Advanced Smoothing

Good-Turing smoothing intuition

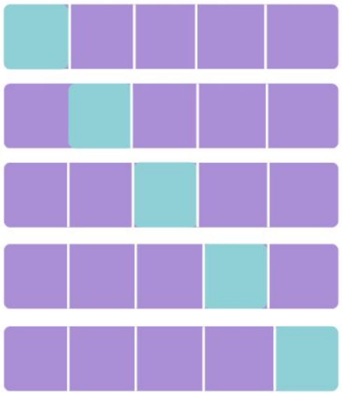
- You are fishing (a scenario from Josh Goodman), and caught:
  - 10 carp, 3 perch, 2 whitefish, 1 trout, 1 salmon, 1 eel = 18 fish
- How likely is it that next species is trout?
  - 1/18
- How likely is it that next species is new (i.e. catfish or bass)
  - Let's use our estimate of things-we-saw-once to estimate the new things.
  - 3/18 (because N_1=3)
- Assuming so, how likely is it that next species is trout?
  - Must be less than 1/18
  - How to estimate?

Good Turing calculations

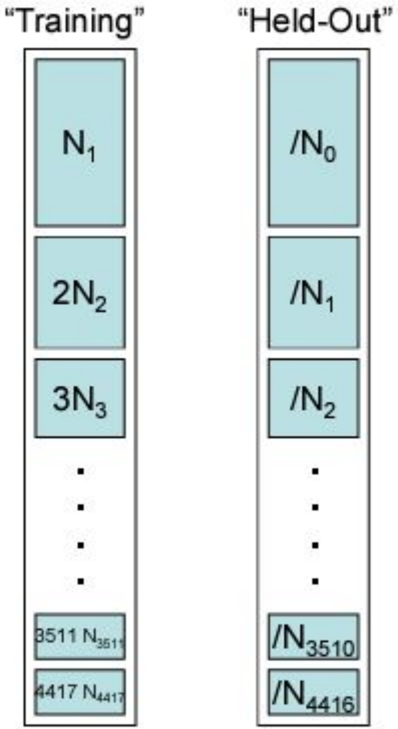
$$P^*_{GT}(\text{things with zero frequency}) = \frac{N_1}{N} \qquad c^* = \frac{(c+1)N_{c+1}}{N_c}$$

- Unseen (bass or catfish)
  - c = 0:
  - MLE p = 0/18 = 0
  - P^*_{GT} (unseen) = N_1/N = 3/18
- Seen once (trout)
  - c = 1
  - MLE p = 1/18
  - C*(trout) = 2 * N2/N1
    = 2 * 1/3
    = 2/3
  - P^*_{GT}(trout) = 2/3 / 18 = 1/27

Proof Sketch



- N_k: number of types which occur k times in the entire corpus
- Take each of the c tokens out of corpus in turn
- c "training" sets of size c-1, "held-out" of size 1
- How many "held-out" tokens are unseen in "training"?
  - N_1
- How many held-out tokens are seen k times in training?
  - $(k+1)N_{k+1}$
- There are N_k words with training count k
- Each should occur with expected count
  - $(k+1)N_{k+1}/N_k$
- Each should occur with probability:
  - $(k+1)N_{k+1}/(cN_k)$



Absolute discounting: just subtract a little from each count

- Church and Gale (1991)'s clever idea
- Divide up 22 million words of AP Newswire
  - Training and held-out set
  - for each bigram in the training set
  - see the actual count in the held-out set!
- It sure looks like $c^* = (c - .75)$

Bigram count in training	Bigram count in heldout set
0	.0000270
1	0.448
2	1.25
3	2.24
4	3.23
5	4.21
6	5.23
7	6.21
8	7.21
9	8.26

Absolute Discounting Interpolation

- Save ourselves some time and just subtract 0.75 (or some d)!

discounted bigram

Interpolation weight

$$P_{\text{AbsoluteDiscounting}}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i) - d}{c(w_{i-1})} + \lambda(w_{i-1})P(w)$$

unigram

- (Maybe keeping a couple extra values of d for counts 1 and 2)
- But should we really just use the regular unigram P(w)?

Kneser-Ney Smoothing I

- Better estimate for probabilities of lower-order unigrams!
  - Shannon game: *I can't see without my reading*____________? Kong glasses
  - "Kong" turns out to be more common than "glasses"
  - … but "Kong" always follows "Hong"
- The unigram is useful exactly when we haven't seen this bigram!
- Instead of P(w): "How likely is w"
- $P_{\text{continuation}}(w)$: "How likely is w to appear as a novel continuation?
  - For each word, count the number of bigram types it completes
  - Every bigram type was a novel continuation the first time it was seen

$$P_{CONTINUATION}(w) \propto \left|\{w_{i-1} : c(w_{i-1}, w) > 0\}\right|$$

Kneser-Ney Smoothing II

- How many times does w appear as a novel continuation:

$$P_{CONTINUATION}(w) \propto \left|\{w_{i-1} : c(w_{i-1}, w) > 0\}\right|$$

- Normalized by the total number of word bigram types

$$\left|\{(w_{j-1}, w_j) : c(w_{j-1}, w_j) > 0\}\right|$$

$$P_{CONTINUATION}(w) = \frac{\left|\{w_{i-1} : c(w_{i-1}, w) > 0\}\right|}{\left|\{(w_{j-1}, w_j) : c(w_{j-1}, w_j) > 0\}\right|}$$

Kneser-Ney Smoothing: Recursive formulation

$$P_{KN}(w_i \mid w_{i-n+1}^{i-1}) = \frac{\max(c_{KN}(w_{i-n+1}^{i}) - d, 0)}{c_{KN}(w_{i-n+1}^{i-1})} + \lambda(w_{i-n+1}^{i-1}) P_{KN}(w_i \mid w_{i-n+2}^{i-1})$$

$$c_{KN}(\bullet) = \begin{cases} count(\bullet) & \text{for the highest order} \\ continuationcount(\bullet) & \text{for lower order} \end{cases}$$

Continuation count = Number of unique single word contexts for •

Smoothing for Web-scale N-grams

- "Stupid backoff" (Brants *et al*. 2007 from Google)
- No discounting, just use relative frequencies

$$S(w_i \mid w_{i-k+1}^{i-1}) = \begin{cases} \dfrac{\text{count}(w_{i-k+1}^{i})}{\text{count}(w_{i-k+1}^{i-1})} & \text{if } \text{count}(w_{i-k+1}^{i}) > 0 \\ 0.4 S(w_i \mid w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases}$$

$$S(w_i) = \frac{\text{count}(w_i)}{N}$$

Vector Space Model and Information Retrieval

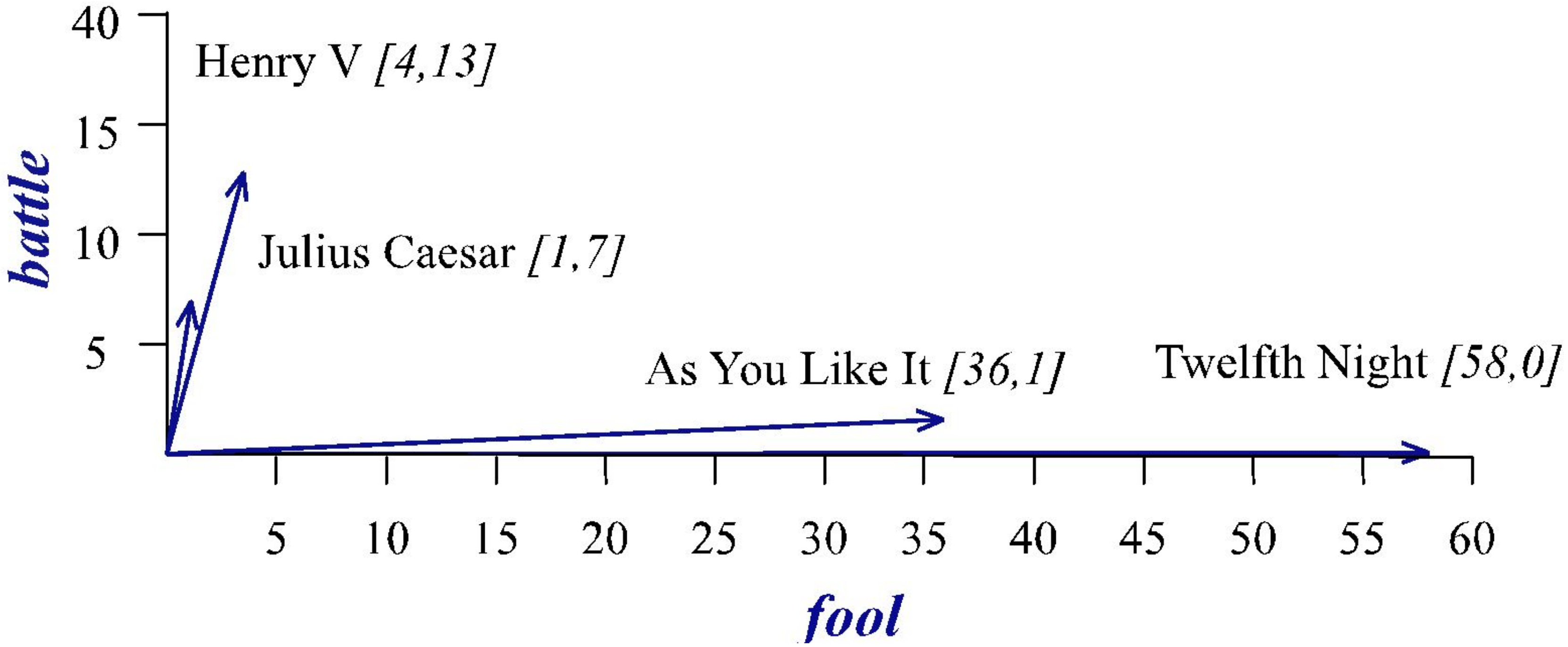
Sparse Vector

Term-document matrix

Each document is represented by a vector of words

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Visualizing document vectors



Vectors are the basis of information retrieval

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Vectors are similar for the two comedies
Different than the history

Comedies have more *fools* and *wit* and fewer *battles*.

Idea for word meaning: Words can be vectors too!!!

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

battle is "the kind of word that occurs in Julius Caesar and Henry V"

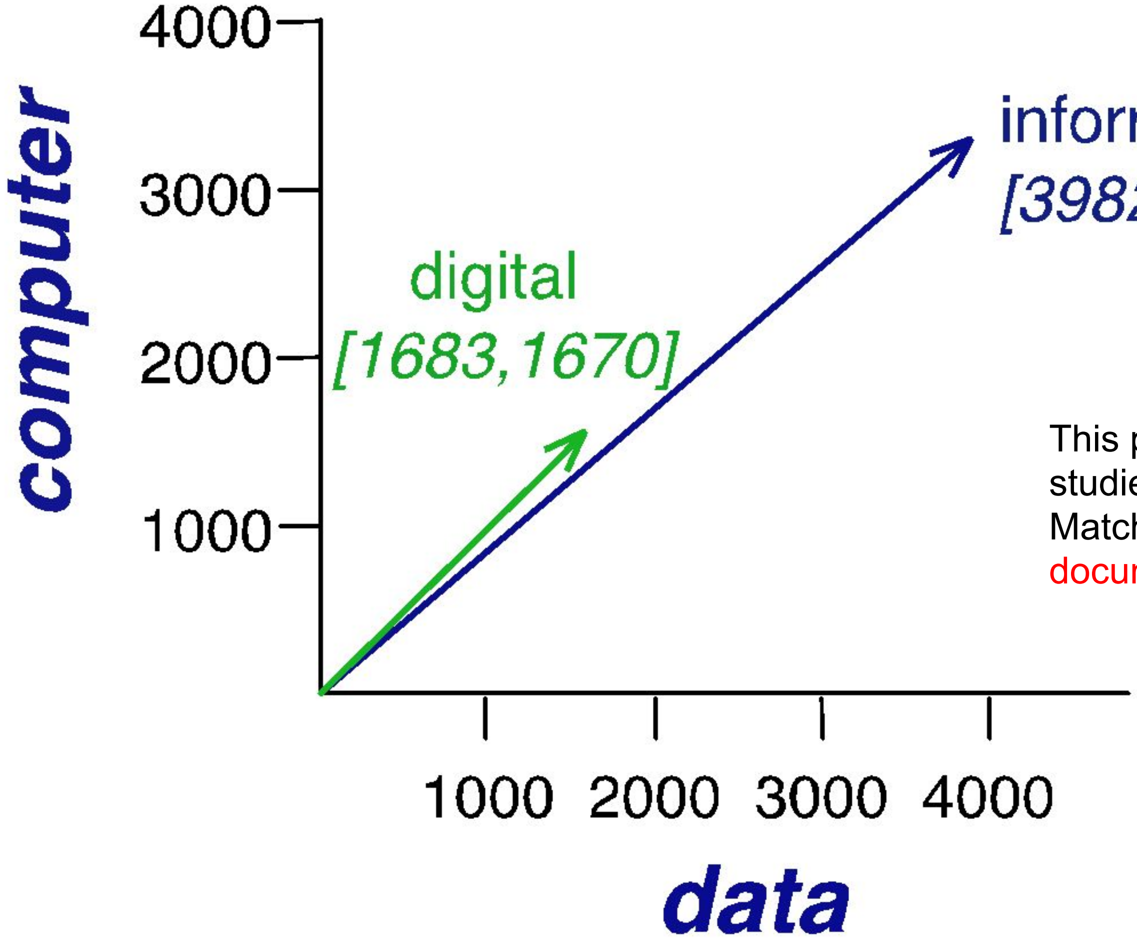
fool is "the kind of word that occurs in comedies, especially Twelfth Night"

More common: word-word matrix (or "term-context matrix")

- Two **words** are similar in meaning if their context vectors are similar

is traditionally followed by	**cherry**	pie, a traditional dessert
often mixed, such as	**strawberry**	rhubarb pie. Apple pie
computer peripherals and personal	**digital**	assistants. These devices usually
a computer. This includes	**information**	available on the internet

	aardvark	...	**computer**	**data**	**result**	**pie**	**sugar**	...
cherry	0	...	2	8	9	442	25	...
strawberry	0	...	0	0	1	60	19	...
digital	0	...	1670	1683	85	5	4	...
information	0	...	3325	3982	378	5	13	...



This problem was heavily studied in the context of Matching query and document

Web Search: Boolean ⍰ Ranked retrieval

- Old system: query-doc match is 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

- Exact match 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

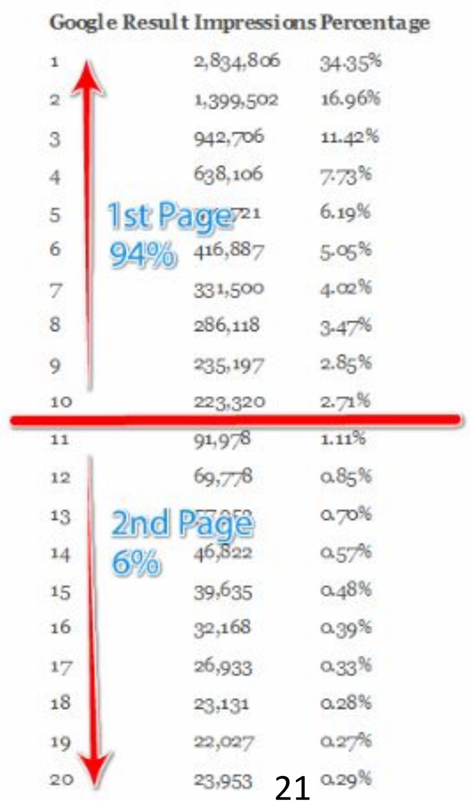
Ranked retrieval models

- Rather than a set of documents satisfying a query expression, in ranked retrieval, 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

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



Google Result	Impressions	Percentage
1	2,834,806	34.35%
2	1,399,502	16.96%
3	942,706	11.42%
4	638,106	7.73%
5	[illegible]	6.19%
6	416,887	5.05%
7	331,500	4.02%
8	286,118	3.47%
9	235,197	2.85%
10	223,320	2.71%
11	91,978	1.11%
12	69,778	0.85%
13	[illegible]	0.70%
14	46,822	0.57%
15	39,635	0.48%
16	32,168	0.39%
17	26,933	0.33%
18	23,131	0.28%
19	22,027	0.27%
20	23,953	0.29%

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: 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.

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.

Term frequency tf

- The term frequency $\mathrm{tf}_{t,d}$ of term *t* in document *d* is defined as the number of times that *t* occurs in *d*.
- Raw term frequency is not what we want:
- Relevance does not increase proportionally with term frequency.

Log-frequency weighting

- The log frequency weight of term t in d is

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

- 0 → 0, 1 → 1, 2 → 1.3, 10 → 2, 1000 → 4, etc.
- Score for a document-query pair: sum over terms *t* in both *q* and *d*:
- score $= \sum_{t \in q \cap d} (1 + \log \mathrm{tf}_{t,d})$

Document frequency

- 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., *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, continued

- 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 it's not a sure indicator of relevance.
- → For frequent terms, we want high positive weights for words like *high, increase, and line*
- But lower weights than for rare terms.
- We will use document frequency (df) to capture this.

idf weight

- df_t is the <u>document</u> frequency of t: the number of documents that contain t
  - df_t is an inverse measure of the informativeness of t
  - $\mathrm{df}_t \leq N$ (total number of docs)
- 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.

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

$$\mathrm{idf}_t = \log_{10} \left(N/\mathrm{df}_t \right)$$

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.

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
insurance	10440	3997
try	10422	8760

Quiz: Collection Frequency

Word	Collection frequency	Document frequency
insurance	10440	3997
try	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.

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

- 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

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 → 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 $\in R^{|V|}$

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

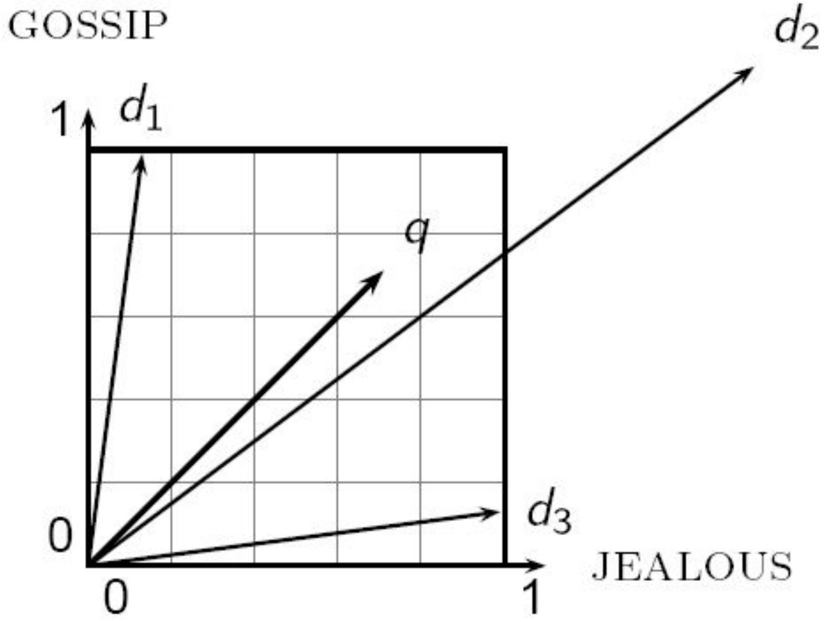
- Key idea 1: Do the same for queries: represent them as vectors in the space
- Key idea 2: Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors
- proximity ≈ inverse of distance

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 $\vec{q}$ and $\vec{d_2}$ 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.



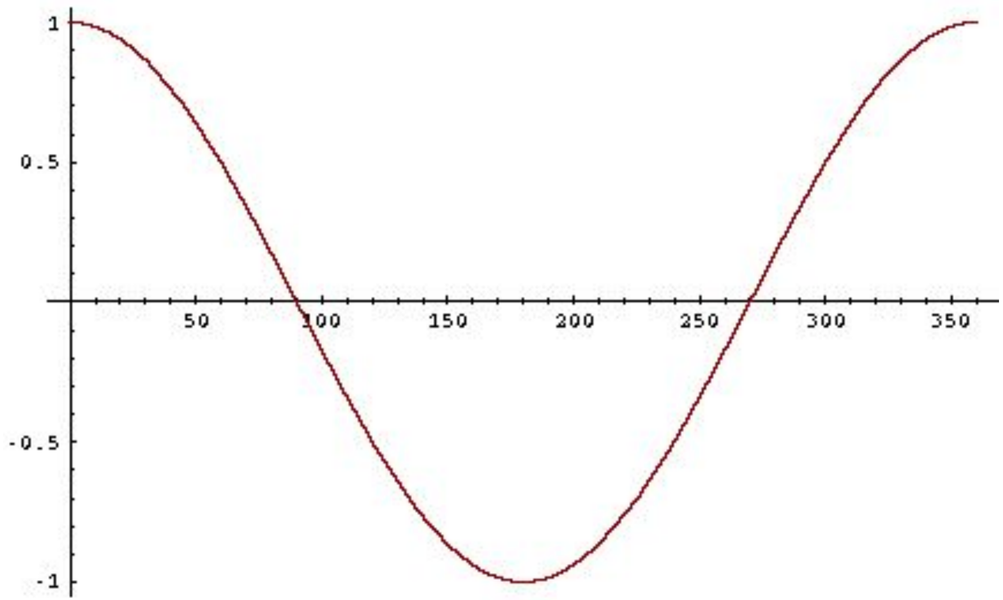
Use angle instead of distance

- 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 <u>decreasing</u> order of the angle between query and document
  - Rank documents in <u>increasing</u> order of cosine(query,document)
- Cosine is a monotonically decreasing function for the interval [0°, 180°]

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.

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 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 $\vec{d}$.

The law of cosines generalizes the Pythagorean theorem, which holds only for right triangles: if the angle γ is a right angle (of measure 90° or $\frac{\pi}{2}$ radians), then $\cos\gamma = 0$, and thus the law of cosines reduces to the Pythagorean theorem:

$$c^2 = a^2 + b^2.$$

The law of cosines is useful for computing the third side of a triangle when two sides and their enclosed angle are known, and in computing the angles of a triangle if all three sides are known.

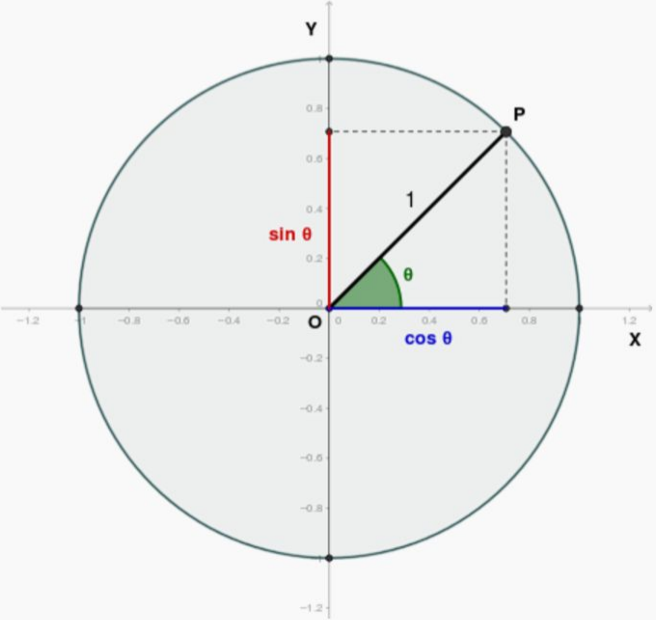
By changing which sides of the triangle play the roles of a, b, and c in the original formula, the following two formulas also state the law of cosines:

$$a^2 = b^2 + c^2 - 2bc\cos\alpha$$
$$b^2 = a^2 + c^2 - 2ac\cos\beta.$$

Though the notion of the cosine was not yet

sides a, b, and c.

Trigonometry



Outline · History · Usage

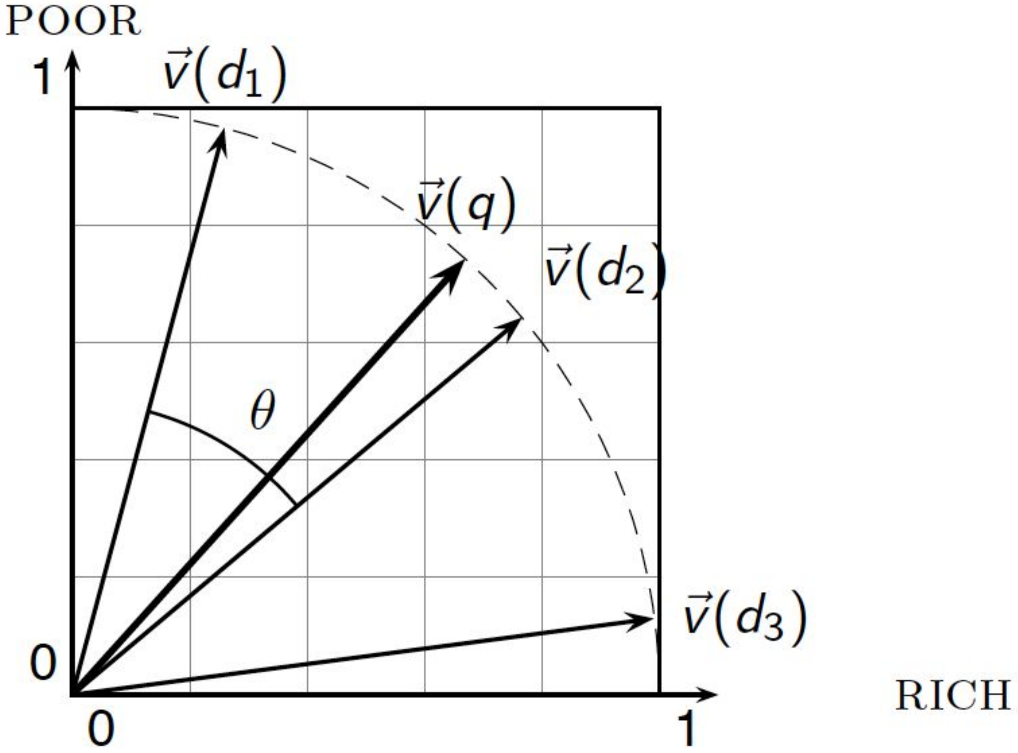
Cosine for length-normalized vectors

- 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: *Sense and Sensibility*
- PaP: *Pride and Prejudice*
- 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

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

$0.789 \times 0.832 + 0.515 \times 0.555 + 0.335 \times 0.0 + 0.0 \times 0.0 \approx 0.94$

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

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

Computing cosine scores

CosineScore(q)

float $Scores[N] = 0$
float $Length[N]$
for each query term t
do calculate $w_{t,q}$ and fetch postings list for t
    for each pair$(d, \mathrm{tf}_{t,d})$ in postings list
    do $Scores[d] += w_{t,d} \times w_{t,q}$
Read the array $Length$
for each d
do $Scores[d] = Scores[d]/Length[d]$
return Top K components of $Scores[]$

tf-idf 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)	$1/\mathit{CharLength}^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$				

Columns headed 'n' are acronyms for weight schemes.

Why is the base of the log in idf immaterial?

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