Introduction to

Information Retrieval

Introducing ranked retrieval

This unit and repetion

- **Today:** IR, VSM, mini intro to word embeddings
- **Repetition of last unit: NLP Basics**
  - Preprocessing: **tokenization –** what is it?
  - Preprocessing: **stopword removal –** what is it?
  - Preprocessing: **lemmatization / stemming –** what is it?
  - **NLTK** included corpora, functions to make: collocations, frequency distributions, ...

Repetition – unit 1

- What is **IR**? What **evaluation measures** did we discuss?
- What is a **term-document matrix**? How to do Boolean queries on it?
- What is an **inverted index**?
  - Which **runtime** do basic (Boolean) **queries** have on an inverted index?
- What is a **Biword-index?** Why would we want to use it? Why not?

Repetition – unit 1

- What is a **positional index**?
- **Which queries** can we do with the positional index?

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: "*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 models, the system returns an **ordering over the (top) documents** in the collection with respect to 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**
- In principle, there are two separate choices here, but in practice, ranked retrieval models have normally been associated with free text queries and vice versa

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

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

Introduction to

Information Retrieval

Introducing ranked retrieval

Introduction to

Information Retrieval

Scoring with the Jaccard coefficient

Take 1: Jaccard coefficient

- Simple **starting point for ranking**
- A **commonly used measure** of overlap of two sets *A* and *B* is the Jaccard coefficient
- jaccard$(A,B) = |A \cap B| / |A \cup B|$
- 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.**

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*
- **Document 1**: *caesar died in march*
- **Document** 2: *the long march*

- ***Do Jaccard(q, d1) and Jaccard(q, d2) – what are the results***

Exercise in Class (?)

- Implement Jaccard coefficient for some dataset which is short and simple (eg. first 100 words of documents in gutenberg NLTK corpus)
- Can also be used for **document similarity**

Issues with Jaccard for scoring

- It doesn't consider ***term frequency*** (how many times a term occurs in a document)
  - **Rare terms** in a collection are more informative than frequent terms
  - Jaccard doesn't consider this information
- We need a more sophisticated way of **normalizing** for length
  - Later in this lecture, we'll use $|A \cap B|/\sqrt{|A \cup B|}$

    . . . instead of $|A \cap B|/|A \cup B|$ (Jaccard) for length normalization (as used in cosine similarity (later))

Introduction to

Information Retrieval

Scoring with the Jaccard coefficient

Introduction to

Information Retrieval

Term frequency weighting

Recall: 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* $\in \{0,1\}^{|V|}$

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.
- **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 on
  - **For now: bag of words model**

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:
  - 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**.
- Relevance does **not increase proportionally** (linearly) with term frequency.

NB: frequency = count in IR

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

Exercise TDM with log10-tf

- For gutenberg corpus, create a TDM, with log-frequency weighting.
  (To make it faster, again use only the first 100 words per document).

Introduction to

Information Retrieval

Term frequency weighting

Introduction to

Information Retrieval

(Inverse) Document frequency weighting

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 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 document frequency of *t*: the number of documents that contain *t*
  - df_t is an **inverse measure** of the informativeness of *t*
- 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.

Will turn out the base of the log is immaterial.

idf example, suppose *N* = 1M

term	df_t	idf_t
calpurnia	1	→ 6
animal	100	?
sunday	1,000	?
fly	10,000	?
under	100,000	?
the	1,000,000	?

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

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

Effect of idf on ranking

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

Effect of idf on ranking

- Question: 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 a collection, counting multiple occurrences. **Usually not used in IR, because:**
- Example:

Word	Collection frequency	Document frequency
insurance	10440	3997
try	10422	8760

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

Introduction to

Information Retrieval

(Inverse) Document frequency weighting

Introduction to

Information Retrieval

tf-idf weighting

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 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

Final ranking of documents for a query

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

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 $\in \mathbb{R}^{|V|}$

Exercise

- Extend TDM from previous exercise to have tf-idf weights
- You can base your code on `code9_tdm.py`

Introduction to

Information Retrieval

tf-idf weighting

Introduction to

Information Retrieval

The Vector Space Model (VSM)

Documents as vectors

- Now 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

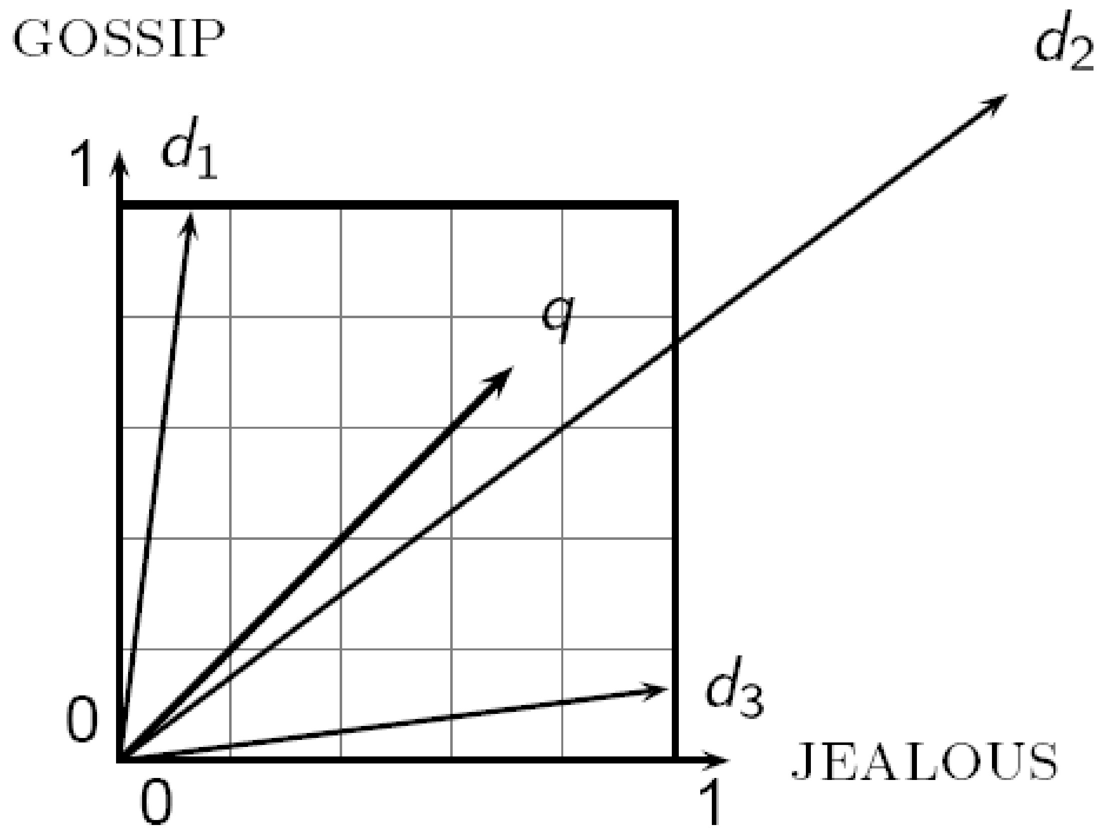
- Key idea 1: Do the same for queries: represent them as vectors in the space
- Key idea 2: Rank documents according to their **proximity to the query** in this space
- proximity = similarity of vectors
- proximity ≈ inverse of distance
- Reason: We do this because we want to get away from the you're-either-in-or-out Boolean model
- Instead: **rank** more relevant documents higher than less relevant documents

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

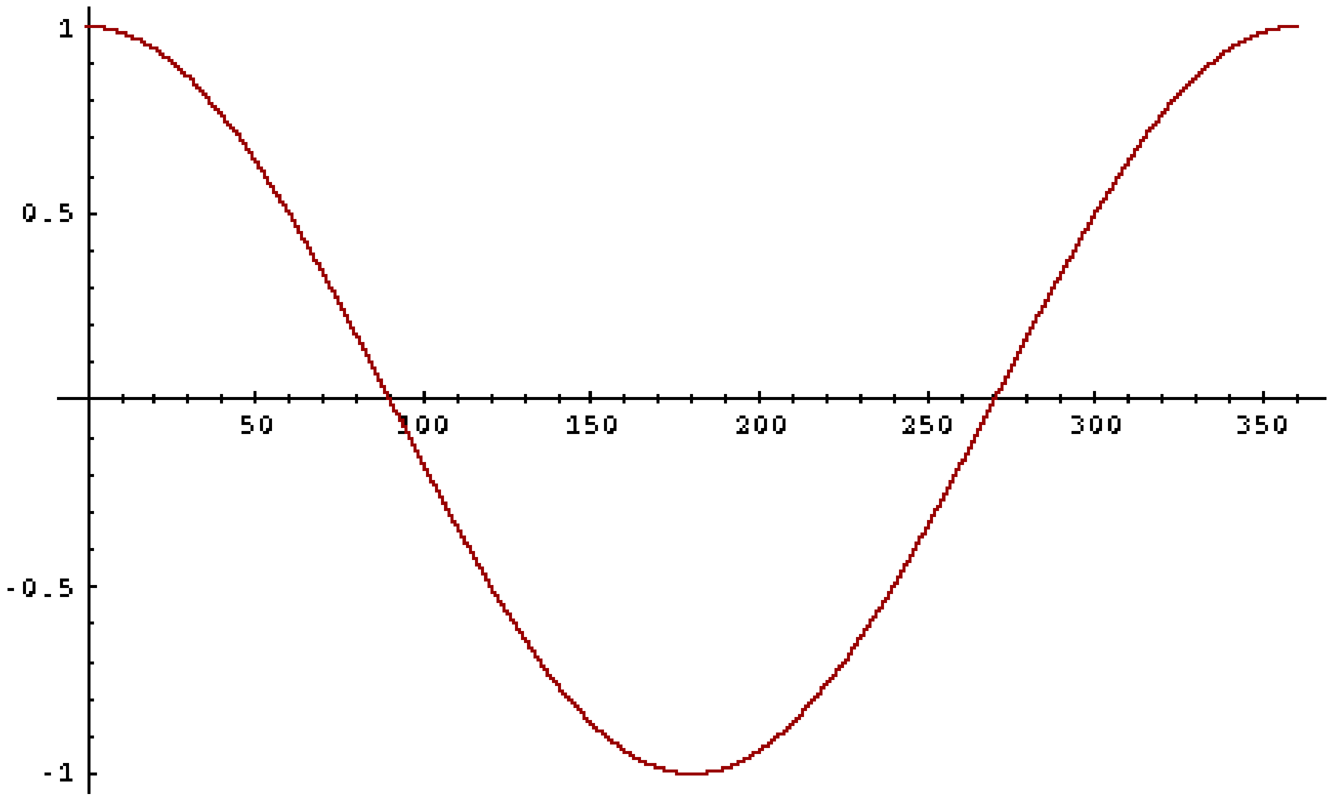
- Thought experiment: take a document *d* and append it to itself. Call this document *d*′.
- "Semantically" *d* and *d*′ have the same content
- The Euclidean distance between the two documents can be quite large
- The **angle between the two documents** is 0, corresponding to maximal similarity.

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

From angles to cosines

- The following two notions are equivalent.
  - Rank documents in 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°]

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.
  - **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 for length-normalized vectors

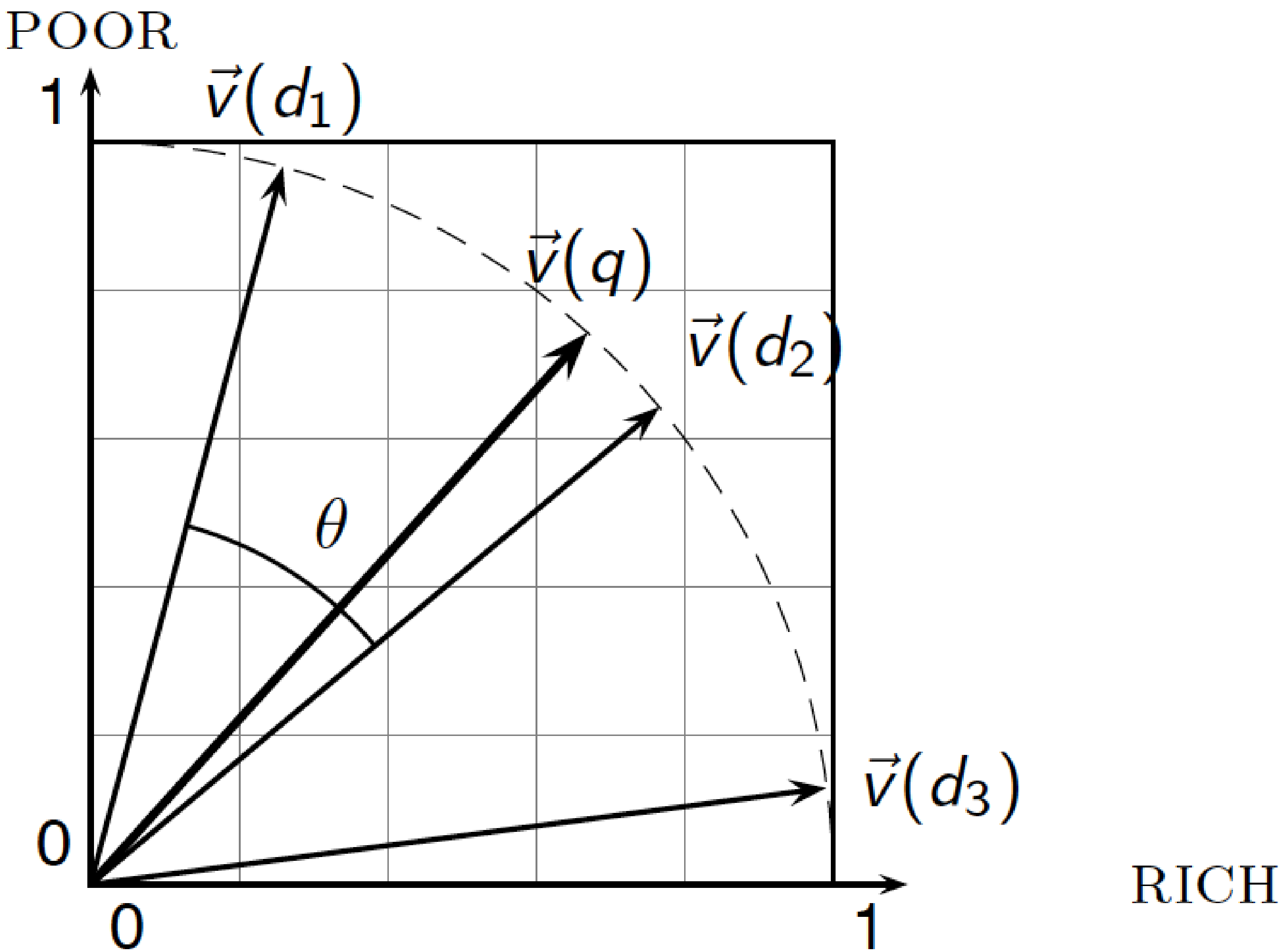
- For **length-normalized vectors**, cosine similarity is simply the **dot product** (or scalar product):

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

for *q*, *d* length-normalized.

(→ Length-normalize document vectors in advance!)

Cosine similarity illustrated



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) ≈
$0.789 \times 0.832 + 0.515 \times 0.555 + 0.335 \times 0.0 + 0.0 \times 0.0 \approx 0.94$
cos(SaS,WH) ≈ 0.79
cos(PaP,WH) ≈ 0.69

Why do we have cos(SaS,PaP) > cos(SAS,WH)?

Big exercise

- Implement the VSM (re-use TDM)
- Implement VSM normalization (with L2 norm)
- Implement cosine-ranked search

Introduction to

Information Retrieval

The Vector Space Model (VSM)

Introduction to

Information Retrieval

Calculating tf-idf cosine scores

in an IR system

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)	$\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$				

Weighting may differ in queries vs documents

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

A bad idea?

- Query: logarithmic tf (l in leftmost column), idf (t in second column), cosine normalization …

tf-idf example: lnc.ltc

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

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

Exercise: 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

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*$[]$

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

Introduction to

Information Retrieval

Calculating tf-idf cosine scores in an IR system

Introduction to

Information Retrieval

Evaluating search engines

Measures for a search engine

- How fast does it index
  - Number of documents/hour
  - (Average document size)
- How fast does it search
  - Latency as a function of index size
- Expressiveness of query language
  - Ability to express complex information needs
  - Speed on complex queries
- Uncluttered UI
- Is it free?

Measures for a search engine

- All of the preceding criteria are *measurable*: we can quantify speed/size
  - we can make expressiveness precise
- The key measure: user happiness
  - What is this?
  - Speed of response/size of index are factors
  - But blindingly fast, useless answers won't make a user happy
- Need a way of quantifying user happiness with the results returned
  - Relevance of results to user's information need

Evaluating an IR system

- An **information need** is translated into a **query**
- Relevance is assessed relative to the **information need** *not* the **query**
- E.g., <u>Information need</u>: *I'm looking for information on whether drinking red wine is more effective at reducing your risk of heart attacks than white wine.*
- <u>Query</u>: ***wine red white heart attack effective***
- You evaluate whether the doc addresses the information need, not whether it has these words

Evaluating ranked results

- Evaluation of a result set:
  - If we have
    - a benchmark document collection
    - a benchmark set of queries
    - assessor judgments of whether documents are relevant to queries

  Then we can use Precision/Recall/F measure as before
- Evaluation of ranked results:
  - The system can return any number of results
  - By taking various numbers of the top returned documents (levels of recall), the evaluator can produce a *precision-recall curve*

Recall/Precision

		R	P
1	R		
2	N		
3	N		
4	R		
5	R		
6	N		
7	R		
8	N		
9	N		
10	N		

Assume 10 rel docs in collection

Two current evaluation measures…

- Mean average precision (MAP)
  - AP: Average of the precision value obtained for the top *k* documents, each time a relevant doc is retrieved
  - Avoids interpolation, use of fixed recall levels
  - Does weight most accuracy of top returned results
  - MAP for set of queries is arithmetic average of APs
    - Macro-averaging: each query counts equally

Introduction to

Information Retrieval

Evaluating search engines

Term-Term Matrices

- In IR systems we typically use Term-Document matrices
- But in many NLP applications we are interested in **term-term co-occurrence** matrices
- Co-occurrence can be measured in different ways, for example
  - Within a unit like a **sentence** or **paragraph**
  - Within a **word window** (left and/or right) of the target word – eg. a word window of 5
  - Q: when could a co-occurrence matrix be useful?

Co-occurrence matrix - Example

1. I enjoy flying.
2. I like NLP.
3. I like deep learning.

The resulting counts matrix will then be:

$$X = \begin{array}{c} \\ I \\ like \\ enjoy \\ deep \\ learning \\ NLP \\ flying \\ . \end{array} \begin{array}{c} \begin{array}{cccccccc} I & like & enjoy & deep & learning & NLP & flying & . \end{array} \\ \begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Co-occurrence Matrices

- Distributional hypothesis – "a word is characterized by the company it keeps"
- Words are defined by their context (words)

A bottle of **tesgüino** is on the table
Everybody likes **tesgüino**
Tesgüino makes you drunk
We make **tesgüino** out of corn.

Co-occurrence Matrices

- These matrices are often input to further processing, eg. for word embeddings, SVD, etc.
- Weighting of the matrix:
  - Raw counts: but not best option
    - "the" and "of" are very frequent, but maybe not the most discriminative
  - PPMI (see next slide)
    - Goal: context is informative about the target word
  - …

Co-occurrence Matrices – PPMI

- Starts from co-occurrence counts

PMI between two words: (Church & Hanks 1989)

Do words x and y co-occur more than if they were independent?

$$\text{PMI}(word_1, word_2) = \log_2 \frac{P(word_1, word_2)}{P(word_1)P(word_2)}$$

- PMI goes from neg. inf. to pos. inf.
- **PPMI:** we just set all negative values to 0.
  - Low counts are unreliable, and neg. relation hard to understand for humans

Co-occurrence Matrices

- Co-occ. matrices are sparse representations
- Next time we look at dense representations, like word embeddings

Mini Intro to Language Models and Word Embeddings

- What are **language models**?
- What are **word embeddings**?
  - Dense, floating point
  - Popular esp since word2vec (Mikolov et al, 2013)
  - Based on distributional hypothesis (Harris, 1957)
  - What does a vector look like, what are characteristics?
  - Trained on large corpora
  - Different training algorithms (GloVe, word2vec, ..)
  - Applications: many, eg. clustering, input to ML/DL, almost NLP task as representations of words

Homework – Background: Wikidata

- **Wikidata**, https://www.wikidata.org
- **Structured data** eg for Wikipedia – for example to make it consistent over language versions
- Consists mainly of **items**, which are described in a Semantic Web-like format, with **triples** of subject-property-object
- https://www.wikidata.org/wiki/Q42
- https://www.wikidata.org/wiki/Property:P249