Intro to Text Processing
Lecture 9

Behrang Mohit

Some ideas and slides in this presentation are borrowed from Chris Manning and Dan Jurafsky.

15383: txt proc

Review

• Bag of word view
  – Document classification
• Information Extraction
  – Named entity recognition
• Some linguistics
  – POS tagging
  – Lexical semantics
    • WordNet

15383: txt proc
Today: 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

![Bar chart showing data volume and market cap comparison between unstructured and structured data](chart.png)
Unstructured (text) vs. structured (database) data today

The IR Terminology

- Document
- Collection
- Term
- Query
- Similarity
Bag of words model

- Modeling assumption: documents are bags of words
  - Word order doesn’t matter

The IR problem

- Access to unstructured information
- Which Shakespeare’s play has words *Caesar* and *Brutus* but not *Calpurnia*?

- `grep` all Shakespeare's plays?
  - `not Calpurnia`
  - Other operations (*romans* near *countrymen*)
  - Ranked retrieval
Term-document Matrix

<table>
<thead>
<tr>
<th></th>
<th>Anthony &amp; Cleopatra</th>
<th>Julius Caesar</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anthony</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</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>
</tr>
<tr>
<td>Cleopatra</td>
<td>1</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>
</tr>
<tr>
<td>Worser</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Brutus and Caesar, NOT Calpurnia

Vector representation and retrieval

- A vector of binaries for each term
  - Brutus: 11100
  - Calpurnia: 01000 \( \rightarrow \) NOT Calpurnia: 10111
- Do a bitwise AND
  
  \[
  \begin{align*}
  1 & 1 & 1 & 0 & 0 \\
  1 & 1 & 1 & 1 & 1 \\
  1 & 0 & 1 & 1 & 1 \\
  1 & 0 & 1 & 1 & 0 \\
\end{align*}
  \]

  \(- 11100 \text{ AND } 11111 \text{ AND } 10111 = 10100 \)
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.

Not the best representation

• 1 Million documents ~ 6GB data
  – 500K token types ➔ 500k x 1M = 500 GB matrix
    • too much space for storing all zeros

• The matrix has no more than one billion 1s.

• Need a better representation
  – Only record the 1s.
Not the best representation

- 1 Million documents ~ 6GB data
  - 500K token types \(\Rightarrow\) 500k x 1M = 500 GB matrix
    - Too much space for storing all zeros

- The matrix has no more than one billion 1s.

- Need a better representation
  - Only record the 1s.

Inverted Index

- Example: books
Data structure for indexing

• For each term $t$, we store a list of the document identifier which has $t$.

<table>
<thead>
<tr>
<th>Term</th>
<th>List of Document Ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brutus</td>
<td>3 5 6 10 23 57</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>2 5 9 17 19 23 46</td>
</tr>
<tr>
<td>Caesar</td>
<td>12 19 23 29</td>
</tr>
</tbody>
</table>

• Data Structure? Array?
  – What if a new document with Caesar is added?

Data structure for indexing

• Dynamic space allocation
• Easy insertion

*Dictionary*

Brutus

3 → 5 → 6 → 10

Cesar

2 → 5 → 7 → 10

*Posting List*: sorted by docid
Indexing pipeline

Text to be indexed

Tokenized text

Shallow linguistic proc.

Inverted index

15383: txt proc

Constructing an index: Pairing

• Text ➔ (token, document id)
• Tokens are lower-cased

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
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

Constructing an index: Sorting

- Sort the pairs alphabetically
Constructing an index: Counting

- Merge the same terms
  - Collect & add term frequency

- Intuition:
  Frequency contributes to the relevance.

Constructing the index: final
Storage cost

- Terms
- Links

Processing an AND query

- Brutus AND Cesar
For each word:
  Locate in dictionary and retrieve its posting

Brutus

Cesar

11/20/13
Processing an AND query

• Brutus **AND** Cesar

For each word:
  Locate in dictionary and retrieve its posting

Merge the postings:
• Walk through the postings concurrently and find the overlaps
  — Linear time

```
Brutus  Cesar
  3     2
  2     5
  5     5
  6     7
  10    10
```

Boolean queries: Exact match

• 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
  – Perhaps the simplest model to build an IR system on

• Primary commercial retrieval tool for 3 decades.
• Many search systems you still use are Boolean:
  – Email, library catalog, Mac OS X Spotlight
Boolean queries: Exact match

- 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
  - Perhaps the simplest model to build an IR system on
- Primary commercial retrieval tool for 3 decades.
- Many search systems you still use are Boolean:
  - Email, library catalog, Mac OS X Spotlight


- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992; new federated search added 2010)
- Tens of terabytes of data; ~700,000 users
  - Majority of users still use boolean queries
  - Example query:
    - What is the statute of limitations in cases involving the federal tort claims act?
    - LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM
    * /3 = within 3 words, /S = in same sentence
Example: WestLaw  
http://www.westlaw.com/

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992; new federated search added 2010)
- Tens of terabytes of data; ~700,000 users
- Majority of users still use boolean queries
- Example query:
  - What is the statute of limitations in cases involving the federal tort claims act?
  - LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM
    - /3 = within 3 words, /S = in same sentence

Example: WestLaw  
http://www.westlaw.com/

- Many professional searchers still like Boolean search
  - You know exactly what you are getting
- But that doesn’t mean it actually works better....
Query optimization

• What is the best order for query processing?
• Consider a query that is an **AND** of \( n \) terms.
• For each of the \( n \) terms, get its postings, then **AND** them together.

```
Brutus    2 4 8 16 32 64 128
Caesar    1 2 3 5 8 16 21 34
Calpurnia 13 16
```

Query: **Brutus** **AND** **Calpurnia** **AND** **Caesar**

---

Query optimization example

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

```
Brutus    2 4 8 16 32 64 128
Caesar    1 2 3 5 8 16 21 34
Calpurnia 13 16
```

This is why we kept document freq. in dictionary

Execute the query as (**Calpurnia** **AND** **Brutus**) **AND** **Caesar**.
More general optimization

- e.g., \((\text{madding OR crowd}) \text{ AND } (\text{ignoble OR strife})\)
- Get doc. freq.’s for all terms.
- Estimate the size of each OR by the sum of its doc. freq.’s (conservative).
- Process in increasing order of OR sizes.

Exercise

- Recommend a query processing order for \((\text{tangerine OR trees}) \text{ AND } (\text{marmalade OR skies}) \text{ AND } (\text{kaleidoscope OR eyes})\)
- Which two terms should we process first?

<table>
<thead>
<tr>
<th>Term</th>
<th>Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>eyes</td>
<td>213312</td>
</tr>
<tr>
<td>kaleidoscope</td>
<td>87009</td>
</tr>
<tr>
<td>marmalade</td>
<td>107913</td>
</tr>
<tr>
<td>skies</td>
<td>271658</td>
</tr>
<tr>
<td>tangerine</td>
<td>46653</td>
</tr>
<tr>
<td>trees</td>
<td>316812</td>
</tr>
</tbody>
</table>
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
  -- 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

- 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, ...>

• 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; ...
Need for ranking

• Boolean model: Which articles are relevant?

• Reality: Millions of relevant documents
  – Which one is more relevant?
    • How similar is one document to another one?

• Need to rank documents
  – Estimate the similarity between the query and documents.