Lecture 2: Overview of Information Retrieval

ISM293 - Introduction to Computational Advertising
University of California, Santa Cruz
Spring 2009
Instructors: Andrei Broder and Vanja Josifovski
Acknowledgements and Authorship

- Slides in this lecture are based on slides of lectures 1, 2 and 6 of CS276 at Stanford
  - Prabhakar Raghavan
  - Chris Manning
Lecture Overview

- Overview – what is Information Retrieval (IR), basic system components, documents and queries
- Document Analysis
- Models for IR
- Indexing and Query Evaluation
**IR from 100,000 feet**

- **Collection**: Fixed set of documents
- **Query**: Description of the user’s **information need**
- **Goal**: Retrieve documents with information that is **relevant** to user’s information need and helps him complete a **task**
I want to stop sucking my thumb without suffering

Info on devices that prevent thumb sucking

What kind of gloves are there that prevent thumb sucking

Query: thumb sucking gloves
Searching through documents

- 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**?
  - 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)
Conceptual view of the document corpus: A 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>0</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

• 0/1 vector for each term.

• To answer a Boolean Query: bitwise AND of the vectors for Brutus, Caesar and Calpurnia (complemented)
  • $110100 \text{ AND } 110111 \text{ AND } 101111 = 100100$.
  • Each bit denotes one document.
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.
Document Retrieval System Architecture

Indexing

1. Docs
   - Doc Analysis
     - Inversion
       - Index write

Retrieval

1. Query
   - Query Analysis
     - Query Evaluation

Terms:
- per document term list
- per term document list
- inverted index
- dictionary
- posting lists
Important question: How good are the retrieved docs?

- **Precision**: Fraction of retrieved docs that are relevant to user’s information need
- **Recall**: Fraction of relevant docs in collection that are retrieved
- More precise definitions and measurements to follow in later lectures
Document Analysis
To retrieve a document, we need to understand what it is about
- The IR system “reads it”

Start with a raw document and produce a sequence of atomic document units: terms

Sequence of steps grouped into two phases:

- **Lexical Analysis:** word separation, eliminate basic word variations.
  - Result: sequence of tokens

- **Semantic Analysis:** entity extraction, grammatical and conceptual structure of the documents.
  - Result: sequence of terms
Before parsing a document

- What format is it in?
  - pdf/word/excel/html?
- What language is it in?
- What character set is in use?
Tokenization

- **Input**: “*Friends, Romans and Countrymen*”
- **Output**: Tokens
  - *Friends*
  - *Romans*
  - *Countrymen*
- Each such token is now a candidate for an index entry, after further processing
  - Described below
- But what are valid tokens to emit?
Tokenization

- Issues in tokenization:
  - Finland’s capital → Finland? Finlands? Finland’s?
  - Hewlett-Packard → Hewlett and Packard as two tokens?
    - state-of-the-art: break up hyphenated sequence.
    - co-education
    - lowercase, lower-case, lower case?
    - It’s effective to get the user to put in possible hyphens
  - San Francisco: one token or two? How do you decide it is one token?
Tokenization: language issues

- French
  - *L'ensemble* → one token or two?
    - Want *l’ensemble* to match with *un ensemble*

- German noun compounds are not segmented
  - *Lebensversicherungsgesellschaftsangestellter*
  - ‘life insurance company employee’
  - German retrieval systems benefit greatly from a *compound splitter* module

- And then Japanese with multiple alphabets!

\[\begin{array}{cccc}
\text{フォーチュン} & \text{500社} & \text{情報不足} & \text{ため} \\
\text{Katakana} & \text{Hiragana} & \text{Kanji} & \text{Romaji}
\end{array}\]

\$500K (約6,000万円)
Stop words

• With a stop list, you exclude from dictionary entirely the commonest words. Intuition:
  • They have little semantic content: *the*, *a*, *and*, *to*, *be*
  • There are a lot of them: ~30% of postings for top 30 wds

• Modern Web Search engines do not eliminate stop words:
  • Good compression techniques means the space for including stopwords in a system is very small
  • Good query optimization techniques mean you pay little at query time for including stop words.
  • You need them for:
    • Phrase queries: “King of Denmark”, “Let it be”, “To be or not to be”
  • What about ads?
Normalization

• Need to “normalize” terms in indexed text as well as query terms into the same form
  • We want to match U.S.A. and USA

• We most commonly implicitly define equivalence classes of terms
  • e.g., by deleting periods in a term

• Alternative is to do asymmetric expansion:
  • Enter: window  Search: window, windows
  • Enter: windows  Search: Windows, windows, window
  • Enter: Windows  Search: Windows

• Potentially more powerful, but less efficient
Case folding

- Reduce all letters to lower case
  - exception: upper case in mid-sentence?
    - e.g., *General Motors*
    - *Fed* vs. *fed*
    - *SAIL* vs. *sail*
  - Often best to lower case everything, since users will use lowercase regardless of ‘correct’ capitalization…

- Aug 2005 Google example:
  - *C.A.T.* ➔ Cat Fanciers website not Caterpillar Inc.
Thesauri

- Handle synonyms and homonyms
  - Hand-constructed equivalence classes
    - e.g., *car* = *automobile*
    - *color* = *colour*
- Rewrite to form equivalence classes
- Index such equivalences
  - When the document contains *automobile*, index it under *car* as well (usually, also vice-versa)
- Or expand query?
  - When the query contains *automobile*, look under *car* as well
Stemming

- Reduce terms to their “roots” before indexing
- “Stemming” suggest crude affix chopping
  - language dependent
  - e.g., automate(s), automatic, automation all reduced to automat.

For example, compressed and compression are both accepted as equivalent to compress.
Porter’s algorithm

- Commonest algorithm for stemming English
  - Results suggest it’s at least as good as other stemming options
- Conventions + 5 phases of reductions
  - phases applied sequentially
  - each phase consists of a set of commands
  - sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*
Typical rules in Porter

- **sses** $\rightarrow$ **ss**
- **ies** $\rightarrow$ **i**
- **ational** $\rightarrow$ **ate**
- **tional** $\rightarrow$ **tion**

- Weight of word sensitive rules

- $(m>1)$ **EMENT** $\rightarrow$
  - **replacement** $\rightarrow$ **replac**
  - **cement** $\rightarrow$ **cement**
Other stemmers

- Other stemmers exist, e.g., Lovins stemmer http://www.comp.lancs.ac.uk/computing/research/stemming/general/lovins.htm
  - Single-pass, longest suffix removal (about 250 rules)

- Full morphological analysis – at most modest benefits for retrieval

- Do stemming and other normalizations help?
  - English: very mixed results. Helps recall for some queries but harms precision on others
    - E.g., operative (dentistry) ⇒ oper
  - Definitely useful for Spanish, German, Finnish, ...
Query Analysis

- Break the query into atomic units – terms
- Parsing, tokenization, stemming, etc. **must** match document analysis
- Key issue: **query interpretation**
  - Segment the query into suitable units
  - Determine the intent of the user
- Key issue: **queries are short**
  - Query expansion with external sources of knowledge
  - See lecture note on (optional) reading on query expansion
Information Retrieval Models
What are IR Models?

• How to compare the documents and queries?
  • Assess how good is the match
• Model – algebraic framework with well defined operators
• We will cover two IR models
  • Boolean
    • Term weights: Boolean values (0/1)
    • Operators: Boolean algebra operators
    • Results: Boolean values (0/1)
  • Vector Space
    • Term weights: based on the term importance globally and in the given document
    • Operators: Angles and distances in Euclidian space
    • Result: Score for each document
Boolean Model: Exact match with Boolean expressions

- The Boolean Retrieval model is being able to ask a query that is a Boolean expression:
  - Boolean Queries are queries using AND, OR and NOT to join query terms
    - Views each document as a set of words
    - Is precise: document matches condition or not.
- Primary commercial retrieval tool for 3 decades.
- Professional searchers (e.g., lawyers) still like Boolean queries:
  - You know exactly what you’re getting.
Problem with the 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 skill to come up with a query that produces a manageable number of hits.
- With a ranked list of documents it does not matter how large the retrieved set is.
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

- \( \text{jaccard}(A,B) = \frac{|A \cap B|}{|A \cup B|} \)
- \( \text{jaccard}(A,A) = 1; \quad \text{jaccard}(A,B) = 0 \text{ if } A \cap B = 0 \)
- \( A \) and \( B \) don’t have to be the same size.
- Always assigns a number between 0 and 1.
- Example:
  - **Query**: *ides of march*
  - **Document 1**: *caesar died in march*
  - **Document 2**: *the long march*
- Issues:
  - All terms given equal weight
  - Long documents penalized
Term weighting

- Importance of a term
  - **Global** – how much information is there in a term (‘IBM’ vs. ‘maybe’) independent of how/where it appears in the document?
  - **Local** – how important is a term for the give document (footer vs. title)?
- What information can be used to asses these two?
Simplification: 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.
Local importance: Term Frequency

- Consider the number of occurrences of a term in a document:
  - Each document is a count vector in $\mathbb{N}^v$: a column below
- The term frequency $(tf_{t,d})$ of term $t$ in document $d$ is defined as the number of times that $t$ occurs in $d$.

<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>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>
Term frequency $tf$

- How to use $tf$ to weight the terms?
- Raw term frequency is not what we want:
  - A document with 10 occurrences of the term is more relevant than a document with one occurrence of the term.
  - But not 10 times more relevant.
  - Adversarial behavior – spam
- Relevance does not increase proportionally with term frequency.
Log-frequency weighting

- The log frequency weight of term $t$ in $d$ is
  \[ w_{t,d} = \begin{cases} 
  1 + \log_{10} tf_{t,d}, & \text{if } tf_{t,d} > 0 \\
  0, & \text{otherwise}
  \end{cases} \]

- $0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 1.3$, $10 \rightarrow 2$, $1000 \rightarrow 4$, etc.
- 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., EXL257)
- A document containing this term is very likely to be relevant to the query EXL257
- We want a high weight for rare terms like EXL257.
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 in the score.

df (≤ N) is the number of documents that contain the term
idf weight

- $df_t$ is the document frequency of $t$: the number of documents that contain $t$
  - $df$ is a measure of the informativeness of $t$
- We define the idf (inverse document frequency) of $t$ by
  - We use $\log N/df_t$ instead of $N/df_t$ to “dampen” the effect of idf.

$$idf_t = \log_{10} \frac{N}{df_t}$$
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>1</td>
<td>6</td>
</tr>
<tr>
<td>animal</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>sunday</td>
<td>1,000</td>
<td>3</td>
</tr>
<tr>
<td>fly</td>
<td>10,000</td>
<td>2</td>
</tr>
<tr>
<td>under</td>
<td>100,000</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1,000,000</td>
<td>0</td>
</tr>
</tbody>
</table>

There is one idf value for each term $t$ in a collection.
The collection frequency of $t$ is the number of occurrences of $t$ in the collection, counting multiple occurrences.

Example:

<table>
<thead>
<tr>
<th>Word</th>
<th>Collection frequency</th>
<th>Document frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>insurance</td>
<td>10440</td>
<td>3997</td>
</tr>
<tr>
<td>try</td>
<td>10422</td>
<td>8760</td>
</tr>
</tbody>
</table>

Which word is a better search term (and should get a higher weight)?
Compete tf-idf weighting

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

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

- 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
Binary $\rightarrow$ count $\rightarrow$ weight matrix

<table>
<thead>
<tr>
<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>0.35</td>
</tr>
<tr>
<td>Brutus</td>
<td>1.21</td>
<td>6.1</td>
<td>0</td>
<td>1</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>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1.54</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>
</tr>
<tr>
<td>mercy</td>
<td>1.51</td>
<td>0</td>
<td>1.9</td>
<td>0.12</td>
<td>5.25</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>
</tr>
</tbody>
</table>

Each document is now represented by a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$
Vector Space Model: Documents and Queries as Vectors

- So we have a $|V|$-dimensional vector space
- Terms are axes of the space
- Documents are points or vectors in this space
- Very high-dimensional: hundreds of millions of dimensions when you apply this to a web search engine
- This is a very sparse vector - 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 $\approx$ inverse of distance
Formalizing vector space proximity

• First cut: distance between two points
  • ( = distance between the end points of the two vectors)
• Euclidean distance?
• Euclidean distance of the raw vectors does not work for vectors of different length
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. 
- 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 $[0^\circ, 180^\circ]$
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:

$$\left\| \vec{x} \right\|_2 = \sqrt{\sum_i x_i^2}$$

- Dividing a vector by its $L_2$ norm makes it a unit (length) vector

- Effect on the two documents $d$ and $d'$ ($d$ appended to itself) from earlier slide: they have identical vectors after length-normalization.
The cosine similarity between a query `q` and a document `d` is given by:

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

where:
- `q_i` is the tf-idf weight of term `i` in the query `q`.
- `d_i` is the tf-idf weight of term `i` in the document `d`.

The query `q` and document `d` can be represented as unit vectors, and the cosine similarity is the dot product of these vectors. In other words, `cos(\vec{q}, \vec{d})` is the cosine of the angle between `\vec{q}` and `\vec{d}`.
Cosine similarity amongst 3 documents

How similar are
the novels

**SaS**: *Sense and Sensibility*

**PaP**: *Pride and Prejudice*, and

**WH**: *Wuthering Heights?*

<table>
<thead>
<tr>
<th>term</th>
<th>SaS</th>
<th>PaP</th>
<th>WH</th>
</tr>
</thead>
<tbody>
<tr>
<td>affection</td>
<td>115</td>
<td>58</td>
<td>20</td>
</tr>
<tr>
<td>jealous</td>
<td>10</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>gossip</td>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>wuthering</td>
<td>0</td>
<td>0</td>
<td>38</td>
</tr>
</tbody>
</table>

Term frequencies (counts)
### Log frequency weighting

<table>
<thead>
<tr>
<th>term</th>
<th>SaS</th>
<th>PaP</th>
<th>WH</th>
</tr>
</thead>
<tbody>
<tr>
<td>affection</td>
<td>3.06</td>
<td>2.76</td>
<td>2.30</td>
</tr>
<tr>
<td>jealous</td>
<td>2.00</td>
<td>1.85</td>
<td>2.04</td>
</tr>
<tr>
<td>gossip</td>
<td>1.30</td>
<td>0.0</td>
<td>1.78</td>
</tr>
<tr>
<td>wuthering</td>
<td>0.0</td>
<td>0.0</td>
<td>2.58</td>
</tr>
</tbody>
</table>

### After normalization

<table>
<thead>
<tr>
<th>term</th>
<th>SaS</th>
<th>PaP</th>
<th>WH</th>
</tr>
</thead>
<tbody>
<tr>
<td>affection</td>
<td>0.789</td>
<td>0.832</td>
<td>0.524</td>
</tr>
<tr>
<td>jealous</td>
<td>0.515</td>
<td>0.555</td>
<td>0.465</td>
</tr>
<tr>
<td>gossip</td>
<td>0.335</td>
<td>0.0</td>
<td>0.405</td>
</tr>
<tr>
<td>wuthering</td>
<td>0.0</td>
<td>0.0</td>
<td>0.588</td>
</tr>
</tbody>
</table>

\[
\cos(SaS, PaP) \approx 0.789 \times 0.832 + 0.515 \times 0.555 + 0.335 \times 0.0 + 0.0 \times 0.0 \\
\approx 0.94
\]

\[
\cos(SaS, WH) \approx 0.79
\]

\[
\cos(PaP, WH) \approx 0.69
\]

**Why do we have \(\cos(SaS, PaP) > \cos(SaS, WH)\)?**
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) tf&lt;sub&gt;t,d&lt;/sub&gt;</td>
<td>n (no) 1</td>
<td>n (none) 1</td>
</tr>
<tr>
<td>l (logarithm) 1 + log(tf&lt;sub&gt;t,d&lt;/sub&gt;)</td>
<td>t (idf) log&lt;sub&gt;df&lt;/sub&gt;N</td>
<td>c (cosine) ( \frac{1}{\sqrt{w_1^2 + w_2^2 + \ldots + w_M^2}} )</td>
</tr>
<tr>
<td>a (augmented) 0.5 + ( \frac{0.5 \times tf_{t,d}}{\max_t(tf_{t,d})} )</td>
<td>p (prob idf) max{0, log ( \frac{N - df_t}{df_t} )}</td>
<td>u (pivoted unique) 1/u</td>
</tr>
<tr>
<td>b (boolean) ( \begin{cases} 1 &amp; \text{if } tf_{t,d} &gt; 0 \ 0 &amp; \text{otherwise} \end{cases} )</td>
<td></td>
<td>b (byte size) 1/( \text{CharLength}^\alpha ), ( \alpha &lt; 1 )</td>
</tr>
<tr>
<td>L (log ave) ( \frac{1 + \log(tf_{t,d})}{1 + \log(\text{ave}<em>{t \subseteq d}(tf</em>{t,d}))} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
Document Indexing and Query Evaluation
Document representation

- Document x Term matrix
  - Entries for each Document-Term combination
  - Per Term entries
  - Per Document entries
- Consider $N = 1M$ documents, each with about 1K terms.
- Avg 6 bytes/term incl 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 $T$.
- Do we use an array or a list for this?

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

- Two main data structures:
- *Term posting list* – list of all documents where the term appears
- *Dictionary* is used to find the per term data (including the start of the posting list)

Sorted by docID (more later on why).
Document Retrieval System Architecture

**Indexing**
- Docs
  - Doc Analysis
    - Inversion
      - Index write
        - per document term list
        - per term document list

**Retrieval**
- Query
  - Query Analysis
  - Query Evaluation

**Inverted Index**
- Dictionary
- Posting Lists
Step One: Document Analysis

- Sequence of (term, docID) pairs.
- Sorted by docID – we process one document at that time
- Easy to parallelize – divide the documents among the available nodes

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

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Doc #</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>was</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
</tbody>
</table>
Step Two: Inversion

- Sort by <term, docID>
- Group all occurrences of the same term across all documents
- Can be the computationally most intense part of the index building
- How to parallelize?
  - Merge-Sort: sort individual runs locally then merge runs from different nodes
  - Hadoop – Map-Reduce infrastructure
Step Three: Index Write

- Write a Dictionary file and a Postings file
- Sequential process

<table>
<thead>
<tr>
<th>Term</th>
<th>Doc #</th>
<th>Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>hath</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>N docs</th>
<th>Col freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>be</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>it</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>noble</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>you</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Query Evaluation

How to use the index data structures?
Consider processing the query:

\textbf{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:
Posting list merging

- Two general approaches
  - Term at the Time (TAAT):
    - candidate_set = first posting list
    - for each of the remaining terms t
      - candidate_set = intersection(candidate_set, t.posting_list());
  - Document at the Time (DAAT)
    - Open an iterator \( \text{cursor} \) at the posting list beginning
    - Move the cursors forward through the posting lists simultaneously:
      - cursor.next() moves to the next entry in the posting list
      - cursor.goto(docID) moves to the posting for docID or the first document with doc ID larger than docID
    - To perform an OR query we need an union ➔ move the minimum cursor
    - Exercise DAAT for AND
Merging – further questions

What about an arbitrary Boolean formula?

\[(\text{Brutus OR Caesar}) \land \neg (\text{Antony OR Cleopatra})\]

• Can we always merge in “linear” time?
  • Linear in what?

• Can we do better – again think about this in the DAAT exercise (How can you skip efficiently?)
Improving the TAAT algorithms

- What is the best order for query processing?
- Consider a query that is an AND of $t$ terms.
- For each of the $t$ terms, get its postings, then AND them together.

Query: **Brutus AND Calpurnia AND Caesar**
Improving the TAAT algorithms

- Process in order of increasing freq:
  - start with smallest set, then keep cutting further.

Brutus

| 2 | 4 | 8 | 16 | 32 | 64 | 128 |

Calpurnia

| 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 |

Caesar

| 13 | 16 |

Execute the query as (Caesar AND Brutus) AND Calpurnia.
The Merge in DAAT Algorithms

- 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.
Conclusion

• Wealth of work on how to break down documents and queries into atomic units – “terms”
• Several models proposed to compare the documents and queries and retrieve the ones that are most similar
• Practice shows that tuning is very important in IR systems. Many parameters depending on the:
  • Corpus
  • Queries
• Textual advertising lectures: how to apply IR to selection of ads
• Project: how to start testing with your own ad selection based on open source search engines
Exercise
Exercise – DAAT AND algorithm

- Document at the Time Algorithm:
  - Process all posting lists simultaneously
  - Keep a cursor to indicate the current position
  - Cursor interface:
    - position at the first posting in a posting list for a term: `cursor.init(term)`
    - next posting: `cursor.next()`
    - at or after a given docID: `cursor.goto(docID)`
- Write a pseudo-code of an algorithm for AND processing using the cursor interface: `AND(list<terms> terms, int k){…….}`
- First provide a basic, but correct version
- Optimize – can you do sub-linear?