Document Similarity in Information Retrieval

Mausam

(Based on slides of W. Arms, Dan Jurafsky, Thomas Hofmann, Ata Kaban, Chris Manning, Melanie Martin)
Unstructured data in 1620

• Which plays of Shakespeare contain the words Brutus AND Caesar but NOT Calpurnia?

• One could grep all of Shakespeare’s plays for Brutus and Caesar, then strip out lines containing Calpurnia?

• Why is that not the answer?
  • Slow (for large corpora)
  • NOT Calpurnia is non-trivial
  • Other operations (e.g., find the word Romans near countrymen) not feasible
  • Ranked retrieval (best documents to return)
    • Later lectures
## Term-document incidence matrices

<table>
<thead>
<tr>
<th></th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1 if play contains word, 0 otherwise

- **Brutus AND Caesar BUT NOT Calpurnia**
Incidence vectors

• So we have a 0/1 vector for each term.
• To answer query: take the vectors for *Brutus*, *Caesar* and *Calpurnia* (complemented) → bitwise AND.
  • 110100 AND
  • 110111 AND
  • 101111 =
  • 100100

<table>
<thead>
<tr>
<th></th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Antony and Cleopatra, Act III, Scene ii

*Agrippa [Aside to DOMITIUS ENOBARBUS]*: Why, Enobarbus,
When Antony found Julius *Caesar* dead,
He cried almost to roaring; and he wept
When at Philippi he found *Brutus* slain.

Hamlet, Act III, Scene ii

*Lord Polonius*: I did enact Julius *Caesar* I was killed i’ the Capitol; *Brutus* killed me.
Bigger collections

• Consider \( N = 1 \) million documents, each with about 1000 words.
• Avg 6 bytes/word including spaces/punctuation
  • 6GB of data in the documents.
• Say there are \( M = 500K \) distinct terms among these.
Can’t build the matrix

- 500K x 1M matrix has half-a-trillion 0’s and 1’s.
- But it has no more than one billion 1’s.
  - matrix is extremely sparse.
- What’s a better representation?
  - We only record the 1 positions.
Inverted index

• For each term $t$, we must store a list of all documents that contain $t$.
  • Identify each doc by a docID, a document serial number
• Can we use fixed-size arrays for this?

What happens if the word Caesar is added to document 14?
**Inverted index**

- We need variable-size **postings lists**
  - On disk, a continuous run of postings is normal and best
  - In memory, can use linked lists or variable length arrays
    - Some tradeoffs in size/ease of insertion

```plaintext
Dictionary

<table>
<thead>
<tr>
<th>Term</th>
<th>Postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brutus</td>
<td>1 2 4 11 31 45 173 174</td>
</tr>
<tr>
<td>Caesar</td>
<td>1 2 4 5 6 16 57 132</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>2 31 54 101</td>
</tr>
</tbody>
</table>
```

Sorted by docID (more later on why).
Inverted index construction

Documents to be indexed

Token stream

Modified tokens

Inverted index

Tokenizer

Linguistic modules

Indexer

Friends, Romans, countrymen.

friend

roman

countryman

friend

roman

countryman

2 4

1 2

13 16
Initial stages of text processing

• Tokenization
  • Cut character sequence into word tokens
    • Deal with “John’s”, a state-of-the-art solution

• Normalization
  • Map text and query term to same form
    • You want U.S.A. and USA to match

• Stemming
  • We may wish different forms of a root to match
    • authorize, authorization

• Stop words
  • We may omit very common words (or not)
    • the, a, to, of
Indexer steps: Token sequence

- Sequence of (Modified token, Document ID) pairs.

Doc 1

I did enact Julius Caesar I was killed
i’ the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

<table>
<thead>
<tr>
<th>Term</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>i’</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>hath</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
</tbody>
</table>
Indexer steps: Sort

- Sort by terms
  - And then docID

Core indexing step
Indexer steps: Dictionary & Postings

• Multiple term entries in a single document are merged.
• Split into Dictionary and Postings
• Doc. frequency information is added.

Why frequency? Will discuss later.
Query processing: AND

• Consider processing the query:
  \textit{Brutus AND Caesar}
  
  • Locate \textit{Brutus} in the Dictionary;
    • Retrieve its postings.
  
  • Locate \textit{Caesar} in the Dictionary;
    • Retrieve its postings.
  
  • “Merge” the two postings (intersect the document sets):
The merge

• Walk through the two postings simultaneously, in time linear in the total number of postings entries.

If the list lengths are $x$ and $y$, the merge takes $O(x+y)$ operations.
Crucial: postings sorted by docID.
Intersecting two postings lists (a “merge” algorithm)

```
INTERSECT(p₁, p₂)
  1  answer ← ⟨⟩
  2  while p₁ ≠ NIL and p₂ ≠ NIL
  3    do if docID(p₁) = docID(p₂)
  4      then ADD(answer, docID(p₁))
  5      p₁ ← next(p₁)
  6    p₂ ← next(p₂)
  7  else if docID(p₁) < docID(p₂)
  8      then p₁ ← next(p₁)
  9      else p₂ ← next(p₂)
 10  return answer
```
Phrase queries

• We want to be able to answer queries such as “stanford university” – as a phrase
• Thus the sentence “I went to university at Stanford” is not a match.
  • The concept of phrase queries has proven easily understood by users; one of the few “advanced search” ideas that works

• For this, it no longer suffices to store only <term : docs> entries
A first attempt: Biword indexes

• Index every consecutive pair of terms in the text as a phrase
• For example the text “Friends, Romans, Countrymen” would generate the biwords
  • \textit{friends romans}
  • \textit{romans countrymen}
• Each of these biwords is now a dictionary term
• Two-word phrase query-processing is now immediate.
Longer phrase queries

- Longer phrases can be processed by breaking them down
- \textit{stanford university palo alto} can be broken into the Boolean query on biwords: \textit{stanford university AND university palo AND palo alto}

Without the docs, we cannot verify that the docs matching the above Boolean query do contain the phrase.

Can have false positives!
Issues for biword indexes

• False positives, as noted before
• Index blowup due to bigger dictionary
  • Infeasible for more than biwords, big even for them

• Biword indexes are not the standard solution (for all biwords) but can be part of a compound strategy
Solution 2: Positional indexes

• In the postings, store, for each term the position(s) in which tokens of it appear:

  <term, number of docs containing term;
  doc1: position1, position2 ... ;
  doc2: position1, position2 ... ;
  etc.>
Positional index example

<be: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

Which of docs 1, 2, 4, 5 could contain “to be or not to be”?

• For phrase queries, we use a merge algorithm recursively at the document level
• But we now need to deal with more than just equality
Processing a phrase query

• Extract inverted index entries for each distinct term: \textit{to, be, or, not}.

• Merge their \textit{doc:position} lists to enumerate all positions with \textit{“to be or not to be”}.
  
  • \textit{to}:

  • 2:1,17,74,222,551; 4:8,16,190,429,433; 7:13,23,191; ...

  • \textit{be}:

  • 1:17,19; 4:17,191,291,430,434; 5:14,19,101; ...

• Same general method for proximity searches
Proximity queries

• LIMIT! /3 STATUTE /3 FEDERAL /2 TORT
  • Again, here, /k means “within k words of”.

• Clearly, positional indexes can be used for such queries; biword indexes cannot.

• Exercise: Adapt the linear merge of postings to handle proximity queries. Can you make it work for any value of k?
  • This is a little tricky to do correctly and efficiently
Positional index size

• A positional index expands postings storage \textit{substantially}
  • Even though indices can be compressed

• Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries ... whether used explicitly or implicitly in a ranking retrieval system.
Positional index size

- 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%

<table>
<thead>
<tr>
<th>Document size</th>
<th>Postings</th>
<th>Positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100,000</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>
Rules of thumb

• A positional index is 2–4 as large as a non-positional index

• Positional index size 35–50% of volume of original text
  • Caveat: all of this holds for “English-like” languages
Combination schemes

• These two approaches can be profitably combined
  • For particular phrases ("Michael Jackson", "Britney Spears") it is inefficient to keep on merging positional postings lists
    • Even more so for phrases like "The Who"

• 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
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 $k \ (\approx 10)$ 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
Take 1: Jaccard coefficient

• A commonly used measure of overlap of two sets $A$ and $B$ is the Jaccard coefficient
• $\text{jaccard}(A,B) = \frac{|A \cap B|}{|A \cup B|}$
• $\text{jaccard}(A,A) = 1$
• $\text{jaccard}(A,B) = 0$ 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*
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.
Recall: Binary term-document incidence matrix

<table>
<thead>
<tr>
<th></th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>157</td>
<td>73</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Brutus</td>
<td>4</td>
<td>157</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>232</td>
<td>227</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>57</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
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 $tf_{t,d}$

- 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 with term frequency.
Log-frequency weighting

• The log frequency weight of term \( t \) in \( d \) is

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

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

\[
\text{score} = \sum_{t \in q \cap d} (1 + \log \text{tf}_{t,d})
\]

• The score is 0 if none of the query terms is present in the document.
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., *arachnocratic*)
- A document containing this term is very likely to be relevant to the query *arachnocratic*
- → We want a high weight for rare terms like *arachnocratic*. 
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 $t$
  - $df_t \leq N$

- We define the idf (inverse document frequency) of $t$ by
  $$idf_t = \log_{10} \left( \frac{N}{df_t} \right)$$

- We use $\log \left( \frac{N}{df_t} \right)$ instead of $\frac{N}{df_t}$ to “dampen” the effect of idf.

Will turn out the base of the log is immaterial.
**idf example, suppose N = 1 million**

<table>
<thead>
<tr>
<th>term</th>
<th>$df_t$</th>
<th>$idf_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>calpurnia</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>animal</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>sunday</td>
<td></td>
<td>1,000</td>
</tr>
<tr>
<td>fly</td>
<td></td>
<td>10,000</td>
</tr>
<tr>
<td>under</td>
<td></td>
<td>100,000</td>
</tr>
<tr>
<td>the</td>
<td></td>
<td>1,000,000</td>
</tr>
</tbody>
</table>

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

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.

\[ w_{t,d} = (1 + \log \text{tf}_{t,d}) \times \log_{10}(N / df_t) \]

• 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

\[ \text{Score}(q, d) = \text{tf}.\text{idf}_{t,d} \]
### Binary → count → weight matrix

<table>
<thead>
<tr>
<th></th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>5.25</td>
<td>3.18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.35</td>
</tr>
<tr>
<td>Brutus</td>
<td>1.21</td>
<td>6.1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>8.59</td>
<td>2.54</td>
<td>0</td>
<td>1.51</td>
<td>0.25</td>
<td>0</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1.54</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>2.85</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1.51</td>
<td>0</td>
<td>1.9</td>
<td>0.12</td>
<td>5.25</td>
<td>0.88</td>
</tr>
<tr>
<td>worser</td>
<td>1.37</td>
<td>0</td>
<td>0.11</td>
<td>4.15</td>
<td>0.25</td>
<td>1.95</td>
</tr>
</tbody>
</table>

Each document is now represented by a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$
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
• **Recall:** 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
  • \( d = \text{distance between the end points of the two vectors} \)

• 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 $q$ and $d_2$ is large even though the distribution of terms in the query $q$ and the distribution of terms in the document $d_2$ 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:
  \[ \| \mathbf{x} \|_2 = \sqrt{\sum_i x_i^2} \]

• 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**

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

**Unit vectors**

\[ q_i \text{ is the tf-idf weight of term } i \text{ in the query} \]
\[ d_i \text{ is the tf-idf weight of term } i \text{ in the document} \]

\( \cos(\vec{q}, \vec{d}) \) 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.
Cosine similarity illustrated
### tf-idf weighting has many variants

<table>
<thead>
<tr>
<th>Term frequency</th>
<th>Document frequency</th>
<th>Normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>n (natural)</td>
<td>tf_{t,d}</td>
<td>n (no) 1</td>
</tr>
<tr>
<td>l (logarithm)</td>
<td>1 + log(tf_{t,d})</td>
<td>t (idf) log(N/df_t)</td>
</tr>
<tr>
<td>a (augmented)</td>
<td>0.5 + 0.5 tf_{t,d} / max_t(tf_{t,d})</td>
<td>p (prob idf) max{0, log(N df_t)}</td>
</tr>
<tr>
<td>b (boolean)</td>
<td>{ 1 \text{ if } tf_{t,d} &gt; 0, \ 0 \text{ otherwise}</td>
<td>c (cosine) \frac{1}{\sqrt{w_1^2 + w_2^2 + ... + w_M^2}}</td>
</tr>
<tr>
<td>L (log ave)</td>
<td>\frac{1+log(tf_{t,d})}{1+log(ave_{t \in d}(tf_{t,d}))}</td>
<td>u (pivoted unique) \frac{1}{u}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b (byte size) \frac{1}{\text{CharLength}^\alpha}, \alpha &lt; 1</td>
</tr>
</tbody>
</table>
Similarity Measures Compared

\[
\begin{align*}
\frac{|Q \cap D|}{\sqrt{|Q| + |D|}} & \quad \text{Simple matching (coordination level match)} \\
\frac{2|Q \cap D|}{|Q| + |D|} & \quad \text{Dice’s Coefficient} \\
\frac{|Q \cap D|}{|Q \cup D|} & \quad \text{Jaccard’s Coefficient} \\
\frac{|Q \cap D|}{\sqrt{|Q| \times |D|}} & \quad \text{Cosine Coefficient (what we studied)} \\
\frac{|Q \cap D|}{\min(|Q|, |D|)} & \quad \text{Overlap Coefficient}
\end{align*}
\]
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
Evaluating ranked results: Recall/Precision

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

Assume 10 rel docs in collection
Common evaluation measure...

• 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
Problems

• **Synonyms**: separate words that have the same meaning.
  - E.g. ‘car’ & ‘automobile’
  - They tend to reduce recall

• **Polysems**: words with multiple meanings
  - E.g. ‘Java’
  - They tend to reduce precision

→ The problem is more general: there is a disconnect between *topics and words*
• ‘… a more appropriate model should consider some conceptual dimensions instead of words.’ (Gardenfors)
Latent Semantic Analysis (LSA)

• LSA aims to discover something about the meaning behind the words; about the topics in the documents.
• What is the difference between topics and words?
  • Words are observable
  • Topics are not. They are latent.
• How to find out topics from the words in an automatic way?
  • We can imagine them as a compression of words
  • A combination of words
  • Try to formalise this
Latent Semantic Analysis

- Singular Value Decomposition (SVD)

- $A(m\times n) = U(m\times r) \ E(r\times r) \ V(r\times n)$

- Keep only $k$ eigen values from $E$
  - $A(m\times n) = U(m\times k) \ E(k\times k) \ V(k\times n)$

- Convert terms and documents to points in $k$-dimensional space
Latent Semantic Analysis

- Singular Value Decomposition
  \[ \{A\} = \{U\}\{S\}\{V\}^T \]

- Dimension Reduction
  \[ \{\sim A\} \sim = \{\sim U\}\{\sim S\}\{\sim V\}^T \]
Latent Semantic Analysis

• LSA puts documents together even if they don’t have common words if
  • The docs share frequently co-occurring terms

• Disadvantages:
  • Statistical foundation is missing

  *PLSA addresses this concern!*
PLSA

- Latent Variable model for general co-occurrence data
  - Associate each observation \((w,d)\) with a class variable \(z \in Z\{z_1,\ldots,z_K\}\)

- Generative Model
  - Select a doc with probability \(P(d)\)
  - Pick a latent class \(z\) with probability \(P(z|d)\)
  - Generate a word \(w\) with probability \(p(w|z)\)
PLSA

To get the joint probability model

\[ P(d, w) = P(d)P(w|d), \text{ where} \]
\[ P(w|d) = \sum_{z \in Z} P(w|z)P(z|d). \]

\[ P(d, w) = \sum_{z \in Z} P(z)P(w|z)P(d|z). \]
Model fitting with EM

• We have the equation for log-likelihood function from the aspect model, and we need to maximize it.

\[ \mathcal{L} = \sum_{d \in D} \sum_{w \in W} n(d, w) \log P(d, w), \]

• Expectation Maximization (EM) is used for this purpose.
EM Steps

• **E-Step**
  • Expectation step where expectation of the likelihood function is calculated with the current parameter values

• **M-Step**
  • Update the parameters with the calculated posterior probabilities
  • Find the parameters that maximizes the likelihood function
E Step

• It is the probability that a word $w$ occurring in a document $d$, is explained by aspect $z$

$$P(z|d,w) = \frac{P(z)P(d|z)P(w|z)}{\sum_{z'} P(z')P(d|z')P(w|z')} ,$$

(based on some calculations)
M Step

• All these equations use $p(z|d,w)$ calculated in E Step

\[
P(w|z) = \frac{\sum_d n(d, w) P(z|d, w)}{\sum_{d, w'} n(d, w') P(z|d, w')},
\]

\[
P(d|z) = \frac{\sum_w n(d, w) P(z|d, w)}{\sum_{d', w} n(d', w) P(z|d', w')},
\]

\[
P(z) = \frac{1}{R} \sum_{d, w} n(d, w) P(z|d, w), \quad R \equiv \sum_{d, w} n(d, w).
\]
<table>
<thead>
<tr>
<th>“Arts”</th>
<th>“Budgets”</th>
<th>“Children”</th>
<th>“Education”</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW</td>
<td>MILLION</td>
<td>CHILDREN</td>
<td>SCHOOL</td>
</tr>
<tr>
<td>FILM</td>
<td>TAX</td>
<td>WOMEN</td>
<td>STUDENTS</td>
</tr>
<tr>
<td>SHOW</td>
<td>PROGRAM</td>
<td>PEOPLE</td>
<td>SCHOOLS</td>
</tr>
<tr>
<td>MUSIC</td>
<td>BUDGET</td>
<td>CHILD</td>
<td>EDUCATION</td>
</tr>
<tr>
<td>MOVIE</td>
<td>BILLION</td>
<td>YEARS</td>
<td>TEACHERS</td>
</tr>
<tr>
<td>PLAY</td>
<td>FEDERAL</td>
<td>FAMILIES</td>
<td>HIGH</td>
</tr>
<tr>
<td>MUSICAL</td>
<td>YEAR</td>
<td>WORK</td>
<td>PUBLIC</td>
</tr>
<tr>
<td>BEST</td>
<td>SPENDING</td>
<td>PARENTS</td>
<td>TEACHER</td>
</tr>
<tr>
<td>ACTOR</td>
<td>NEW</td>
<td>SAYS</td>
<td>BENNETT</td>
</tr>
<tr>
<td>FIRST</td>
<td>STATE</td>
<td>FAMILY</td>
<td>MANIGAT</td>
</tr>
<tr>
<td>YORK</td>
<td>PLAN</td>
<td>WELFARE</td>
<td>NAMPHY</td>
</tr>
<tr>
<td>OPERA</td>
<td>MONEY</td>
<td>MEN</td>
<td>STATE</td>
</tr>
<tr>
<td>THEATER</td>
<td>PROGRAMS</td>
<td>PERCENT</td>
<td>PRESIDENT</td>
</tr>
<tr>
<td>ACTRESS</td>
<td>GOVERNMENT</td>
<td>CARE</td>
<td>ELEMENTARY</td>
</tr>
<tr>
<td>LOVE</td>
<td>CONGRESS</td>
<td>LIFE</td>
<td>HAITI</td>
</tr>
</tbody>
</table>

The William Randolph Hearst Foundation will give $1.25 million to Lincoln Center, Metropolitan Opera Co., New York Philharmonic and Juilliard School. “Our board felt that we had a real opportunity to make a mark on the future of the performing arts with these grants an act every bit as important as our traditional areas of support in health, medical research, education and the social services.” Hearst Foundation President Randolph A. Hearst said Monday in announcing the grants. Lincoln Center’s share will be $200,000 for its new building, which will house young artists and provide new public facilities. The Metropolitan Opera Co. and New York Philharmonic will receive $400,000 each. The Juilliard School, where music and the performing arts are taught, will get $250,000. The Hearst Foundation, a leading supporter of the Lincoln Center Consolidated Corporate Fund, will make its usual annual $100,000 donation, too.
The performance of a retrieval system based on this model (PLSI) was found superior to that of both the vector space based similarity (cos) and a non-probabilistic latent semantic indexing (LSI) method. (We skip details here.)

From Th. Hofmann, 2000
Comparing PLSA and LSA

• LSA and PLSA perform dimensionality reduction
  • In LSA, by keeping only K singular values
  • In PLSA, by having K aspects

• Comparison to SVD
  • U Matrix related to \( P(d|z) \) (doc to aspect)
  • V Matrix related to \( P(z|w) \) (aspect to term)
  • E Matrix related to \( P(z) \) (aspect strength)

• The main difference is the way the approximation is done
  • PLSA generates a model (aspect model) and maximizes its predictive power
  • Selecting the proper value of K is heuristic in LSA
  • Model selection in statistics can determine optimal K in PLSA