Lecture 2: The Term-Document Matrix and Boolean Retrieval

Trevor Cohn (tcohn@unimelb.edu.au)

COMP90042, 2015, Semester 1
What we’ll learn today

- Concepts of the *Term-Document Matrix* and *inverted index*
- How this is used to query documents for specific terms
- Allowing for binary operators AND/OR/NOT
- Efficient query methods
Collection representation

- Each document is modelled as a “bag of words”
- I.e., a list of terms it contains and the count of each term, but discarding the word order
- The whole collection could be modelled as a “list of bag of words”
- ...but that fails to capture commonality of terms between documents (which is the core to the text analysis tools we’ll be looking at)
A simple solution, ‘grepping’

Imagine we wish to search through the texts of Project Gutenberg for **Pangolin**

- Can simply use *grep* which performs a linear scan over the text searching for a match
- Reasonable for small corpora, but linear time is much too slow for large collections
- And how to handle more complex queries?
  - **Pangolin** AND ant-eater
  - **Pangolin** OR ant-eater
  - **Pangolin** NEAR ant-eater
  - Pang*in
- To speed up queries, we first pre-compute an index
The term-document matrix (TDM)

- A central representation in IR is a “term–document matrix” or TDM.
- Recall that a “matrix” is a two-dimensional array, with rows and columns.
- In the TDM, rows represent documents, columns represent terms (in the collection vocabulary).
- Cell values are binary term indicators, frequency counts, or, more generally, a “score” attached to a term for a document.
Example TDM

<table>
<thead>
<tr>
<th>doc1</th>
<th>Two for tea and tea for two</th>
</tr>
</thead>
<tbody>
<tr>
<td>doc2</td>
<td>Tea for me and tea for you</td>
</tr>
<tr>
<td>doc3</td>
<td>You for me and me for you</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>two</th>
<th>tea</th>
<th>me</th>
<th>you</th>
</tr>
</thead>
<tbody>
<tr>
<td>doc1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>doc2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>doc3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Let’s reduce the matrix to binary, 0s and 1s, to reflect word inclusion (but not frequency)

And transpose the matrix, so now our rows are terms and columns are documents

<table>
<thead>
<tr>
<th></th>
<th>doc1</th>
<th>doc2</th>
<th>doc3</th>
</tr>
</thead>
<tbody>
<tr>
<td>two</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tea</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>me</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>you</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Each row is called the **posting list** (or vector) for the term

Queries will operate over the postings
Querying

- For the query `tea AND me`
- Take the bit representations
  
  \[
  \text{tea} = 110 \quad \text{me} = 011
  \]
- Perform bitwise and, \(\land\), resulting in \(010\)
- Therefore document 2 is the only match

To support:
- disjunction, simply use bitwise or, \(\lor\)
- negation, use bitwise complement, \(\hat{\ }\)

E.g., `tea AND ((NOT me) OR two)`
\[
\text{tea} = 110 \quad \text{AND} \quad (\text{not me} = 100 \quad \text{OR} \quad \text{two} = 100) = 100
\]
Efficiency of binary TDM

- For any realistic sized corpus, the TDM will be too large to store
- E.g., 1M documents and 500K unique terms, leads to 500B matrix cells
- But the matrix is extremely sparse
  - most documents only contain a handful of terms
  - the majority of terms only occur in a handful of documents
- Therefore need to use a **sparse encoding**
Inverted Index

- Store only the 1s, aka **postings** using linked list of **docIDs** for every term

![Diagram of inverted index]

- This data structure is known as an inverted index
- Posting lists are sorted, e.g., by a document identifier
- Postings use variable length sequence, e.g., a linked list in memory or contiguous sequence on disk
Querying an Inverted Index

- Querying involves operations on posting lists
- E.g., conjunctive queries (AND) require finding the set of common entries

```
<table>
<thead>
<tr>
<th>tea 2</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>me  2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Result 2

- Merging done iteratively, scanning over the lists in order and advancing over the smaller entry
- When entries identical, record the result and advance both lists
- Complexity $O(x + y)$ where $x$ and $y$ are the lengths of the posting lists
- Query cost can depend on query order for long queries
Process of Index Construction

Step 1: tokenize and collect term occurrences.

Step 2: sort by term.

Step 3: collate into posting lists and doc. freq.

Figure from Manning et al, IIR.
Storage costs

- Storage requirements
  - terms and counts, $|T|$ entries
  - pointers to postings, $|T|$ instances
  - lists of docIDs, $|T| \times s$

- Zipf’s law: the majority of terms have few occurrences, $s$ is typically small

- Worse case for stop words $s \approx |D|$

- Long posting lists also leads to poor search time
Faster intersection with skip lists

Skip lists can accelerate intersections of posting lists

- some nodes have **skip pointers** further into the list
- when comparing two lists, can advance several elements in one go when end point of skip list is less than current item on other list
After matching 8 – 8
Consider 16 – 41
But also have skip pointer, so check 28 – 41
Still smaller than 41, so take the ‘skip’ step
Skip list considerations

Pay a space cost for these skip pointers, and time cost for skip comparisons

- Where to place skips and how many?
- Short skips are more likely to be followed; but fewer items end up skipped each time
- Also incurs a higher storage cost and entails more skip comparisons
- Reverse argument applies for longer skips

Rule of thumb

- $\sqrt{P}$ uniformly spaced works well in general, see Zobel & Moffat, 2006.
- but this ignores the distribution over terms
- and becomes difficult if $P$ changes
Looking back and forward

- Concept of the term-document matrix and inverted index
- Efficient data structures and query methods for boolean queries
- Skip lists as a means of speeding up intersection operations during queries
Looking back and forward

Forward

▶ In the next lecture, we will look at how to perform queries over multiword phrases
▶ And efficient search for queries with wildcards
Further reading

- Chapter 1, “Boolean Retrieval” of Manning, Raghavan, and Schutze, *Introduction to Information Retrieval*
- Section 2.3, “Faster posting list intersection via skip pointers” of Manning, Raghavan, and Schutze, *Introduction to Information Retrieval*