower weights than for rare terms, since they do not characterize a single document
- We will use <u>document frequency</u> (*df*) to capture the intuition that terms appearing in many documents of the collection should have a lower weight
- df ($\leq N$) = number of documents that contain the term, N= dimension of the document collection

Improve document weighting scheme: Inverse Document frequency (3)

- df_t is the document frequency of t: the number of documents in the collection that contain t
  - df is a measure of the informativeness of t
- We define the idf (**inverse document frequency**) of t by:

$$\mathrm{idf}_t = \log_{10} N/\mathrm{df}_t$$

  - We use log N/df_t instead of N/df_t to "dampen" the effect of idf.

Will turn out that the base of the log is immaterial.

idf example, suppose N = 1 million

term	df_t = *# of documents including the term*	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

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

Digression: Collection vs. Document frequency

- The **in-collection frequency** of *a word i* is the number of occurrences of *i* in the collection, **counting multiple occurrences**.

Word	Collection frequency	Document frequency
insurance	10440	3997
try	10422	8760

- df_i measures the **document**, not the collection, frequency. +1 every times a document includes one ore more instances of word *i*.

Improve document weighting scheme: tf-idf

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

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

- 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

Binary → count → weight matrix

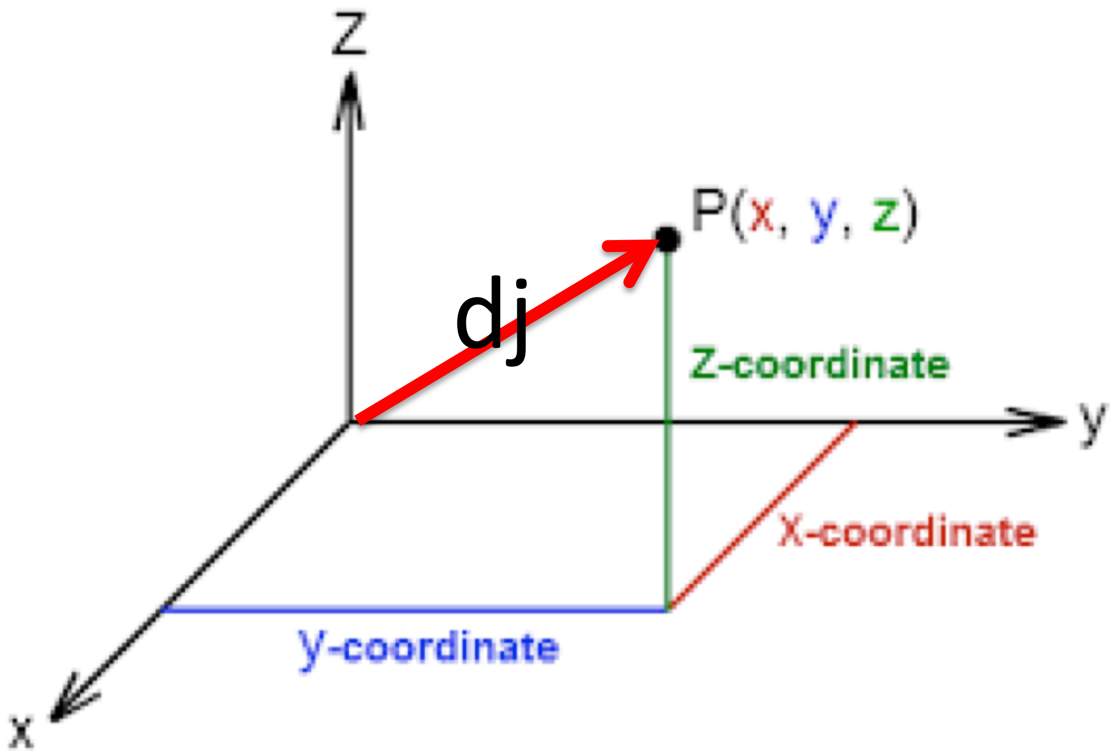
	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	5.25	3.18	0	0	0	0.35
Brutus	1.21	6.1	0	1	0	0
Caesar	8.59	2.54	0	1.51	0.25	0
Calpurnia	0	1.54	0	0	0	0
Cleopatra	2.85	0	0	0	0	0
mercy	1.51	0	1.9	0.12	5.25	0.88
worser	1.37	0	0.11	4.15	0.25	1.95

Each document is now represented by a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$

Geometric interpretation of VSM: documents as vectors

- The term-document matrix can be geometrically interpreted as a set of vectors (the documents) in a |V|-dimensional vector space, one dimension for each term.
- Terms are the axes of the vector space
- Documents are vectors in this space. The coordinate of a vector d_j on dimension i is the tf-idf weight of word i in document j.
- **Very high-dimensional**: hundreds of millions of dimensions when you apply this to a web search engine
- Very "sparse" vectors - most entries are zero (will see later in this course how to reduce dimensionality).

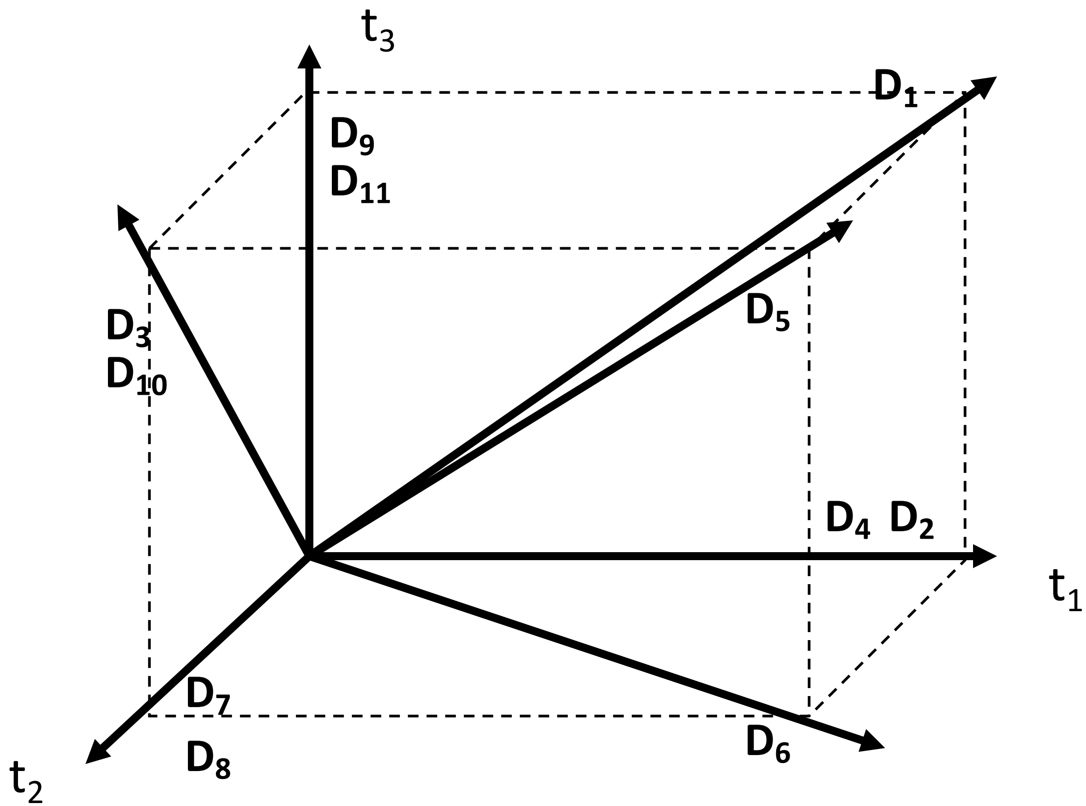
Vector space model (for |V|=3)



X, Y, Z are the 3 dimensions associated to keywords k_x, k_y, k_z

x, y, z are the 3 weights of keywords k_x, k_y, k_z in d_j

Documents in Vector Space



Scoring similarity between document and query: vector space **scoring** model

- 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 ≈ inverse of distance

Vector Space scoring model: formalizing "vector space proximity"

- First cut: distance between two points

  (= distance between the end points of the two vectors)

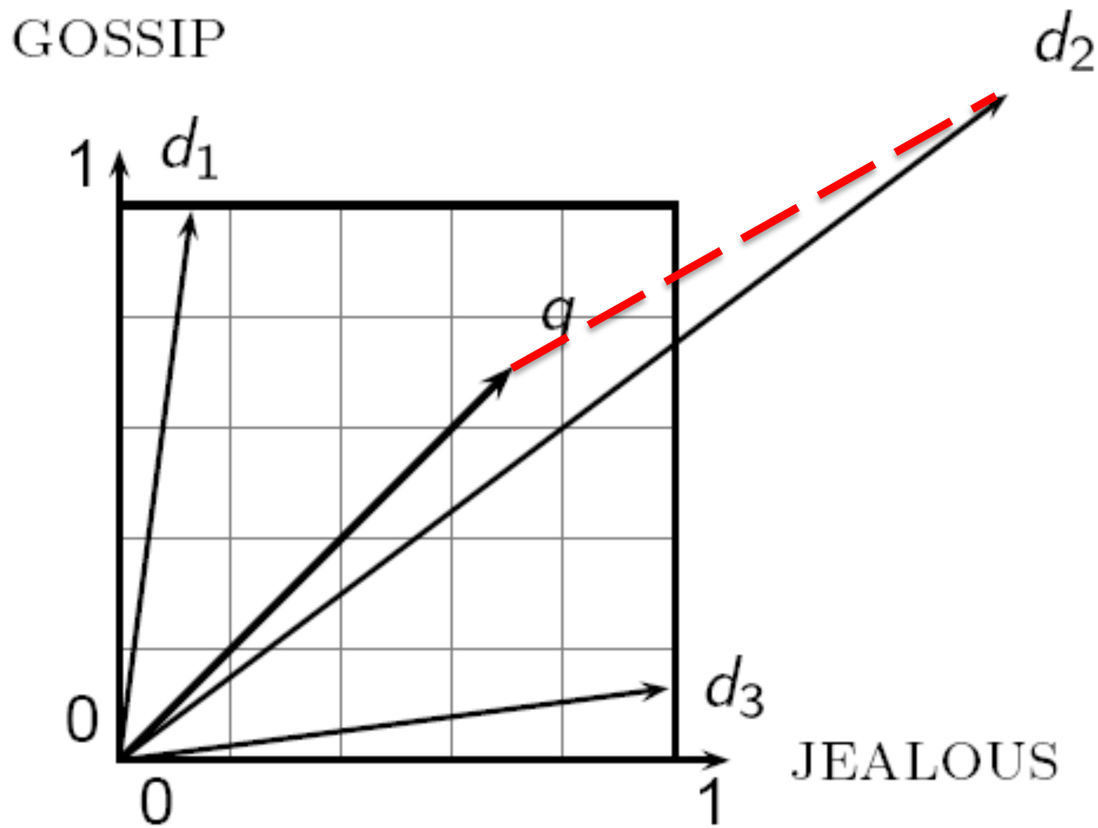
- Euclidean distance?

$$d(d_j, q) = \sqrt{\sum_i (w_{ij} - w_{iq})^2}$$

- Euclidean distance is a bad idea . . .
- . . . because Euclidean distance is large for vectors of different lengths.

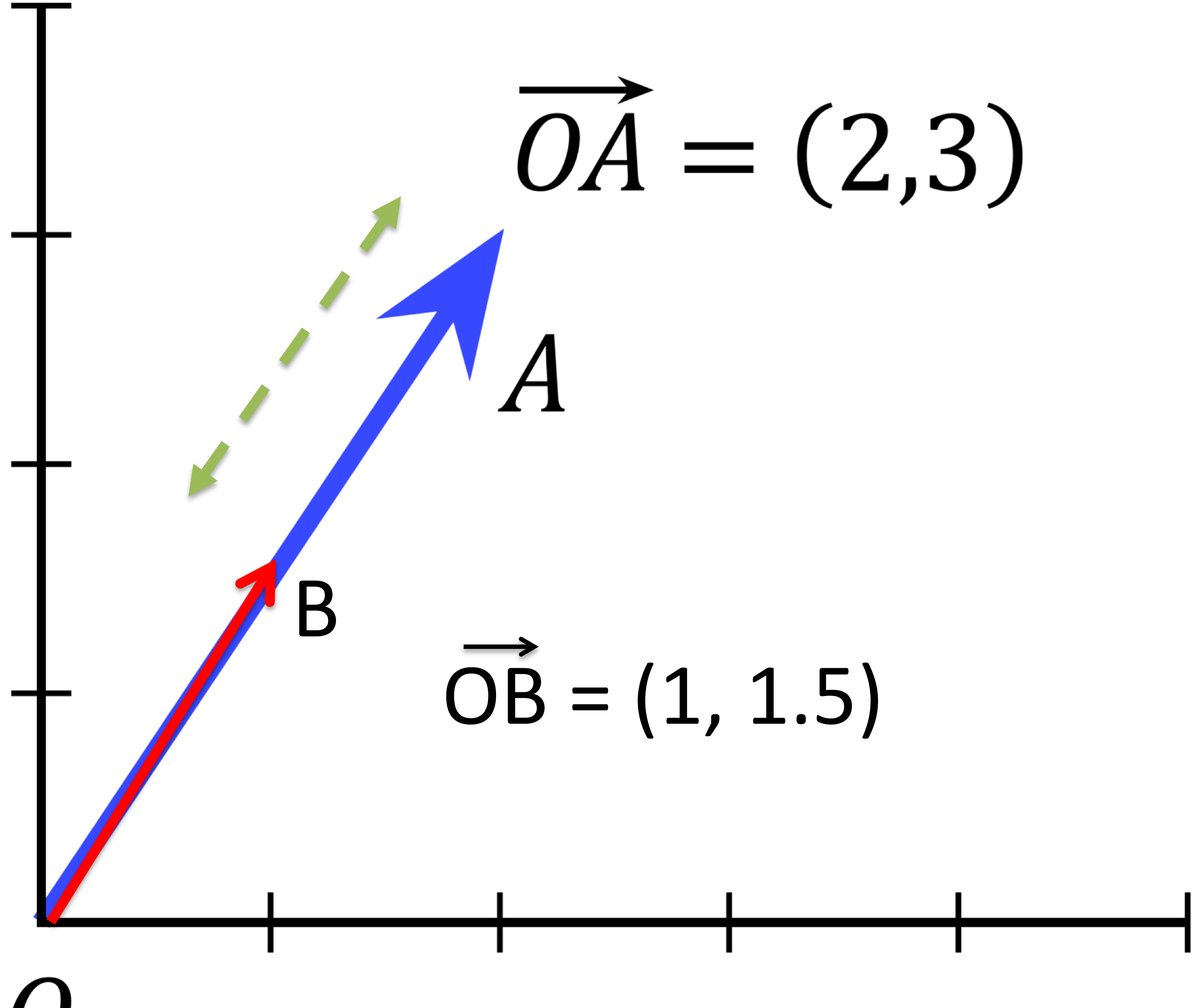
Why Euclidean distance is a bad idea

The Euclidean distance between $\vec{q}$ and $\vec{d_2}$ (red dashed line) is large even though the distribution of terms in the query $\vec{q}$ and the **distribution** of terms in the document $\vec{d_2}$ are very similar (about 50% “gossip”, 50% “Jealous”). Absolute frequencies cause the difference.



Why Euclidean distance is a bad idea (2)

- 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 (word frequency doubles in d')



Document A is document B appended to itself

A better distance measure: the angle between two vectors

- In previous example, the angle between the two documents is 0.
- Key idea: Rank documents according to angle with query.
- In previous example, the angle is zero, corresponding to maximum similarity!
- In fact the two documents have the same words, with same relative weight.
- Small angle = similar distribution of keywords in the document

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^o, 180^o]
- Cosine is 0 when vectors are orthogonal (no words in common!) , cosine is 1 when they are parallel (same distribution of keywords – not same frequency)

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:

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

  - Dividing a vector by its L_2 norm makes it a unit (length) vector
  - Effect on the two documents d and d' (where d' is d appended to itself) from earlier slide: they have **identical vectors** after length-normalization.

Vector Space Model: the cosin-similarity

Dot product

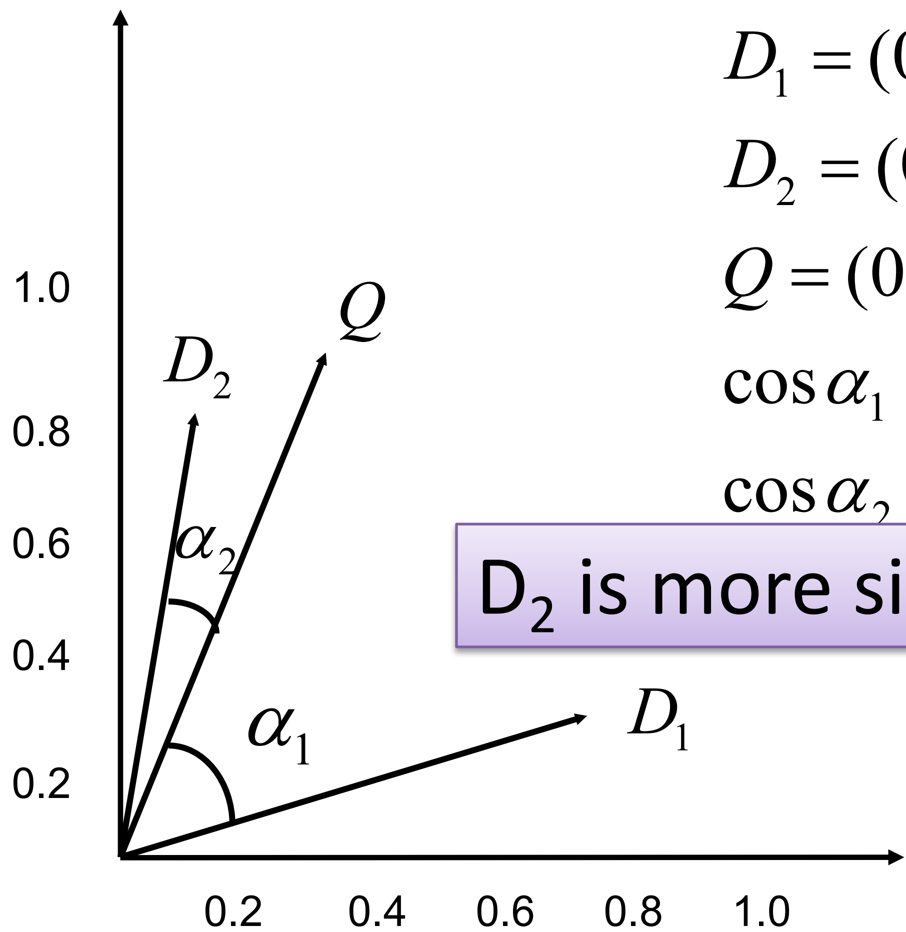
Unit vectors

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q} \bullet \vec{d}}{|\vec{q}||\vec{d}|} = \boxed{\frac{\vec{q}}{|\vec{q}|}} \bullet \frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

q_i is the tf-idf **weight** of term *i* in the query (also denoted as $w_{i,q}$)
d_i is the tf-idf **weight** of term *i* in the document (also denoted as $w_{i,d}$)
$\cos(\vec{q},\vec{d})$ is the cosine similarity of $\vec{q}$ and $\vec{d}$ … or,
equivalently, the cosine of the angle between $\vec{q}$ and $\vec{d}$.

Cosin-similarity is the **cosin of the angle between normalized query end document vectors**.

Examples: Computing Similarity Scores



$$D_1 = (0.8, 0.3)$$

$$D_2 = (0.2, 0.7)$$

$$Q = (0.4, 0.8)$$

$$\cos \alpha_1 = 0.74$$

$$\cos \alpha_2 = 0.98$$

D_2 is more similar to Q than D_1!!

A complete example

A small collection of N=3 documents, |V|=6 words

d1: "new york times"
d2: "new york post"
d3: "los angeles times"

Compute idf

angles	$log_2(3/1)=1.584$
los	$log_2(3/1)=1.584$
new	$log_2(3/2)=0.584$
post	$log_2(3/1)=1.584$
times	$log_2(3/2)=0.584$
york	$log_2(3/2)=0.584$

A complete example

Document-term matrix (we use **normalized tf**, however here each word appears just once in each document)

	angeles	los	new	post	times	york
d1	0	0	1	0	1	1
d2	0	0	1	1	0	1
d3	1	1	0	0	1	0

tf-idf: multiply tf by idf values

	angeles	los	new	post	times	york
d1	0	0	0.584	0	0.584	0.584
d2	0	0	0.584	1.584	0	0.584
d3	1.584	1.584	0	0	0.584	0

A complete example (2)

Query: “new new times”

When computing the *tf-idf* values for the query terms we divide the frequency by the maximum frequency (2) to normalize, and multiply with the *idf* values

q	0	0	(2/2)*0.584=0.584	0	(1/2)*0.584=0.292	0

We calculate the length (the NORM) of each document vector and of the query:

Length of d1 = sqrt(0.584^2+0.584^2+0.584^2)=1.011
Length of d2 = sqrt(0.584^2+1.584^2+0.584^2)=1.786
Length of d3 = sqrt(1.584^2+1.584^2+0.584^2)=2.316
Length of q = sqrt(0.584^2+0.292^2)=0.652

A complete example (3)

Similarity values are computed using cosin-sim formula:

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q}\bullet\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\vec{q}}{|\vec{q}|}\bullet\frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

cosSim(d1,q) = (0*0+0*0+0.584*0.584+0*0+0.584*0.292+0.584*0) / (1.011*0.652) = 0.776
cosSim(d2,q) = (0*0+0*0+0.584*0.584+1.584*0+0*0.292+0.584*0) / (1.786*0.652) = 0.292
cosSim(d3,q) = (1.584*0+1.584*0+0*0.584+0*0+0.584*0.292+0*0) / (2.316*0.652) = 0.112

According to the computed similarity values, the final order in which the documents are presented as result to the query will be: d1, d2, d3.

Cos-sim can be used also to measure similarity between documents

How similar are the novels:

SaS: *Sense and Sensibility*

PaP: *Pride and Prejudice*, and

WH: *Wuthering Heights*?

Cosine similarity amongst 3 documents

term	SaS	PaP	WH
affection	115	58	20
jealous	10	7	11
gossip	2	0	6
wuthering	0	0	38

Term frequencies (counts)

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

Tf-idf and normalize

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(SaS,PaP) ≈
$0.789 * 0.832 + 0.515 * 0.555 + 0.335 * 0.0 + 0.0 * 0.0$
≈ 0.94
cos(SaS,WH) ≈ 0.79
cos(PaP,WH) ≈ 0.69

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 as the normalized dot product $$\frac{\vec{q} \bullet \vec{d}}{|\vec{q}||\vec{d}|}$$
- Rank documents with respect to the query by score
- Return the top *K* (e.g., *K* = 10) to the user

Computing cosine scores efficiently

- Input:
  - Query
  - Posting list of document collection
  - Remember:

    $(t_i, df_i) \Rightarrow [(Doc\_ID_1, tf_{(i,1)}), (Doc\_ID_2, tf_{(i,2)}), ecc]$
- Note: **The posting list has all the information that we need to calculate the similarity scores**
- Output: List of K top ranked documents

Computing cosine scores

- We are going to compute the cos-sim scores, but in a "clever" way
- Here are some constants that we need:
  - The number N of documents in the corpus
  - The *document frequency* df_i of each term t_i (this is equals the number of ITEMS in the posting list of t_i)
  - The *term frequency* tf_i of each term t_i in a document (which is the second argument of a (Doc_ID,tf) pair in the posting list)

Remember the cos-sim formula

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q}\bullet\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\vec{q}}{|\vec{q}|}\bullet\frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

$$\cos(\vec{q},\vec{d}) = \frac{\sum_{i=1}^{|V|} tfidf_{i,q}\,\boxed{tfidf_{i,d}}}{\sqrt{\sum_{i=1}^{|V|} tfidf_{i,q}^2}\sqrt{\sum_{i=1}^{|V|} tfidf_{i,d}^2}}$$

The coordinates q_i,d_i of the **q** and **d** vectors are the tfidf values for term *i* of numerator

Computing cosine scores

1. Get a query from user (e.g. "*information retrieval for retrieval of documents*")
2. After removal of stop words and stemming, we have 3 terms*: information, retrieval, document*
3. Compute *tfidf* for those terms (e.g. using log-tf)

Compute tf*idf for query terms (e.g. using logs)

$$tfidf(t,q) \quad = \quad WTF(t,q) * log\left(\frac{|corpus|}{df_{t,q}}\right)$$

$\mathrm{WTF}(t,q)$

if $tf_{t,q} = 0$
  then $return(0)$
  else $return(1 + log(tf_{t,q}))$

WTF(information)=(1+log(2))
WTF(retrieval)=1
WTF(document)=1

|corpus| is number of documents in archive

Computing cosine scores for documents

$$\cos(\vec{q},\vec{d}) = \frac{\sum_{i=1}^{|V|} tfidf_{i,q} tfidf_{i,d}}{\boxed{\sqrt{\sum_{i=1}^{|V|} tfidf_{i,q}^2}}\sqrt{\sum_{i=1}^{|V|} tfidf_{i,d}^2}}$$

- Note in the formula that we have a numerator which is the sum of tfidf, and a "normalizing" denominator which is a **product of the square of the sum** of (tfidf)^2
- We can compute numerator and denominator **incrementally and separately**.
- Define two variables, *Score* and *Magnitude*: the first to compute numerator, the second to compute denominator (vector norm)
- For each keyword t_i in the query:
  1. Get **posting list** for that word
  2. For each document dj in posting list of keyword ti, **update** the entry in Score(dj,q): ***Score(dj,q)=Score(dj,q)+tfidf(ti,q)*tfidf(ti,dj)***
  3. We also need to compute the NORMs of dj (denominator), and we do this incrementally, as well:

     Magnitude(dj)=Magnitude(dj)+*tfidf(ti,dj)^2*

NOTE WE DO NOT NEED TO COMPUTE THE NORM OF q SINCE THIS VALUE IS THE SAME FOR ALL cos(q,d) and does not affect the ranking order

Algorithm to compute cosin similarity scores

Initialize(Scores [d in Collection])

Initialize (Magnitude [d in Collection]))

***For** each keyword **t** in query q:*

Fetch df_t

Fetch posting list of t, p(t)

Compute $tfidf_{t,q}$ for the query

***For** each **d** in p(t):*

Compute $tfidf_{t,d}$

*Score(d)= Score(d)+$tfidf_{t,q}$*$tfidf_{t,d}$*

Magnitude(d)= Magnitude(d)+($tfidf_{t,d}$)^2

***For** d in Scores:*

***Do** NORMALIZE (Scores(d)/SQRT(Magnitude(d)))*

***Return** top K scores*

tfidf weighting has many variants

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

Augmented used to assign same relevance to very rare words

Weighting may differ in queries vs documents

- Many search engines allow for different weightings for queries vs documents (queries are very short, each word occurs typically once)
- To denote the specific combination in use in an search engine, we use the notation qqq.ddd with the acronyms from the previous table
- Example: ltn.lnc means:
  - Query: logarithmic tf (l in leftmost column), idf (t in second column), no normalization …
  - Document: logarithmic tf, no idf and cosine normalization

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

- A well-formed algebraic space for retrieval
- 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.
  - Documents and queries are expressed as ***bags of words***

Summary: What's the point of using vector spaces? (2)

- Non-binary (numeric) term weights used to compute *degree of similarity* between a query and each of the documents.
- Enables
  - ❑*partial matches*
    - to deal with incompleteness
  - ❑*answer set ranking*
    - to deal with information overload

The Vector Model : Pros and Cons

- Advantages:
  - ❑term-weighting improves answer set quality
  - ❑partial matching allows retrieval of docs that approximate the query conditions
  - ❑cosine ranking formula sorts documents according to degree of similarity to the query
- Disadvantages:
  - ❑assumes independence of index terms, which is a step back from proximity search

Google ranking method

- Ranking is (also) based on the content and on the specific page (later in this course, PageRank)
- Unknown, but over 200 methods/algorithms are jointly used. PageRank is one of the 200!
- In a query, keywords are interpreted as a boolean AND search (advanced options for complex boolean queries)
- However, answers are returned even if a word is not included (basically, it is a mixed boolean-vector space model)
- Additionally, query words are spell-corrected, and additional words can be added (see Query Expansion)