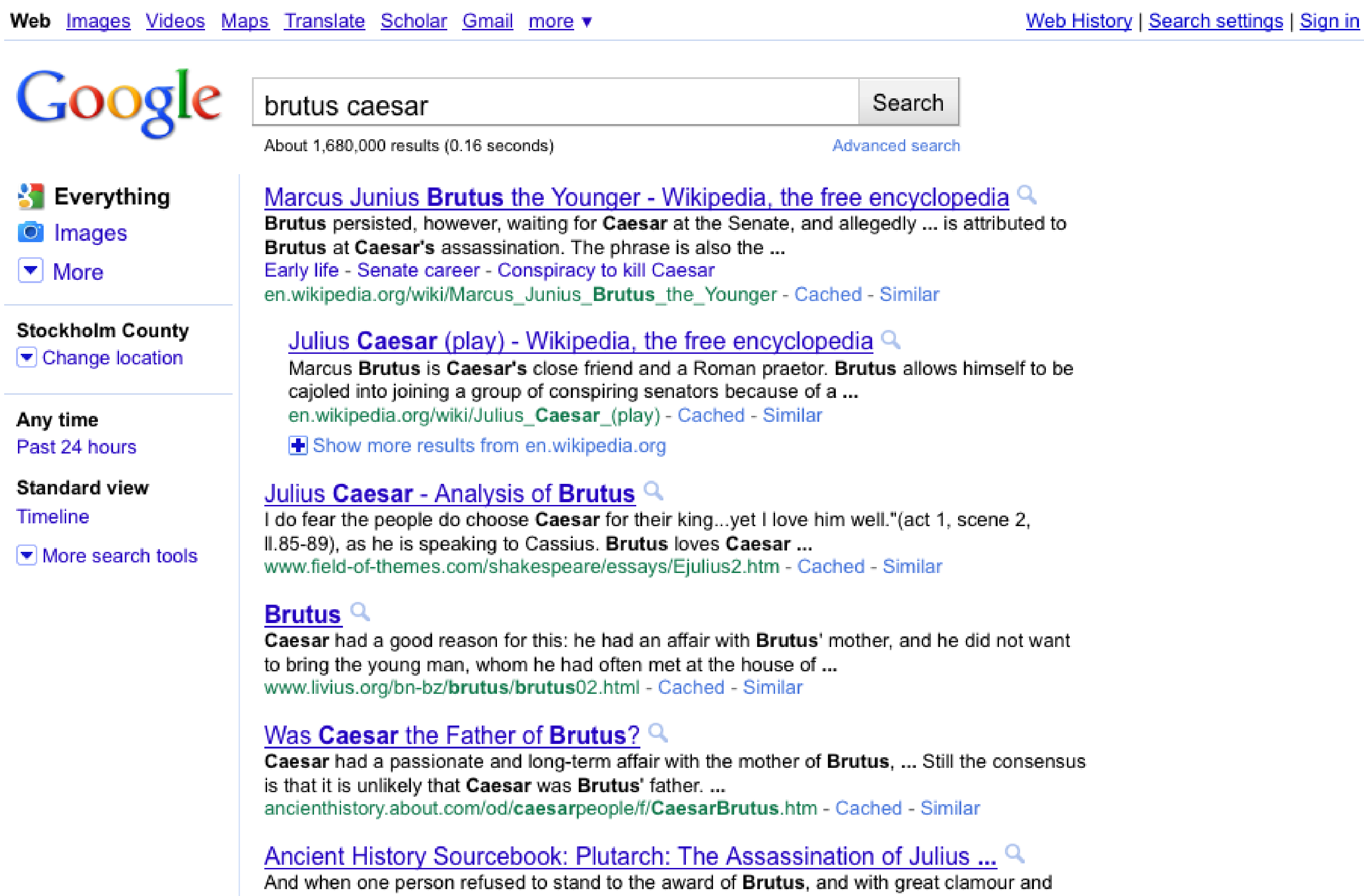
DD2476: Lecture 4

- So far: **Boolean retrieval**
  - In computer assignment 1 you have implemented a special case of Boolean retrieval(intersection).
  - Boolean retrieval is good for **expert users** (e.g. users of *Westlaw*).
  - But it is bad for most users, especially for web search.

Problems with Boolean search

- Boolean queries often return **too many** or **too few** results
  - ”zyxel P-660h” → 192 000 results
  - ”zyxel P-660h” ”no card found” → 0 results
- Takes skill to formulate a search query that gives a manageable number of hits.
  - ”AND” gives too few, ”OR” too many

Ranked retrieval



Ranked retrieval

- Every matching document is given a score, say in [0..1]
- The **higher** the score, the **better** the match
- Large result sets do not pose problems
  - Show top *k* results (k≈10)
  - Option to see more.
  - **Premise**: The ranking algorithm works!

Today's topics

- The **Vector Space model**
  - translates the matching problem into a geometric problem
- ***tf-idf***-weighting
  - takes **frequency** of search terms into account
- Vector Space + *tf-idf* → model where documents are ranked according to the **similarity to the query**

Term-document incidence matrix

	Antony & Cleopatra	Julius Caesar	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	0	1	1	1
citizen	1	1	0	0	1	0

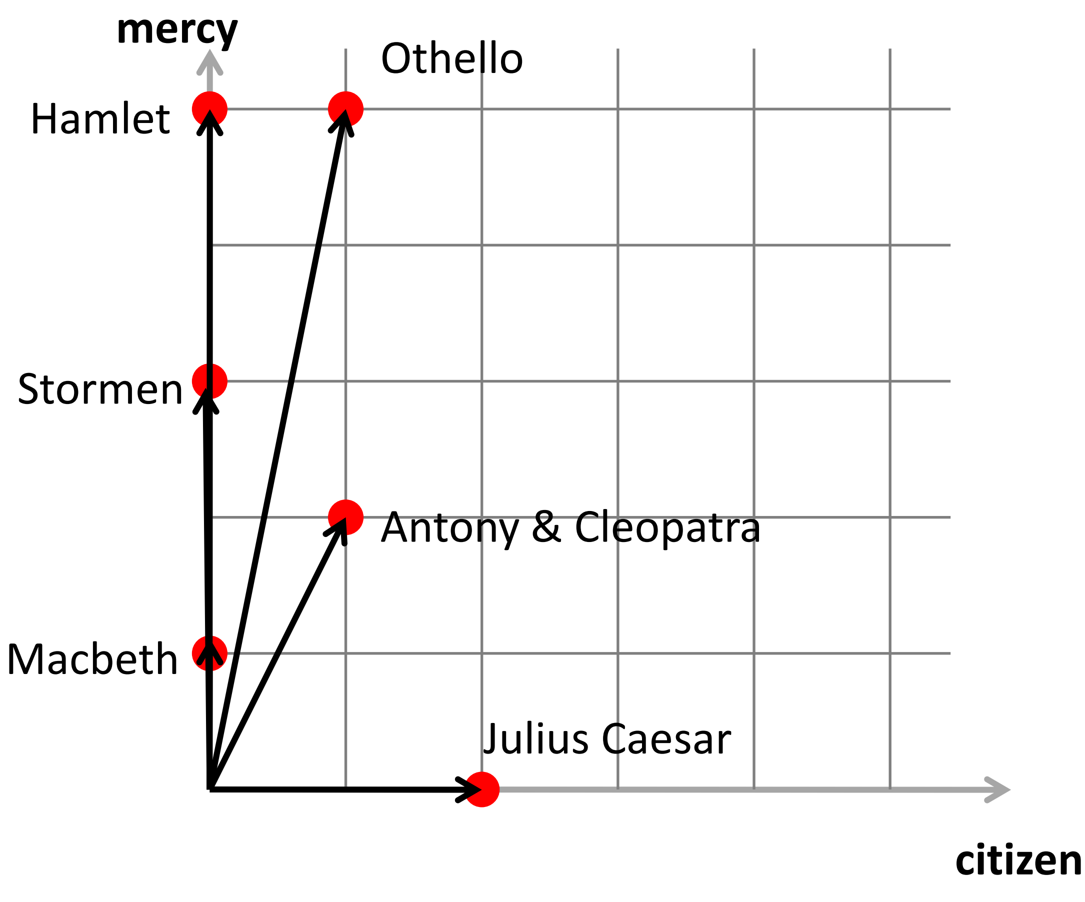
1 if term is present in document, 0 otherwise

Word count matrix

	Antony & Cleopatra	Julius Caesar	Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	1
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
citizen	1	2	0	0	1	0

Every document is a vector in term space.

Documents as vectors



Bag-of-words model

- Don’t consider ordering of words
  - ”Carl is wiser than Mary” and ”Mary is wiser than Carl” has the **same** vector
- In a sense, step back:
  - The positional index (assignment 1.3) could distinguish between these two documents.

Term frequency *tf*

	Antony and Cleopatra
ANTONY	$\boxed{157}$

- $\boldsymbol{tf_{t,d}}$ = number of times term *t* occurs in document *d*
- How can we use *tf* for query-document matching scores?
- 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

log-frequency weighting

- **Log-frequency weight** of term *t* in document *d*

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

Example

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

term	$\mathrm{tf}_{t,d}$	$w_{t,d}$
airplane	0	
shakespeare	1	
calpurnia	10	
under	100	
the	1,000	

Simple query-document score

- Score for a query-document pair: sum over terms *t* in both *q* and *d*:

$$\text{Score} = \sum_{t \in q \cap d} (1 + \log_{10} \mathrm{tf}_{t,d})$$

- Score is 0 if no query term is present in the document.

Document frequency *df*

- Rare terms are more informative than frequent terms
- Example: rare word ARACHNOCENTRIC
  - Document containing this term is very likely to be relevant to query ARACHNOCENTRIC
    → High weight for rare terms like ARACHNOCENTRIC
- Example: common word THE
  - Document containing this term can be about anything
    → Very low weight for common terms like THE
- We will use **document frequency** (df) to capture this.

idf (inverse ***df***)

- Informativeness ***idf*** (inverse document frequency) of *t*:

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

where N is the number of documents.

log (N/df_t) instead of N/df_t to “dampen” the effect of idf.

Example

Suppose N = 1,000,000 $\mathrm{idf}_t = \log_{10}(N/\mathrm{df}_t)$

term	df_t	idf_t
calpurnia	1	
animal	100	
sunday	1,000	
fly	10,000	
under	100,000	
the	1,000,000	

Effect of idf on ranking

- Note that idf has no effect on ranking for one-term queries, like ‘CAPRICIOUS’.

- Only effect for >1 term
  - Query CAPRICIOUS PERSON: idf puts more weight on CAPRICIOUS than PERSON.

tf-idf weighting

- **tf-idf weight** of a term: product of tf weight and idf weight

- 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**

Weight matrix

	Antonius och Cleopatra	**Julius Caesar**	**Tempest**	**Hamlet**	**Othello**	**Macbeth**
Antonius			0	0	0	
Brutus			0		0	0
Caesar			0			
Calpurnia	0		0	0	0	0
Cleopatra		0	0	0	0	0
nåd		0				
medborgare			0	0		0

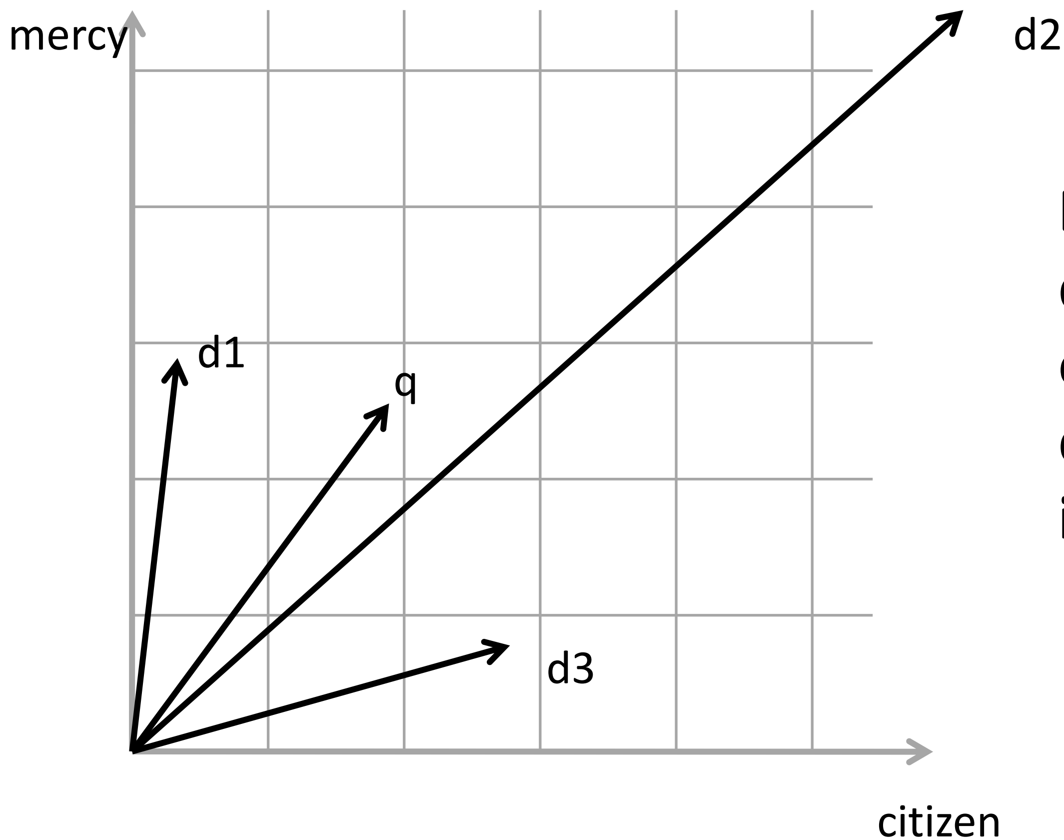
Documents as vectors

- So we have a |V|-dimensional vector space
  - Terms are axes/dimensions
  - Documents are points in this space
- Very high-dimensional
  - ~3.2×10^6 dimensions for our Wikipedia corpus, much more for entire web
- Very sparse vectors - most entries zero

Queries as vectors

- **Key idea 1:** Represent queries as vectors in same space
- **Key idea 2:** Rank documents according to proximity to query in this space
  - proximity = similarity of vectors
  - proximity ≈ inverse of distance
- Recall:
  - Get away from Boolean model
  - Rank more relevant documents higher than less relevant documents

Euclidean distance: bad idea



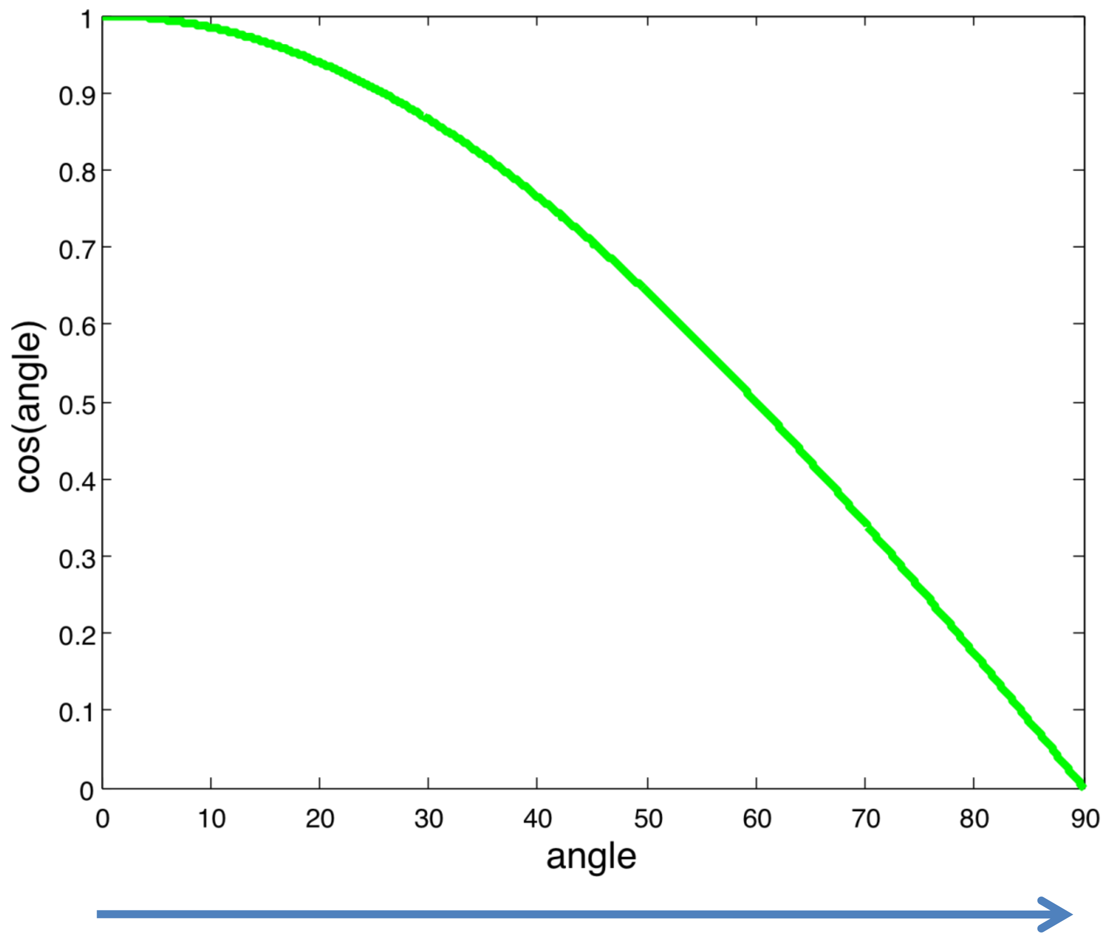
Distance between q and d2 is big, even though the distribution of terms in q and d2 are similar.

Angles 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:
  - Length unimportant
  - Rank documents according to **angle from query**

cos(angle) better than angle



Monotonically decreasing

Cosine similarity

- **Scalar product** of *u* and *v*:

$$u \cdot v = \sum_{i=0}^{n} u_i v_i$$

- It holds that:

$$u \cdot v = |u||v| \cos\theta$$

where |*u*| = the length of *u*, and θ the angle between *u* och *v*

- Therefore:

$$\cos\theta = \frac{\sum_{i=0}^{n} u_i v_i}{|u||v|}$$

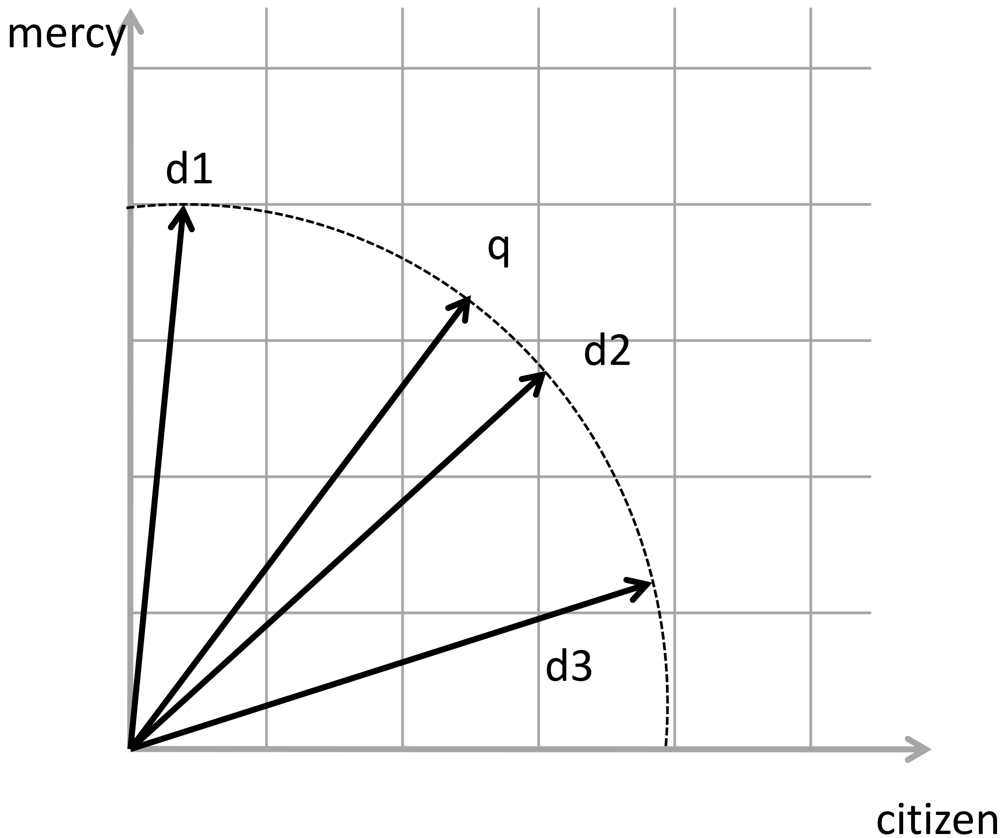
Length normalisation

- **Divide each component** in the vector *v* with the length of *v:*

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

- Every (document) vector then has unit length (1), with endpoint on the **unit hypersphere**.
- Note: Documents *d* are *dd* are the same after normalisation.

Cosine similarity



Cosine similarity

Scalar product Unit vectors

$$\cos(q,d) = \frac{q \cdot d}{|q||d|} = \frac{q}{|q|} \cdot \frac{d}{|d|} = \frac{\sum_{i=0}^{n} q_i d_i}{\sqrt{\sum_{i=0}^{n} q_i^2}\sqrt{\sum_{i=0}^{n} d_i^2}}$$

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(q,d) = is the **cosine similarity** of *q* and *d*

= the cosine of the angle between *q* and *d*.

Cosine similarity - Example

- How similar are the following novels?
  - SaS: Sense and Sensibility
  - PaP: Pride and Prejudice
  - WH: Wuthering Heights?

- Term frequency tf_t

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

Cosine similarity - Example

- Log frequency weights: $w_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{tf}_{t,d}, & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$
- Term frequency tf_t

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

Cosine similarity - Example

- After length normalisation:

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 ≈ 0.94

cos(SaS,WH) ≈ 0.79

cos(PaP,WH) ≈ 0.69

Summary – Vector Space ranking

- Vector space ranking:
  - Represent the query as a tf-idf vector
  - Represent each document as a 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

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 $Scores[]$

Efficient cosine ranking

- Find the *K* docs in the collection "nearest" to the query $\Rightarrow$ *K* largest query-document cosine scores

- Efficient cosine ranking:
  - **Computing each cosine score efficiently**
  - **Choosing the K largest scores efficiently**

Computing cosine scores efficiently

- Approximation:
  - Assume that terms only occur once in query document

$$w_{t,q} \leftarrow \begin{cases} 1, & \text{if } \mathrm{w}_{t,q} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- Works for short documents ($|d| << N$)
- Works since ranking only relative

Computing cosine scores efficiently

FastCosineScore(q)

float $Scores[N] = 0$
for each d
do Initialize $Length[d]$ to the length of doc d
for each query term t
do calculate $w_{t,q}$ and fetch postings list for t
    for each pair(d, $tf_{t,d}$) in postings list
    do add $wf_{t,d}$ to $Scores[d]$
Read the array $Length[d]$
for each d
do Divide $Scores[d]$ by $Length[d]$
return Top K components of $Scores[]$

Figure 7.1 A faster algorithm for vector space scores.

Computing cosine scores efficiently

- Downside of approximation: **sometimes get it wrong**
  - A document not in the top *K* may creep into the list of *K* output documents
- Is this such a bad thing?
- Cosine similarity is only a proxy
  - User has a task and a query formulation
  - Cosine matches documents to query
  - Thus cosine is anyway a proxy for user happiness
  - If we get a list of *K* documents "close" to the top *K* by cosine measure, should be ok

Choosing K largest scores efficiently

- Retrieve top *K* documents wrt query
  - Not totally order all documents in collection

- Do selection:
  - avoid visiting all documents

- Already do selection:
  - Sparse term-document incidence matrix, $|d| \ll N$
  - Many cosine scores = 0
  - Only visits documents with nonzero cosine scores (≥1 term in common with query)

Generic approach

- Find a set A of contenders, with $K < |A| \ll N$
  - A does not necessarily contain the top K, but has many docs from among the top K
  - Return the top K documents in A
- Think of A as pruning non-contenders
- Same approach used for any scoring function!
- Will look at several schemes following this approach

Index elimination

- Example:

  CATCHER IN THE RYE

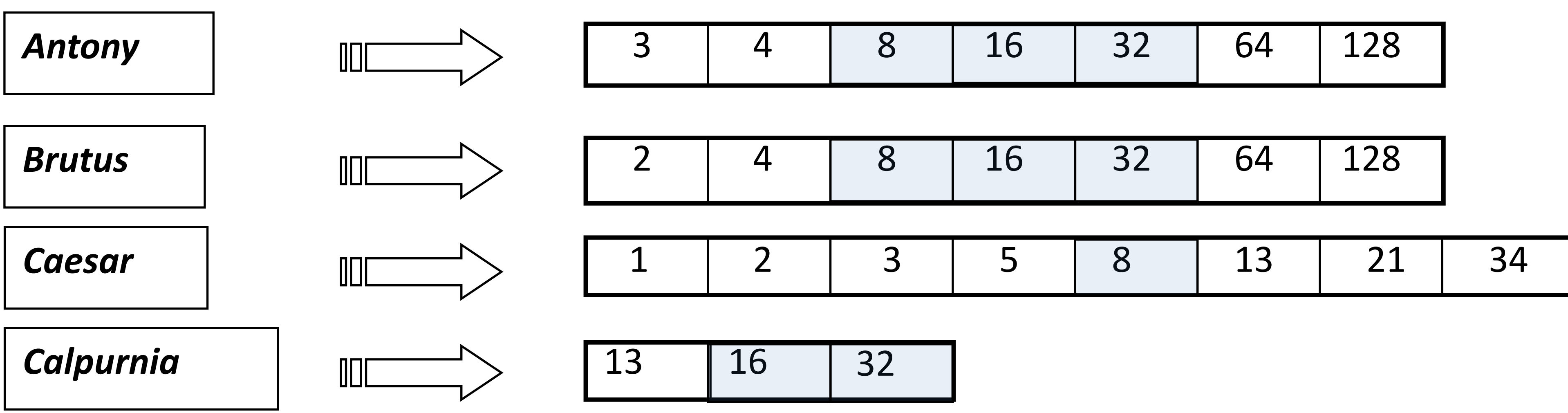
- **Only accumulate scores from CATCHER and RYE**
- Intuition:
  - IN and THE contribute little to the scores – do not alter rank-ordering much
  - Compare to stop words
- Benefit:
  - Posting lists of low-idf terms have many documents $\rightarrow$ eliminated from set A of contenders

Index elimination

- Example:

  CAESAR ANTONY CALPURNIA BRUTUS

- **Only compute scores for documents containing ≥3 query terms**



Champions lists

- Precompute for each dictionary term *t*, the *r* documents of highest tf-idf$_{td}$ weight
  - Call this the champion list (fancy list, top docs) for *t*
- Benefit:
  - At query time, only compute scores for documents in the champion lists – fast
- Issue:
  - *r* chosen at index build time
  - Too large: slow
  - Too small: too few results

Static quality scores

- We want top-ranking documents to be both **relevant** and **authoritative**
  - Relevance – cosine scores
  - Authority – query-independent property

- Examples of authority signals
  - Wikipedia among websites (qualitative)
  - Articles in certain newspapers (qualitative)
  - A paper with many citations (quantitative)
  - PageRank (quantitative)

Static quality scores

- Assign **query-independent quality score** g(*d*) in [0,1] to each document *d*

- net-score(q,d) = g(d) + cos(q,d)
  - Two "signals" of user happiness
  - Other combination than equal weighting
- Seek top *K* documents by net score
  - Can combine champion lists with *g(d)*-ordering
  - Maintain for each term *t* a champion list of the *r* documents with highest $g(d)$ + tf-idf$_{td}$
  - Seek top *K* results from only the documents in these champion lists
- More on this next week.

Query parser

- Query phrase:

  RISING INTEREST RATES

- Sequence:
  - Run as a **phrase query**
  - If <*K* documents contain the phrase RISING INTEREST RATES, run **phrase queries** RISING INTEREST and INTEREST RATES
  - If still <K docs, run **vector space query** RISING INTEREST RATES
  - Rank matching docs by **vector space scoring**

Next

- **Next week:** Link analysis and PageRank