EECS 395/495 Lecture 2: Intro to IR and Inverted Indices

Based on slides by Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze, and Dan Weld
Reminder

• Project Proposals Due next Monday
  – April 11, 11:59PM

• Sign up for project meetings April 12 and 13
  – Link to schedule will be sent via e-mail
Goals

• How does Web search work?
  – Tuesdays

• What’s the future of Web search?
  – Thursdays
How Does Web Search Work?

- **Inverted Indices**
  - Inverted Indices enable fast IR
  - Tokenization & Linguistic issues
  - Handling phrase queries
    - Scalable, distributed construction & deployment

- Web Crawlers

- Ranking Algorithms
Information Retrieval (IR) Example

• 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)
  – Other operations (e.g., find the word *Romans* near *countrymen*) not feasible
  – Ranking? (best documents to return)
    • Later lectures
### 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>

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.

• \[110100 \text{ AND } 110111 \text{ AND } 101111 = 100100.\]
Answers to query

• 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.
Indexer steps

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

<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>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>
</tbody>
</table>

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.
• Sort by terms.

**Core indexing step.**

<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>
</tbody>
</table>

...\

<table>
<thead>
<tr>
<th>Term</th>
<th>Doc #</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
</tbody>
</table>
• Multiple term entries in a single document are merged.
• Frequency information is added.
• The result is split into a *Dictionary* file and a *Postings* file.
• Where do we pay in storage?
Inverted index

Dictionary

Brutus

Calpurnia

Caesar

Postings lists

Sorted by docID (more later on why).
The index we just built

• How do we process a query?
  – Look up each query word in the dictionary
    • Retrieve postings lists
  – Merge
Looking up words in dictionary

Use an auxiliary data structure (e.g. hash table, search tree)
Query processing: AND

• Consider processing the query:
  
  **Brutus AND Caesar**
  
  – Locate *Brutus* in the Dictionary;
    • Retrieve its postings.
  
  – Locate *Caesar* in the Dictionary;
    • Retrieve its postings.
  
  – “Merge” the two postings:

```
  Brutus

  Caesar

  2  4  8  16  32  64  128
  1  2  3  5  8  13  21  34
```
Naive Merge

• For each DocID \(d1\) in list 1
  – For each DocID \(d2\) in list 2
    • If \(d1 = d2\), output the match

• For lists of lengths \(x\) and \(y\), this merge takes time proportional to \(xy\).
  – US: 5 billion hits, China: 1 billion hits
  – US AND China: 5,000,000,000,000,000,000,000 operations?
    • Not promising
Better merge

• Idea: Walk through the lists simultaneously
• Loop until you’re at the end of both lists:
  – If current DocIDs match, output the DocID
  – Step forward on the list w/lowest current DocID

For sorted lists of lengths x and y, a merge takes O(x+y) operations.
More general merges

• **Exercise**: Adapt the merge for the queries:

  - *Brutus AND NOT Caesar*
  - *Brutus OR NOT Caesar*

Can we still run through the merge in time $O(x+y)$?
What can we achieve?
Merging

What about an arbitrary Boolean formula?

\[(\text{Brutus OR Caesar}) \text{ AND NOT (Antony OR Cleopatra)}\]

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

• Can we do better?
Query optimization

• 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**
Query optimization example

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

Execute the query as (Caesar AND Brutus) AND Calpurnia.
Query processing exercises

• If the query is \textit{friends AND romans AND (NOT countrymen)}, how could we use the freq of \textit{countrymen}?

• Exercise: Extend the merge to an arbitrary Boolean query. Can we always guarantee execution in time linear in the total postings size?

• Hint: Begin with the case of a Boolean \textit{formula} query: in this, each query term appears only once in the query.
Outline

• Inverted Indices
  – How they enable fast IR
  – Tokenization & Linguistic issues
  – Handling phrase queries
Inverted index construction

Documents to be indexed.

Token stream.

What are these?

Modified tokens.

Inverted index.

Tokenizer

Linguistic modules

Indexer

friend

roman

countryman

Inverted index.

Friends, Romans, countrymen.

2 4

1 2

13 16

Sec. 1.2

What are these?

Documents to be indexed.
Tokenization

• **Input**: “Friends, Romans and Countrymen”

• **Output**: Tokens
  – *Friends*
  – *Romans*
  – *Countrymen*

• What tokens to emit?
Issues in Tokenization

- *Finland’s capital* →
  *Finland? Finlands? Finland’s?*

- *Hewlett-Packard* →
  *Hewlett* and *Packard* as two tokens?
  – What about *co-education*?

- *San Francisco*: one token or two? How do you decide it is one token?
Stop words

• Exclude from dictionary entirely the commonest words? Intuition:
  – Lose little semantic content *the, a, and, to, be*
  – Gain space (~30% of postings for top 30 wds)

• Not typically done (anymore)
  – Compression & optimization techniques (later) means cost for including stop words is low
  – You need them for:
    • Phrase queries: “King of Denmark,” “Flights to Orlando”
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*
  – *co-education* and *coeducation*

• Ideal: asymmetric expansion
  – Enter: *window* Search: *window, windows*
  – Enter: *windows* Search: *Windows, windows, window*
  – Enter: *Windows* Search: *Windows*

• More powerful; but less efficient, hard to define
Case folding

• Reduce all letters to lower case
  – exception: upper case in mid-sentence?
    • e.g., *Fed* vs. *fed*
      – “we Fed officials” != “we fed officials”
    • *US* vs *us*
  – But...users submit lowercase regardless of ‘correct’ capitalization
Thesauri and Soundex

• Handle synonyms and homonyms
  – Hand-constructed equivalence classes
    • e.g., car = automobile
    • your ≠ you’re

• Index such equivalences?
• Or expand query?
  – More later ...
Stemming

• Reduce terms to their “roots” before indexing
  – e.g., *automate(s), automatic, automation* all reduced to *automat*.

• Language dependent

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

• Most common stemmer for English
  – Available: http://tartarus.org/~martin/PorterStemmer/

• Rules applied sequentially in phases, e.g.:
  – sses $\rightarrow$ ss
  – ies $\rightarrow$ i
  – ational $\rightarrow$ ate
  – tional $\rightarrow$ tion
Challenges

• Sandy
• Sanded ➔ Sand ???
Outline

• Inverted Indices
  – How they enable fast IR
  – Tokenization & Linguistic issues
  – Handling phrase queries
Phrase Queries

Our existing inverted index can’t be used to process phrase queries...

Dictionary

Posting

Postings lists

Brutus
Calpurnia
Caesar

2 4 8 16 32 64
3 5 8 13 21 34
1 2 3 5 8 13 21 34
13 16
Solution: Positional indexes

• In the postings, store, for each term, entries of the form:

  <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, ...>

• Similar merge algorithm
  – But we now need to deal with more than just equality

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

• Extract inverted index entries for each distinct term: to, be, or, not.
• Merge their doc:position lists to enumerate all positions with “to be or not to be”.
  – to:
    • 2:1,17,74,222,551; 4:8,16,190,429,433; 7:13,23,191; ...
  – be:
    • 1:17,19; 4:17,191,291,430,434; 5:14,19,101; ...
• Same general method for proximity searches
Positional index size (1)

• 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>
Positional Index Size (2)

• Rules of thumb (for English-like languages):
  – A positional index is 2–4 times as large as a non-positional index
  – Positional index size 35–50% of volume of original text

• Worth the space
  – Quotes are the one advanced search operator that people actually use
  – Phrases also useful *implicitly* for ranking
Reminder

• Project Proposals Due next Monday
  – April 11, 11:59PM

• Sign up for project meetings April 12 and 13
  – Link to schedule to be sent via e-mail