CS3245

Information Retrieval

7

Lecture 7: Scoring, Term Weighting and the Vector Space Model

Live Q&A
https://pollev.com/jin

Last Time: Index Compression

- Collection and vocabulary statistics: Heaps' and Zipf's laws
- Dictionary compression for Boolean indexes
  - Dictionary string, blocks, front coding
- Postings compression:
  - Gap encoding and variable byte encoding

Data structure	Size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocking, k = 4	7.1
with blocking & front coding	**5.9**
postings, uncompressed (32-bit words)	400.0
postings, variable byte encoded	**116.0**

Today: Ranked Retrieval

- Scoring documents
- Term frequency
- Collection statistics
- Weighting schemes
- Vector space scoring

- Parametric and zone indexes (Section 6.1) will be covered next week.

Problem with Boolean search: Difficulty in query formulation

- Boolean queries
  - Terms + Boolean operators

- Most (non-expert) users are likely to have difficulty in writing Boolean queries.
  - What are the correct terms to use?
  - What do the operators mean and how to use them?

Problem with Boolean search: Feast or Famine with no differentiation

- Boolean logic is quite strict
- They can result in either too few (=0) or too many (1000s) results.
  - Q1: "Windows 10" AND login AND KB3081444 → 0 hits
  - Q2: "Windows 10" OR login OR KB3081444 → 377M hits
    - Also called "information overload"
- All the returned results are considered equally good by the search engine…

Problem with Boolean search: Feast or Famine with no differentiation

- **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 don’t want to wade through 1000s of results.

Ranked retrieval

- **Free text queries**: The user's query is just one or more words in a human language.
- **Ranked results**: The results are ranked in the order of estimated relevance.
- Two separate choices, but a common combination.

Ranked retrieval

- All the users need to do is:
  - Write a free-text query and check the top k (≈ 10) results
    - If the results are good, the search is done.
    - Otherwise, repeat this process with a reformulated query.
- Simple and cost-effective, however…
  - The ranking algorithm must work (i.e., most relevant documents should be ranked as the top results.)

Scoring as the basis of ranked retrieval

How to rank the documents in the collection with respect to a query?

- Assign a score to each document
  - A number in [0, 1] which measures how well the query and the document match.
- Sort the documents based on the scores
  - Documents with score = 1
  - Documents with score = 0.99
  - …

Take 1: Jaccard coefficient

- From Chapter 3 (spelling correction)
- Measures the overlap of two sets A and B

  Jaccard $(A, B) = |A \cap B| / |A \cup B|$

  Jaccard $(A, A) = 1$

  Jaccard $(A, B) = 0$ if $A \cap B = 0$

- Let A = the set of terms in the query, B = the set of terms in a document
  - Jaccard provides an estimate of how well the query and the document match

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*
- Doc 1: *caesar died in march* Jaccard (Q, Doc 1) = 1/6
- Doc 2: *the long march* Jaccard (Q, Doc 2) = 1/5

- Results:
  - Doc 2
  - Doc 1

Information not considered in Jaccard

- **Term Frequency**
  - Query: Caesar
  - Doc A (A story about Caesar): Caesar … Caesar … Caesar …
  - Doc B (A list of dictators): Caesar … Hitler …
  - A > B since *Caesar* appears more often in A (i.e., of higher term frequency).

Recap: **Binary** term-document incidence matrix (from Week 2)

	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

1. Term frequency matrix

- Contains the frequency of a term in a document:

	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

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?

- **Relevance** does not increase proportionally with **raw term frequency**
  - A document with 10 occurrences of the term is more relevant than a document with 1 occurrence. But not 10 times more relevant.

Log-frequency weighting scheme

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

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

e.g. 0 → 0, 1 → 1, 2 → 1.3, 10 → 2, 1000 → 4, etc.

- Let say:

  Q = Antony Cleopatra Calpurnia

  D = the play Anthony and Cleopatra

  Score (D, Q) = $(1 + \log_{10}157)$ + $(1 + \log_{10}57) + 0$

	Antony and Cleopatra
Antony	157
Brutus	4
Caesar	232
Calpurnia	0
Cleopatra	57

Information not considered in Jaccard

- **Document Frequency**
  - Query: the emperor
  - Document A: emperor
  - Document B: the
  - A > B since *the* is too common (i.e., of higher document frequency) and hence less important than *emperor*

2. Document frequency

- **Rare terms are more informative** than frequent terms
  - Given a query: **the emperor**, it is more important to match "emperor" than to match "the".

- We want…
  - **Lower weights** for **more common words** like *the, increase,* and *line*, and
  - **Higher weights** for **rarer ones** like *emperor*, and *arachnocentric*.

- This can be captured by the **inverse document frequency** (idf) weighting scheme.

idf weighting scheme

- 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*
  - $df_t \leq N$ where *N* is the collection size.

- We define the idf (inverse document frequency) of *t* by

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

  - We use log (N/df_t) instead of $1/\mathrm{df}_t$ to keep the value non-negative and dampen the effect of 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

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

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

tf-idf weighting

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

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

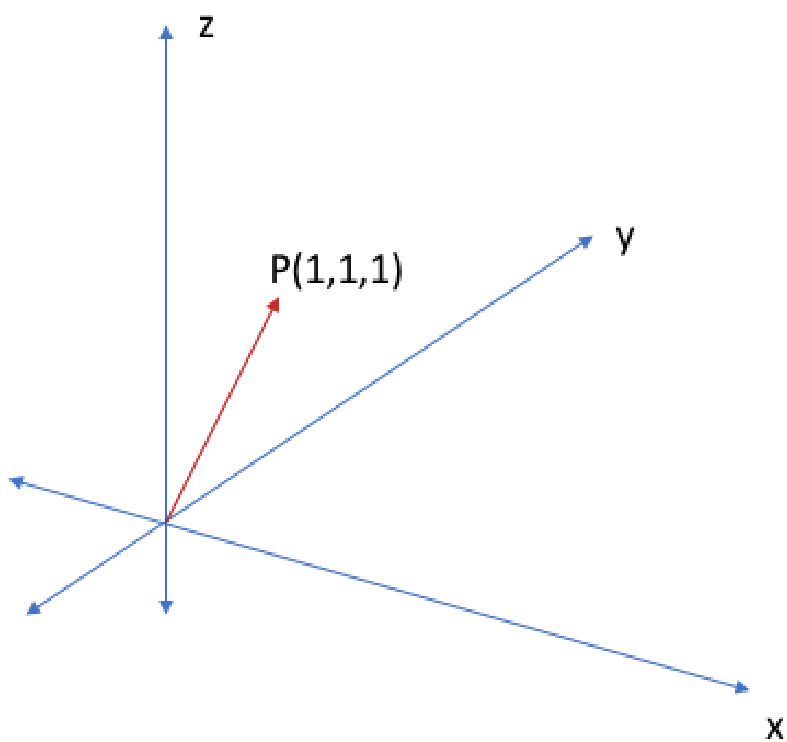
- **Best known weighting scheme IR**
  - Note: the "-" in tf-idf is a hyphen, not a minus sign!
  - Alternative names: tf.idf, tf x idf
- **Increases** with the number of occurrences within a document
- **Increases** with the rarity of the term in the collection

Final ranking of documents for a query

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

Vector and vector space

- A 3-dimensional vector space with a vector P = (1, 1, 1)



tf-idf 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 a vector in a vector space.

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
- High-dimensional: tens of thousands of dimensions; each dictionary term is a dimension
- 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; they are "mini-documents"
- Key idea 2: Rank documents according to their **proximity** to the query in this space

	Q: Antony mercy
Antony	2.45
Brutus	0
Caesar	0
Calpurnia	0
Cleopatra	0
mercy	1.21
worser	0

	Antony and Cleopatra	Julius Caesar
Antony	5.25	3.18
Brutus	1.21	6.1
Caesar	8.59	2.54
Calpurnia	0	1.54
Cleopatra	2.85	0
mercy	1.51	0
worser	1.37	0

Blanks on slides, you may want to fill in

Formalizing vector space proximity

- First cut: distance between two points
  - (= distance between the end points of the two vectors)
- Euclidean distance?

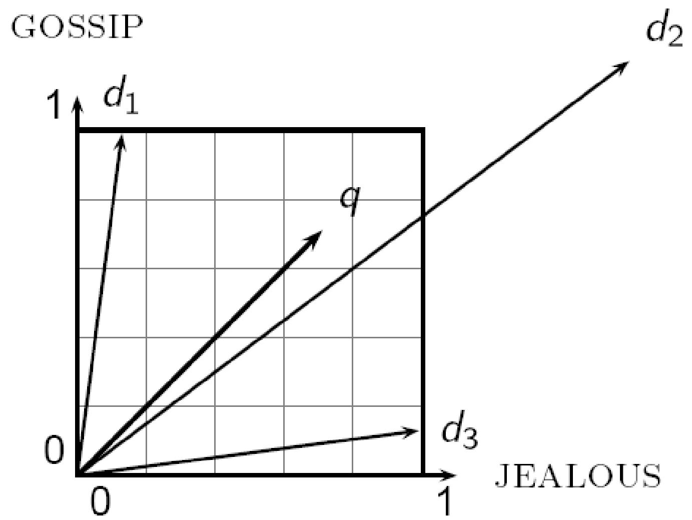
$$d(\mathbf{p},\mathbf{q}) = d(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2 + (q_2-p_2)^2 + \cdots + (q_n-p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}.$$

- Euclidean distance is a bad idea ...

Why distance is a bad idea

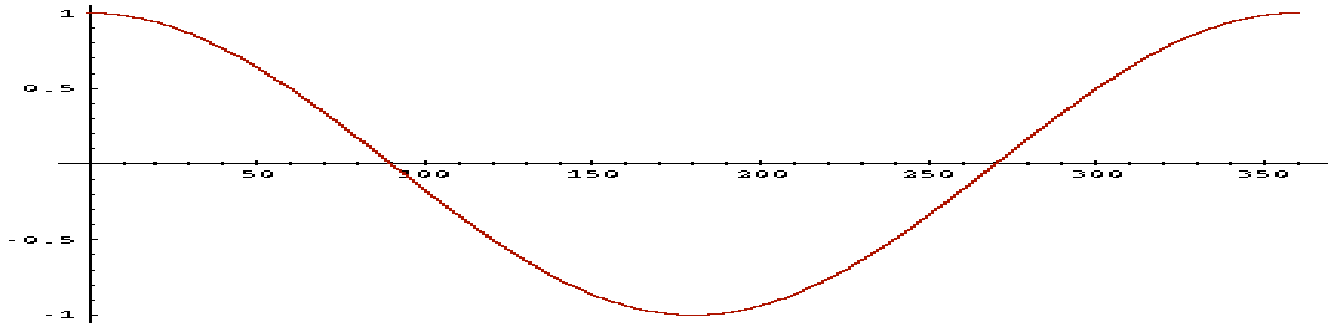
- The Euclidean distance between $\vec{q}$ and $\overrightarrow{d_2}$ is large even though the distribution of terms in the query $\vec{q}$ and the distribution of terms in the document $\overrightarrow{d_2}$ are very similar.



- Key idea: Rank documents according to **the angle** with query instead.

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°]



cosine (query, document)

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

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

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 $\vec{d}$.

Length normalization

- The vectors in the computation of cosine similarity are in fact *length normalized* by dividing each of its components by its length:

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

- Such normalization makes the weights comparable across different vectors despite their original lengths.
- Effect on the two documents $\vec{d}$ and $\overrightarrow{d'}$ (d appended to itself): they have identical vectors after length normalization.

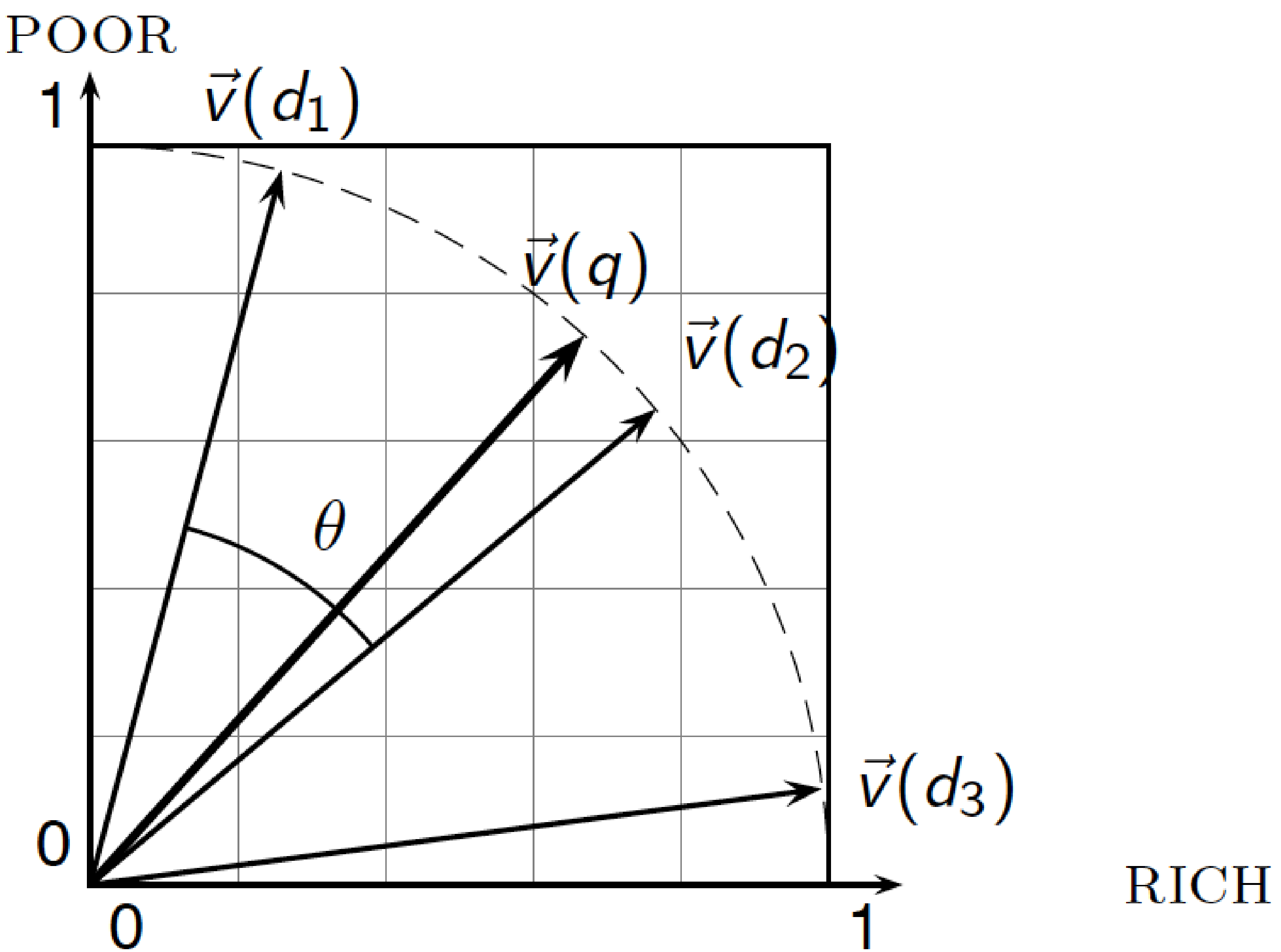
Cosine for length-normalized vectors

- 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 length normalized $\vec{q}$ and $\vec{d}$

Cosine similarity illustrated



Cosine similarity example

How similar are these documents vs the query:

affection jealous

term	Doc 1	Doc 2	Q
affection	115	58	1
jealous	10	7	1

Term frequencies

Note: To simplify this example, we do not do idf weighting and consider only two terms.

Cosine similarity example

Log frequency weighting

term	Doc 1	Doc 2	Q
affection	3.06	2.76	1
jealous	2.00	1.85	1

After length normalization

term	Doc 1	Doc 2	Q
affection	0.84	0.83	0.71
jealous	0.55	0.56	0.71

$\cos(\text{Doc } 1, Q) \approx 0.84 \times 0.71 + 0.55 \times 0.71 \approx 0.99$

$\cos(\text{Doc } 2, Q) \approx 0.99$

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, tf_{t,d}) in postings list
    do Scores[d] += w_{t,d} × w_{t,q}
Read the array Length
for each d
do Scores[d] = Scores[d]/Length[d]
return Top K components of Scores[]
```

This algorithm does not follow the formula exactly. What are the differences and why?

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

Weighting may differ in queries vs documents

- Many search engines allow for different weightings for queries vs. documents
- SMART Notation: denote combination used with the notation *ddd.qqq*, using the acronyms from the table on the previous slide
- A very standard weighting scheme is lnc.ltc
  - Document: logarithmic *tf* (l as first character), no idf, cosine normalization

    A bad idea?
  - Query: logarithmic *tf* (l in the leftmost column), idf (t in the second column) and cosine normalization

tf–idf example: lnc.ltc

Document: *car insurance auto insurance*
Query: *best car insurance*

Term	Document				Query						Prod
	tf-raw	tf-wt	wt	n'lize	tf-raw	tf-wt	df	idf	wt	n'lize	
auto	1	1	1	0.52	0	0	5000	2.3	0	0	0
best	0	0	0	0	1	1	50000	1.3	1.3	0.34	0
car	1	1	1	0.52	1	1	10000	2.0	2.0	0.52	0.27
insurance	2	1.3	1.3	0.68	1	1	1000	3.0	3.0	0.78	0.53

Quick Question: what is *N*, the number of docs?

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

Score = 0+0+0.27+0.53 = 0.8

Bag of words model

- Con: Vector representation doesn't consider the ordering of words in a document

  Moonlight bests La La Land at the Oscars and *La La Land bests Moonlight at the Oscars* have the same vectors

- 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.

Summary and algorithm: Vector space ranking

1. Represent the query as a weighted *tf-idf* vector
2. Represent each document as a weighted *tf-idf* vector
3. Compute the cosine similarity score for the query vector and each document vector
4. Rank documents with respect to the query by score
5. Return the top *K* (e.g., *K* = 10) to the user

Resources for today's lecture

- IIR 6.2 – 6.4.3