Introduction to Information Retrieval

Introducing Information Retrieval and Web Search
Information Retrieval

- Information Retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

  - These days we frequently think first of web search, but there are many other cases:
    - E-mail search
    - Searching your laptop
    - Corporate knowledge bases
    - Legal information retrieval
Unstructured (text) vs. structured (database) data in the mid-nineties
Unstructured (text) vs. structured (database) data today
Basic assumptions of Information Retrieval

- **Collection**: A set of documents
  - Assume it is a static collection for the moment

- **Goal**: Retrieve documents with information that is relevant to the user’s information need and helps the user complete a task
The classic search model

User task

Info need

Query

Get rid of mice in a politically correct way

Info about removing mice without killing them

how trap mice alive

Search engine

Results

Collection

Query refinement

Misconception?

Misformulation?
How good are the retrieved docs?

- **Precision**: Fraction of retrieved docs that are relevant to the user’s information need
- **Recall**: Fraction of relevant docs in collection that are retrieved

- More precise definitions and measurements to follow later
Introduction to

Information Retrieval

Introducing Information Retrieval
and Web Search
Introduction to Information Retrieval

Term-document incidence matrices
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
Incidence vectors

- So we have a 0/1 vector for each term.
- To answer query: take the vectors for *Brutus*, *Caesar* and *Calpurnia* (complemented) \(\Rightarrow\) 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.
Introduction to Information Retrieval

Term-document incidence matrices
Introduction to Information Retrieval

The Inverted Index

The key data structure underlying modern IR
# 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?

<table>
<thead>
<tr>
<th>Term</th>
<th>docIDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brutus</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>173</td>
</tr>
<tr>
<td></td>
<td>174</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>docIDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caesar</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>132</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>docIDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calpurnia</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>101</td>
</tr>
</tbody>
</table>

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

Dictionary

*Brutus*

1 2 4 11 31 45 173 174

*Caesar*

1 2 4 5 6 16 57 132

*Calpurnia*

2 31 54 101

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
Inverted index construction

Documents to be indexed

Token stream

Tokenizer

Linguistic modules

Modified tokens

Indexer

Inverted index

More on these later.

Friends, Romans, countrymen.

Inverted index

Set of associated documents

friend

roman

countryman

2

4

1

2

13

16

1

2

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.
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.

<table>
<thead>
<tr>
<th>Term</th>
<th>docID</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>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>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>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>the</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>1</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>term</th>
<th>doc. freq.</th>
<th>postings lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>be</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
<td>→ 1 → 2</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
<td>→ 1 → 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>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>it</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>let</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>noble</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>so</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td>→ 1 → 2</td>
</tr>
<tr>
<td>told</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>you</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
<td>→ 1 → 2</td>
</tr>
<tr>
<td>with</td>
<td>1</td>
<td>→ 2</td>
</tr>
</tbody>
</table>
Where do we pay in storage?

<table>
<thead>
<tr>
<th>term</th>
<th>doc. freq</th>
<th>postings lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>1</td>
<td>2</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>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>2</td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>1</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>1</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>2</td>
</tr>
<tr>
<td>so</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>told</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>with</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Lists of docIDs

IR system implementation
- How do we index efficiently?
- How much storage do we need?
Introduction to Information Retrieval

The Inverted Index
The key data structure underlying modern IR
Introduction to Information Retrieval

Query processing with an inverted index
The index we just built

- How do we process a query?
  - Later - what kinds of queries can we process?
Query processing: AND

- Consider processing the query:
  \[ \text{Brutus AND Caesar} \]
  - Locate \textbf{Brutus} in the Dictionary;
    - Retrieve its postings.
  - Locate \textbf{Caesar} in the Dictionary;
    - Retrieve its postings.
  - “Merge” the two postings (intersect the document sets):

\[
\begin{align*}
\text{Brutus} & : 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow 128 \\
\text{Caesar} & : 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 13 \rightarrow 21 \rightarrow 34
\end{align*}
\]
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.
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)

\[
\text{INTERSECT}(p_1, p_2)
\]

1. \( \text{answer} \leftarrow \langle \rangle \)
2. while \( p_1 \neq \text{NIL} \) and \( p_2 \neq \text{NIL} \)
3. do if \( \text{docID}(p_1) = \text{docID}(p_2) \)
   then \( \text{ADD}(\text{answer}, \text{docID}(p_1)) \)
4. \( p_1 \leftarrow \text{next}(p_1) \)
5. \( p_2 \leftarrow \text{next}(p_2) \)
6. else if \( \text{docID}(p_1) < \text{docID}(p_2) \)
7. then \( p_1 \leftarrow \text{next}(p_1) \)
8. else \( p_2 \leftarrow \text{next}(p_2) \)
9. return \( \text{answer} \)
Introduction to Information Retrieval

Query processing with an inverted index
Introduction to Information Retrieval

Phrase queries and positional indexes
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
  - Many more queries are implicit phrase queries
- 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
  - *friends romans*
  - *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
- *stanford university palo alto* can be broken into the Boolean query on biwords:
  
  \[ \text{stanford university AND university palo AND palo alto} \]

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

Can have false positives!
Extended biwords

- Parse the indexed text and perform part-of-speech-tagging (POST).
- Bucket the terms into (say) Nouns (N) and articles/prepositions (X).
- Call any string of terms of the form NX*N an extended biword.
  - Each such extended biword is now made a term in the dictionary.
- Example: catcher in the rye
  
  N    X    X    N

- Query processing: parse it into N’s and X’s
  - Segment query into enhanced biwords
  - Look up in index: catcher rye
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:

  
  \[<\text{term}, \text{number of docs containing term};
  \text{doc1}: \text{position1, position2 ... } ;
  \text{doc2}: \text{position1, position2 ... } ;
  \text{etc.}>\]
Positional index example

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

- For phrase queries, we use a merge algorithm recursively at the document level
- 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
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
  - See Figure 2.12 of *IIR*
Positional index size

- A positional index expands postings storage *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
Introduction to Information Retrieval

Phrase queries and positional indexes