CSEN1076: NATURAL LANGUAGE PROCESSING AND INFORMATION RETRIEVAL

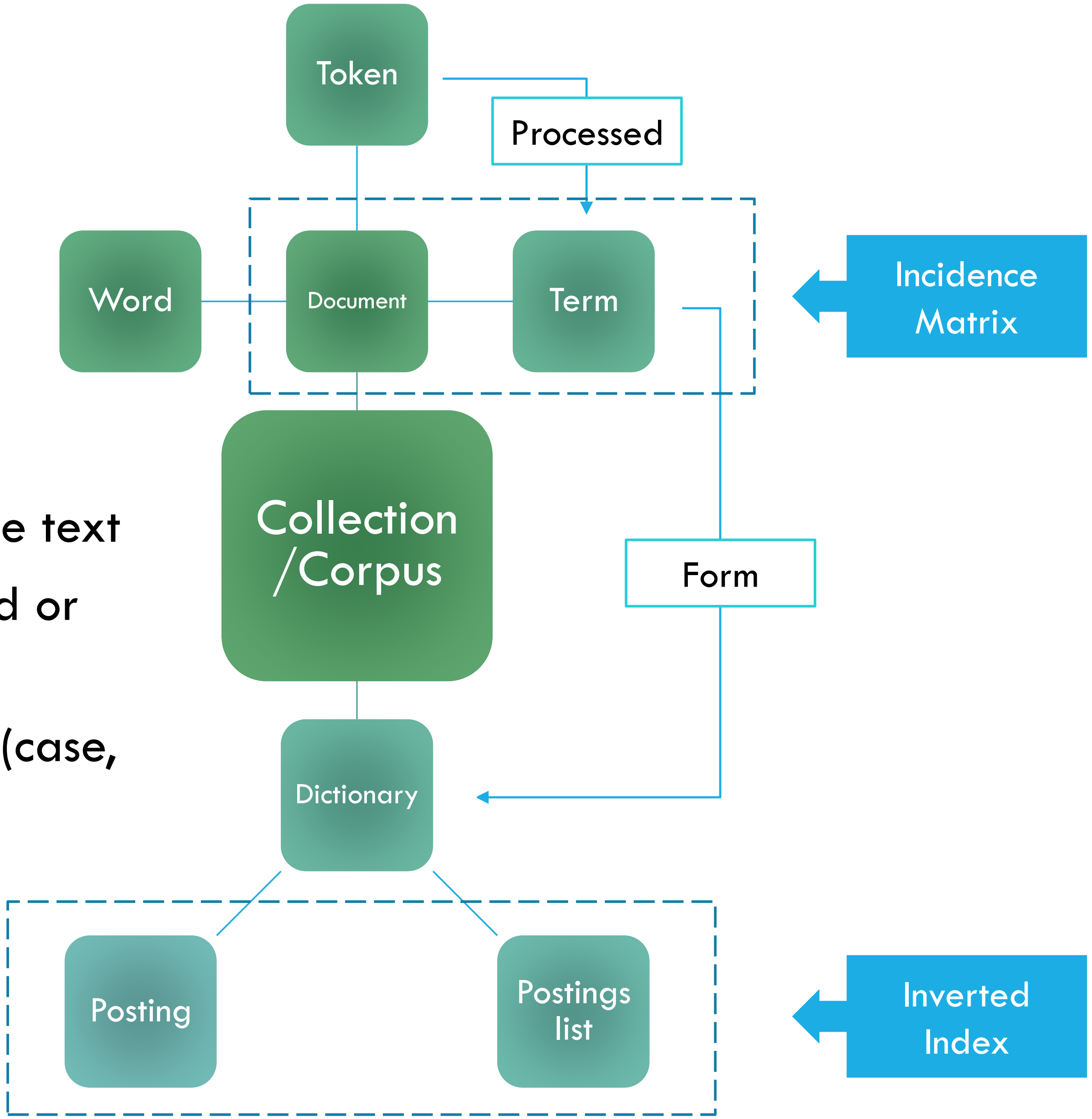
LECTURE 2 –INFORMATION RETRIEVAL II
PHRASE QUERIES, RANKED RETRIEVAL, AND TERM WEIGHTING

MERVAT ABUELKHEIR

1^{ST} NLP QUIZ IS <u>24/2/2020</u> <u>4:00-4:20 PM</u> AT H16 & H17 NO COMPENSATIONS

RECAP

- **Word** – A delimited string of characters as it appears in the text
- **Token** – An instance of a word or term occurring in a document
- **Term** – A “normalized” word (case, morphology, spelling etc.)



IR BASICS – PHRASE QUERIES

PHRASE QUERIES

We want to answer a query such as [**Stanford university**] – as a phrase

Thus [**The inventor Stanford Ovshinsky never went to university**] should not be a match

The concept of **phrase query** has proven easily understood by users

- About 10% of web queries are phrase queries
- Many more are implicit phrase queries (e.g. person names)

Consequence for the inverted index:

- **it no longer suffices to store only <*term* : *docs*> entries**

A FIRST ATTEMPT: BIWORD (BIGRAM) INDEXES

Index every consecutive pair of terms in the text as a **phrase**

For example: **Friends, Romans, Countrymen** would generate two **biwords/bigrams**:

- **friends romans**
- **romans countrymen**

Each of these biwords is now a **dictionary term**

Two-word phrases can now easily be answered

LONGER PHRASE QUERIES

A long phrase like "**stanford university palo alto**" can be processed by being broken down into a Boolean query on bigrams:

- "**stanford university**" **AND** "**university palo**" **AND** "**palo alto**"

However …

- We need to do post-filtering of hits to identify subset that actually contains the 4-word phrase
- False positives for phrases longer than 2 words
- Index blowup due to bigger dictionary
  - Infeasible for more than biwords, big even for them

SOLUTION 2: POSITIONAL INDEXES

Positional indexes are a more efficient alternative to biword indexes

Postings lists in an **inverted index**: each posting is just a *docID*

Postings lists in a **positional index**: each posting is a ***docID*** and a **list of positions**

<***term***, number of docs containing ***term***;

doc1: position1, position2 … ;

doc2: position1, position2 … ;

etc.>

POSITIONAL INDEXES: EXAMPLE

Query: "**to be or not to be**"

TO, 993427:
⟨1: ⟨7, 18, 33, 72, 86, 231⟩;
 2: ⟨1, 17, 74, 222, 255⟩;
 4: ⟨8, 16, 20, 429, 433⟩;
 5: ⟨363, 367⟩;
 7: ⟨13, 23, 191⟩;…⟩

BE, 178239:
⟨1: ⟨17, 25⟩;
 4: ⟨17, 21, 291, 430, 434⟩;
 5: ⟨14, 19, 101⟩;…⟩

Which of docs 1,2,4,5, 7 could contain "to be or not to be"?

POSITIONAL INDEXES: EXAMPLE

Query: “**to be or not to be**”

TO, 993427:
⟨1: ⟨7, 18, 33, 72, 86, 231⟩;
2: ⟨1, 17, 74, 222, 255⟩;
4: ⟨8, **16**, **20**, 429, 433⟩;
5: ⟨363, 367⟩;
7: ⟨13, 23, 191⟩;…⟩

BE, 178239:
⟨1: ⟨17, 25⟩;
4: ⟨**17**, **21**, 291, 430, 434⟩;
5: ⟨14, 19, 101⟩;…⟩

1. Extract inverted index entries for each distinct term: ***to, be, or, not***
2. Merge their *doc:position* lists to enumerate all positions with “***to be or not to be***”

Document 4 is a match!

MERGING TWO POSITIONAL LISTS: THE ALGORITHM

```
POSITIONALINTERSECT(p1, p2, k)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
      then l ← ⟨ ⟩
           pp1 ← positions(p1)
           pp2 ← positions(p2)
           while pp1 ≠ NIL
           do while pp2 ≠ NIL
              do if |pos(pp1) − pos(pp2)| ≤ k
                    then ADD(l, pos(pp2))
                    else if pos(pp2) > pos(pp1)
                            then break
                 pp2 ← next(pp2)
              while l ≠ ⟨ ⟩ and |l[0] − pos(pp1)| > k
              do DELETE(l[0])
              for each ps ∈ l
              do ADD(answer, ⟨docID(p1), pos(pp1), ps⟩)
              pp1 ← next(pp1)
           p1 ← next(p1)
           p2 ← next(p2)
      else if docID(p1) < docID(p2)
              then p1 ← next(p1)
              else p2 ← next(p2)
return answer
```

PROXIMITY SEARCH

We just saw how to use a positional index for phrase searches

We can also use it for **proximity search**

For example: **employment /4 place**

- Find all documents that contain **employment** and **place** within 4 words of each other
- **Employment agencies that place healthcare workers are seeing growth** is a hit
- **Employment agencies that have learned to adapt now place healthcare workers** is not a hit

POSITIONAL INDEX SIZE

A positional index expands postings storage substantially

Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries

Need an entry for each occurrence, not just once per document

Index size depends on average document size

- Average web page has <1000 terms
- SEC filings, books, even some epic poems … easily 100,000 terms

Consider a term with frequency 0.1%

Document size	Postings	Positional postings
1000	1	1
100,000	1	**100**

- Positional index size 35–50% of volume of original text

COMBINATION SCHEME

Biwords and positional index can be profitably combined

- Many biwords are extremely frequent: "Michael Jackson", "Britney Spears"
- For these biwords it is inefficient to keep on merging positional postings lists → A biwords index is faster

Combination scheme: Include frequent biwords as vocabulary terms in the index. Do all other phrases by positional intersection

Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme

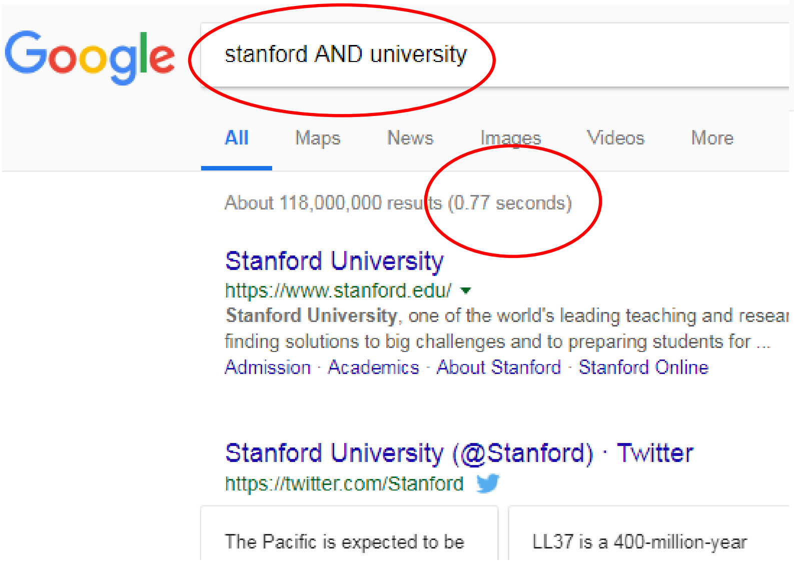
- A typical web query mixture was executed in ¼ of the time of using just a positional index
- It required 26% more space than having a positional index alone

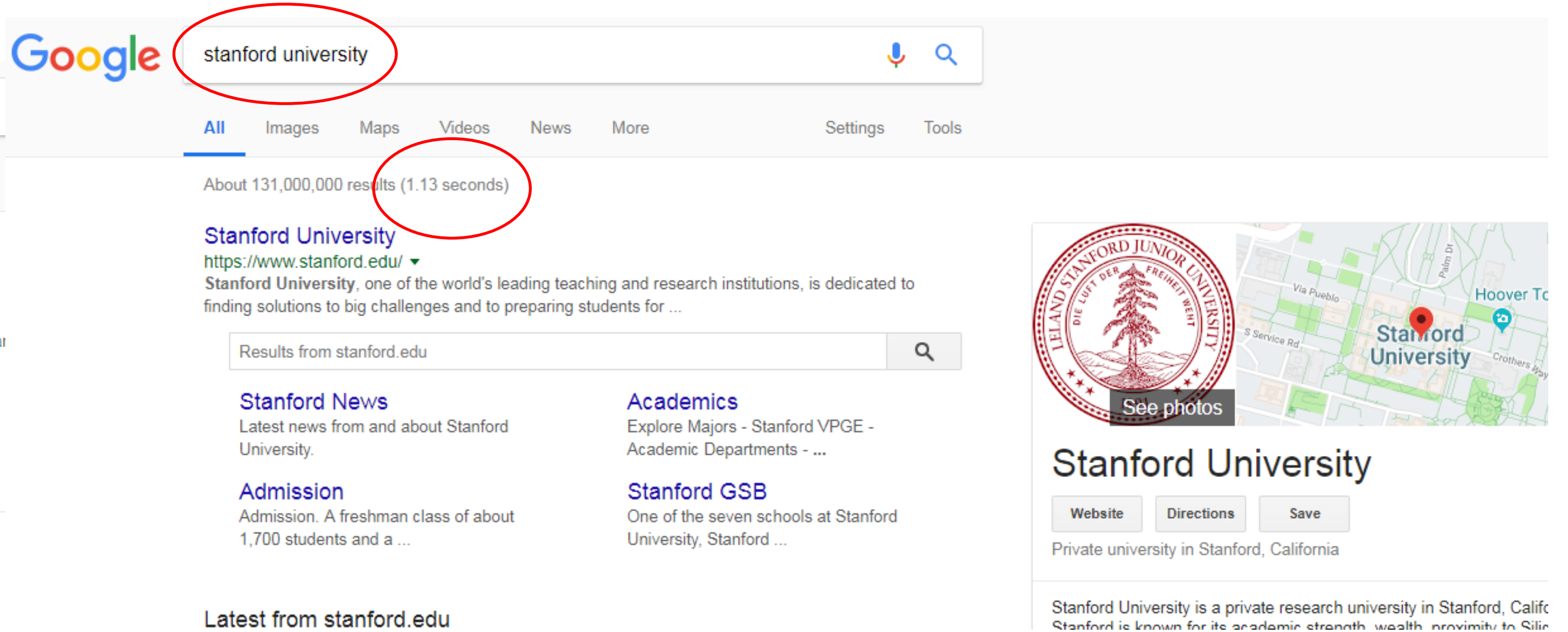
"POSITIONAL" QUERIES ON GOOGLE

For web search engines, **positional queries are much more expensive than regular Boolean queries**

Let's look at the example of phrase queries

- **Why are they more expensive than regular Boolean queries?**





IR BASICS – RANKED RETRIEVAL

RANKED RETRIEVAL

Thus far, our queries have all been 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**

Boolean queries often result in either too few (=0) or too many (1000s) results

Query 1 (Boolean conjunction): [standard user dlink 650]

- → 200,000 hits – **feast**

Query 2 (Boolean conjunction): [standard user dlink 650 no card found]

- → 0 hits – **famine**

In Boolean retrieval, it takes a lot of skill to come up with a query that produces a manageable number of hits

- **AND** gives too few results; **OR** gives too many

RANKED RETRIEVAL MODELS

Rather than a set of documents satisfying a query expression, in **ranked retrieval models**, the system returns an ordering over the (top) documents in the collection with respect to a query

With ranking, large result sets are not an issue

- Just show the top k ($\approx$10) results
- Doesn't overwhelm the user

Premise: the ranking algorithm → More relevant results are ranked higher than less relevant results

SCORING AS THE BASIS OF RANKED RETRIEVAL

How can we accomplish a relevance ranking of the documents with respect to a query?

- Assign a score to each query-document pair, say in [0, 1]
  - This score measures how well document and query “match”
- Sort documents according to scores

SCORING AS THE BASIS OF RANKED RETRIEVAL

How can we accomplish a relevance ranking of the documents with respect to a query?

- Assign a **score** to each query-document pair, say in [0, 1]
  - This score measures how well document and query "match"
- Sort documents according to scores

How do we compute the **score** of a query-document pair?

- If no query term occurs in the document: score should be 0
- The more frequent a query term in the document, the higher the score
- The more query terms occur in the document, the higher the score

TAKE 1: JACCARD COEFFICIENT

A commonly used measure of overlap of two sets

Let A and B be two sets

Jaccard coefficient:

$$Jaccard\ (A,B) = \frac{|A \cap B|}{|A \cup B|}, A \neq 0\ or\ B \neq 0$$

- $Jaccard(A,A) = 1$
- $Jaccard(A,B) = 0\ if\ A \cap B = 0$

A and B don't have to be the same size

Always assigns a number between 0 and 1

What is the query-document match score that the Jaccard coefficient computes for:

- Query: "Ides of March"
- Document "Caesar died in March"

$$Jaccard(q,d) = 1/6$$

WHAT'S WRONG WITH JACCARD?

Jaccard does not consider **term frequency**

- **Rare terms are more informative than frequent terms**

We need a more sophisticated way of normalizing for the length of a document

- Next lecture, we'll use $|A \cap B|/\sqrt{|A \cup B|}$ (cosine) . . .
- . . . instead of $|A \cap B|/|A \cup B|$ (Jaccard) for length normalization

TERM WEIGHTING

RECALL: BINARY TERM-DOCUMENT **INCIDENCE MATRIX**

Shakespeare plays → 37

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
ANTHONY	1	1	0	0	0	1
BRUTUS	1	1	0	1	0	0
CAESAR	1	1	0	1	1	0
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** $\in \{0,1\}^{|V|}$

TERM-DOCUMENT **COUNT MATRIX**

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
ANTHONY	157	73	0	0	0	1
BRUTUS	4	157	0	2	0	0
CAESAR	232	227	0	2	1	0
CALPURNIA	0	10	0	0	0	0
CLEOPATRA	57	0	0	0	0	0
MERCY	2	0	3	8	5	8
WORSER	2	0	1	1	1	0

Considers the number of occurrences of a term in a document:

Each document is **a count vector** in $\mathbb{N}^{|V|}$

BAG OF WORDS (BOW) MODEL

We do not consider the order of words in a document

- John is quicker than Mary and Mary is quicker than John are represented the same way

This is called a bag of words model

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 the course

For now: bag of words model

- A **vector representation** of a document as set words with their respective frequencies

	Antony and Cleopatra	Julius Caesar
ANTHONY	157	73
BRUTUS	4	157
CAESAR	232	227
CALPURNIA	0	10
CLEOPATRA	57	0
MERCY	2	0
WORSER	2	0

$$Anthony\ and\ Cleopatra = (157,4,232,0,57,2,2)$$

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, because:

- A document with $tf = 10$ occurrences of the term is more relevant than a document with $tf = 1$ occurrence of the term
- But not 10 times more relevant

Relevance does not increase proportionally with term frequency

INSTEAD OF RAW FREQUENCY: LOG FREQUENCY WEIGHTING

The log frequency weight of term t in d is defined as:

$$w_{t,d} = \begin{cases} 1 + \log_{10} tf_{t,d} & if \;\; tf_{t,d} > 0 \\ 0 & otherwise \end{cases}$$

$tf_{t,d} \rightarrow w_{t,d}: 0 \;\rightarrow\; 0, \;\; 1 \;\rightarrow\; 1, \;\; 2 \;\rightarrow\; 1.3, \;\; 10 \;\rightarrow\; 2, \;\; 1000 \;\rightarrow\; 4$, etc.

We add 1 to the $\log(tf)$ because when tf is equal to 1, the $\log(1)$ is zero. By adding one, we distinguish between tf = 0 and tf = 1

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

$$tf\text{-}matching\text{-}score(q, d) = \sum_{t \in q \cap d} \left(1 + \log tf_{t,d}\right)$$

The score is 0 if none of the query terms is present in the document

LOG-FREQUENCY WEIGHTING

Why do we use log ?

If tf for word "computer" in $doc1$ is 2 and $doc2$ is 10, we can say that $doc2$ is more relevant than $doc1$

However, if the tf of "computer" for $doc1$ is 100000 and $doc2$ is 200000, at this point, there is no much difference in terms of relevance anymore because they both contain a very high count

ISSUES WITH FREQUENCY WEIGHTING

Consider the "ides of march" query

- *Julius Caesar* has 5 occurrences of ides
- No other play has ides
- march occurs in over a dozen
- All the plays contain lots of of

By this scoring measure, the top-scoring play is likely to be the one with the most ofs

DESIRED WEIGHT FOR **RARE TERMS**

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

DESIRED WEIGHT FOR **FREQUENT TERMS**

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 positive weights for words like high, increase, and line

- But lower weights than for rare terms

DOCUMENT FREQUENCY

We want high weights for rare terms like arachnocentric

We want low (yet positive) weights for frequent terms like good, increase, and line

We will use document frequency to factor this into computing the matching score

The document frequency is **the number of documents in the collection in which the term *t* appears**

INVERSE DOCUMENT FREQUENCY – *idf* WEIGHT

Inverse Document Frequency idf is a measure of informativeness of a term: its rarity across the entire corpus

We define the idf of t by:

$$idf_t = \log_{10}\left(\frac{N}{df_t}\right)$$

N is the number of documents in a collection

We use $\log_{10}\left(\frac{N}{df_t}\right)$ instead of $\frac{N}{df_t}$ to "dampen" the effect of idf

Note that we use the log transformation for both term frequency and document frequency

Assuming N = 1,000,000

term	df_t	idf_t
calpurnia	1	
animal	100	
sunday	1000	
fly	10,000	
under	100,000	
the	1,000,000	

EFFECT OF idf ON RANKING

Question: **Does idf have an effect on ranking for one-term queries**, like

- iPhone?

idf affects the ranking of documents for queries with at least two terms

- For example, in the query “arachnocentric line”, idf weighting increases the relative weight of arachnocentric and decreases the relative weight of line

COLLECTION FREQUENCY 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

Collection frequency of t: number of tokens of t in the collection

Document frequency of t: number of documents in which t appears

Which word is a better search term (and should get a higher weight)?

- Clearly, insurance is a more discriminating search term and should get a higher weight

This example suggests that df (and idf) is better for weighting than cf (and "icf")

tf-idf WEIGHTING

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

$$w_{t,d} = (1 + \log_{10} tf_{t,d}) \times \log_{10} \frac{N}{df_t}$$

Best known weighting scheme in information retrieval

- Note: the “-” in *tf-idf* is a hyphen, not a minus sign!

tf-idf:

- … increases with the number of occurrences within a document (term frequency)
- … increases with the rarity of the term in the collection (inverse document frequency)

BINARY → COUNT → WEIGHT MATRIX

Shakespeare plays → 37

$$\log_{10}\frac{N}{df_t} = \boxed{\log_{10}\frac{37}{3}}$$

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	*df*	*idf*
ANTHONY	157	73	0	0	0	1	3	1.1
BRUTUS	4	157	0	2	0	0	3	1.1
CAESAR	232	227	0	2	1	0	4	0.97
CALPURNIA	0	10	0	0	0	0	1	1.6
CLEOPATRA	57	0	0	0	0	0	1	1.6
MERCY	2	0	3	8	5	8	5	0.9
WORSER	2	0	1	1	1	0	4	0.97

Each document is **a count vector** in $\mathbb{N}^{|V|}$

BINARY → COUNT → WEIGHT MATRIX

$$\boxed{w_{t,d} = (1 + \log_{10} tf_{t,d}) \times \log_{10}\frac{N}{df_t} = (1 + \log_{10} 157) \times \log_{10}\frac{37}{3} = (1 + 2.2) \times 1.1 = 3.52}$$

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	df	idf
ANTHONY	3.52	3.19	0	0	0	1.1	3	1.1
BRUTUS	1.76	3.52	0	1.43	0	0	3	1.1
CAESAR	3.26	3.3	0	1.3	0.97	0	4	0.97
CALPURNIA	0	3.2	0	0	0	0	1	1.6
CLEOPATRA	4.3	0	0	0	0	0	1	1.6
MERCY	1.31	0	1.35	1.71	1.53	1.71	5	0.9
WORSER	1.26	0	0.97	0.97	0.97	0	4	0.97

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

BINARY → COUNT → WEIGHT MATRIX

Final ranking of documents for a query → $score(q, d) = \sum_{t \in q \cap d} tf.idf_{t,d}$

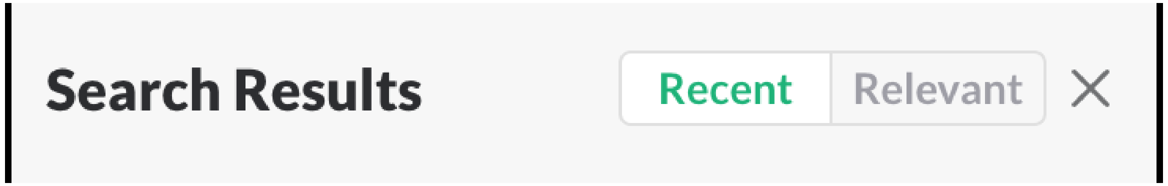
	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	df	idf
ANTHONY	3.52	3.19	0	0	0	1.1	3	1.1
BRUTUS	1.76	3.52	0	1.43	0	0	3	1.1
CAESAR	3.26	3.3	0	1.3	0.97	0	4	0.97
CALPURNIA	0	3.2	0	0	0	0	1	1.6
CLEOPATRA	4.3	0	0	0	0	0	1	1.6
MERCY	1.31	0	1.35	1.71	1.53	1.71	5	0.9
WORSER	1.26	0	0.97	0.97	0.97	0	4	0.97

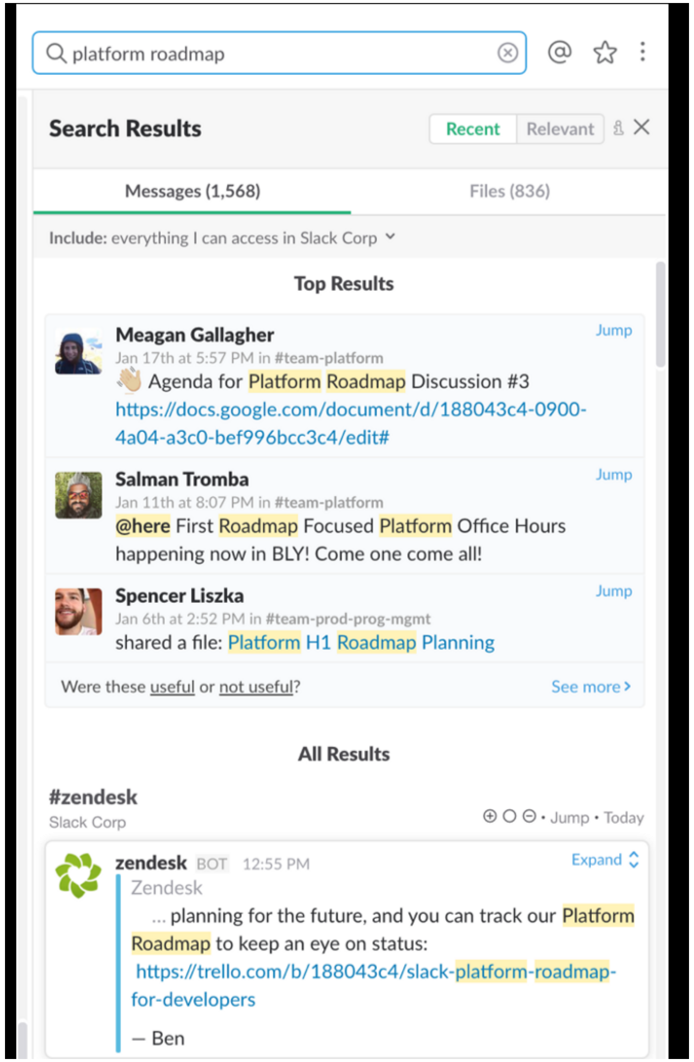
Score for doc $d = Julius\ Caesar\ relative\ to\ q = Anthony\ AND\ Caesar = 6.49$

Score for doc $d = Anthony\ and\ Cleopatra\ relative\ to\ q = Anthony\ AND\ Caesar = 6.78$

THIS WEEK'S READING

Search at Slack: https://slack.engineering/search-at-slack-431f8c80619e





NEXT TIME

Vector Space Model

Measuring Similarity

Evaluating Search Engines

REFERENCES

This lecture is heavily relying on the following courses:

- CS 276 / LING 286: Information Retrieval and Web Search, Stanford University
- Natural Language Processing Lecture Slides from the Stanford Coursera course by Dan Jurafsky and Christopher Manning