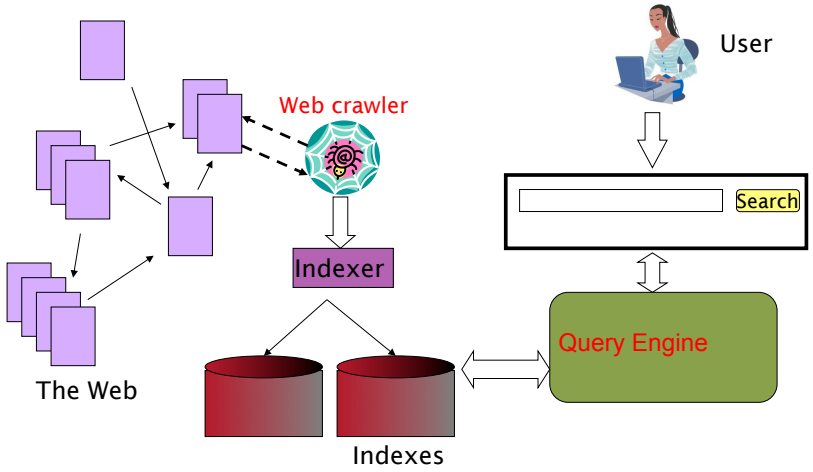
An Application of Vector Space: Ranking Web Search Results

The Web Search Model



Query Engine's operations

- Process query
- Look-up the index
- Retrieve list of documents
- **Order** documents
- Prepare results page

MOTIVATING QUESTION:
Given a large list of documents that match a query, how to **order** them according to their relevance?

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 $\in \{0,1\}^{|V|}$

Why Ranking Search Results?

- Thus far, our queries have all been Boolean.
  - Documents either match or don't. No "top-10" results.
- Boolean queries exhibit the "feast-or-famine" problem:
  - Adding a term to a query goes from '000's of results to no results
- Boolean queries are good for expert users with precise understanding of their needs and the collection.
  - Also good for applications: they can easily handle 1000s of results.
- Boolean are 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.

Solution: Score Documents to Rank

Given document d_i ; Given query q

- Calculate $score(q, d_i)$, usually a number [0, 1]
  - *score* measures how well document and query "match"
- Rank documents in decreasing order of $score(q, d_i)$

Assign 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: what the score of the document should be? ____
- If the query term occurs 5 times in d1 and 50 times in d2, which document should have a higher score? ____
- We will look at a number of alternatives for this.

But first: The Bag of Words Model

- "Bag of words" Model:
  - Document = bag of [unordered] words
  - *d1 = "John is quicker than Mary"* and *d2 = "Mary is quicker than John"* have the same words.
  - Is this a good idea?
- In a sense, this is a step back: The positional index was able to distinguish these two. But that's ok for now.

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

	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

Idea: Query terms that appear more often in a document, match it better

Method 1: Term Frequency $tf_{t,d}$

- Assign to each term a weight
  $tf_{t,d}$ **term frequency** = how often term *t* occurs in document *d*

$$score(q,d) = \sum_{t \in q} tf_{t,d}$$

- query = 'who wrote wild boys'
- doc1 = 'Duran Duran sang Wild Boys in 1984'
- doc2 = 'Wild boys don't remain forever wild'
- doc3 = 'Who brought wild flowers?'
- doc4 = 'It was John Krakauer who wrote *In to the wild.*'

query = {boys: 1, who: 1, wild: 1, wrote: 1}
doc1 = {1984: 1, boys: 1, duran: 2, in: 1, sang: 1, wild: 1}
doc2 = {boys: 1, don't: 1, forever: 1, remain: 1, wild: 2}
doc3 = { }
doc4 = { }

score(q, doc1) = 1 + 1 = 2
score(q, doc2) = 1 + 2 = 3
score(q, doc3) =
score(q, doc4) =

Why using just $tf_{t,d}$ is not good?

- Bigger documents have more terms, thus their score is larger.
  - Will need to fix this: *"length normalization"*
- A document having 10 occurrences of a term is not 10 times better than another that has it once.
  - Will need to fix this
- All terms are equally important.
  - How frequency can indicate importance?

Postulate:
A word that appears in (almost) every document, is not very important
i.e., it has no discriminatory power.

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 $= \sum_{t \in q \cap d} (1 + \log \mathrm{tf}_{t,d})$
- The score is 0 if none of the query terms is present in the document.

Method 2: Weights according to rarity

- 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*.
- df_t - document frequency for term t (= num of docs contain t)
- idf_t - inverse document frequency for term t

$$idf_t = \log \frac{N}{df_t}$$

N - total number of documents

idf example

suppose $N = 1{,}000{,}000$ documents

term	df_t	idf_t
arachnocentric	1	
capricious	100	
sunday	1,000	
person	10,000	
under	100,000	
the	1,000,000	

$$\text{idf}_t = \log_{10}(N/\text{df}_t)$$

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

Effect of idf on ranking

- Does *idf* have an effect on ranking of documents for one-term queries?
  - e.g. person
-
- Does *idf* affects the ranking for queries with > 1 terms
  - e.g. capricious person
  - $idf_{capricious} > idf_{person}$
-

Method 3: Better tf.idf weighting

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

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

- Best known weighting scheme in information retrieval
  - Alternative names: tf x idf, tf-idf (with hyphen, not minus)
- What happens to tf.idf weight when…
  - the **number of occurrences** within a document increases?
  - the **rarity of the term** in the collection increases?

Binary → count → tf.idf weight matrix

Term-Document 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 $\in \mathbb{R}^{|V|}$ of |V| terms

How to rank results for query: "***In the mercy of Caesar***"

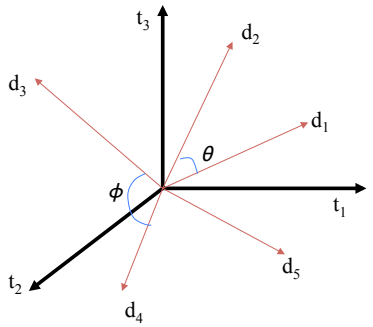
Ranking of documents for a query

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

- Rank by summing tf.idf weights.
- But how do we normalize lengths?
  We need to think in terms of vector arithmetic…

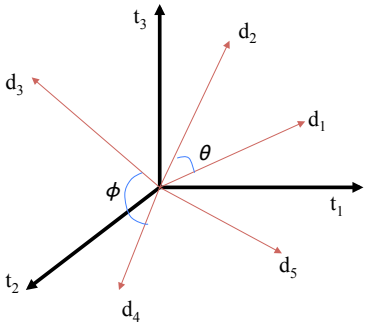
DOCUMENTS are vectors

- 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.



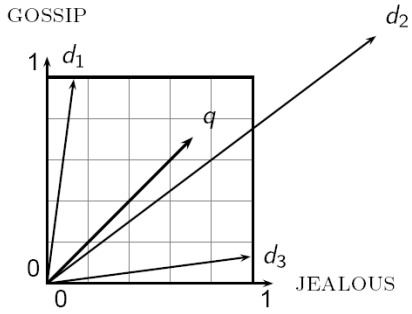
QUERIES are also vectors

- Key idea 1: Represent queries as vectors in the space
- Key idea 2: Rank documents according to their proximity to query in this space
  - proximity = similarity of vectors
  - proximity ≈ inverse of "distant"
- How to determine vector proximity?
  - Let's try shortest distance b/w vectors
  - Euclidean distance?



Euclidean distance is not a good 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 d_2 are very similar.

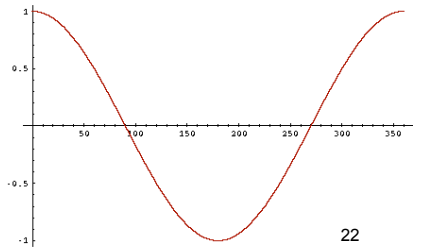


Use angle instead of distance

- Thought experiment:
  take a document *d* and append it to itself.
  Call this document d^2.
  - "Semantically" d and d^2 have the same content
- The Euclidean distance between the two documents can be quite large
- The angle between d and d^2 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°, 180°]



Length normalization

- A vector can be (length-) normalized
  by dividing each of its components by its length –
  for this we will divide by 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^2:
  they have identical vectors after length-normalization.
  - Long and short documents now have comparable weights

cosine(query,document)

Dot product | Unit vectors

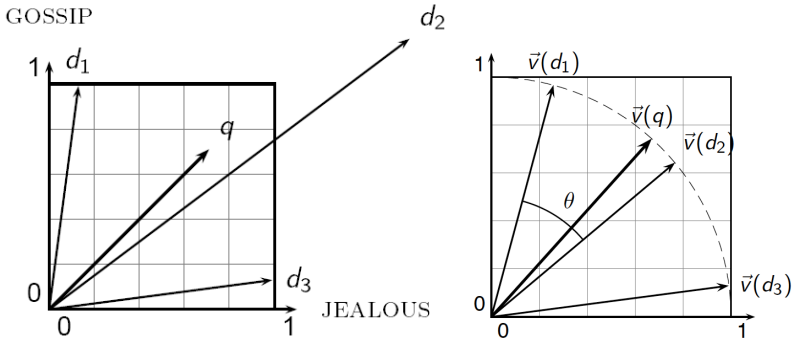
$$\cos(\vec{q},\vec{d}) = \frac{\vec{q}\bullet\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\vec{q}}{|\vec{q}|}\bullet\frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

q_i is the 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 similarity illustrated

- The effect of length normalization



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(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

Calculation of similarities

TERM-DOCUMENT MATRIX				
term vectors	S a S	P a P	Wh Hei	query = jealous gossip
affection	115	58	20	0
jealous	10	7	11	1
gossip	2	0	6	1
EUCLIDIAN NORMS				
euclidian nor	115.451288	58.4208867	23.6008474	1.41421356
NORMALIZED TERM-DOCUMENT MATRIX				
norm term vec	S a S	P a P	Wh Hei	
affection	0.99609109	0.99279561	0.8474272	0
jealous	0.08661662	0.11982016	0.46608496	0.70710678
gossip	0.01732332	0	0.25422816	0.70710678
SIMILARITIES CALCULATION			SIMILARITIES TO QUERY	
sim(SAS, PaP)	0.99929328		sim(q,SaS)	0.07349664	
sim(SaS, WH)	0.88888946		sim(q,PaP)	0.08472565	
sim(PaP,WH)	0.89716838		sim(q,WH)	0.50933829	best match
SaS is much more similar to PaP				
than it is to WH.				

Alternative 1: Jaccard coefficient

- A commonly used measure of overlap of two sets *A* and *B*
  - jaccard*(A,B)* = $|A \cap B| / |A \cup B|$
  - jaccard*(A,A)* = ___
  - jaccard*(A,B)* = ___ if $A \cap B = 0$
- Jaccard() takes values between ___ and ___.
- Example: Query q: *ides of march*
  - What is the score that the Jaccard coefficient computes for each of the two documents below?
  - d1: *caesar died in march*
  - d2: *the long march*
- Score(q,d1) = ____ Score(q,d2) = ____ More relevant: ___

Issues with Jaccard for scoring

- + Easy to implement
- - It doesn't consider how many times a term occurs in a document
  - *Ignores "term frequency"*
- - It considers every word to be equally important. But rare terms in a collection are more informative than frequent terms.
  - *Ignores "term rarity"*
- - Long documents have an advantage. We need a sophisticated way of normalizing for length
  - *Ignores "length normalization"*