IR Models: The Vector Space Model
Boolean Model Disadvantages

- Similarity function is boolean
  - Exact-match only, no partial matches
  - Retrieved documents not ranked
- All terms are equally important
  - Boolean operator usage has much more influence than a critical word
- Query language is expressive but complicated
The Vector Space Model

- Documents and queries are both vectors
  \[ \vec{d}_i = (w_{i,1}, w_{i,2} \ldots w_{i,t}) \]
- each \( w_{i,j} \) is a weight for term \( j \) in document \( i \)
- "bag-of-words representation"
- Similarity of a document vector to a query vector = cosine of the angle between them
Cosine Similarity Measure

\[ sim(d_i, q) = \cos \theta \]

\[ (x \cdot y = |x||y|\cos \theta) \]

\[ \frac{d_i \cdot q}{|d_i||q|} = \frac{\sum_j w_{i,j} \times w_{q,j}}{\sqrt{\sum_j w_{i,j}^2} \sqrt{\sum_j w_{q,j}^2}} \]

- Cosine is a normalized dot product
- Documents ranked by decreasing cosine value
  - \( sim(d,q) = 1 \) when \( d = q \)
  - \( sim(d,q) = 0 \) when \( d \) and \( q \) share no terms
Term Weighting

- Higher weight = greater impact on cosine
- Want to give more weight to the more "important" or useful terms
- What is an important term?
  - If we see it in a query, then its presence in a document means that the document is relevant to the query.
  - How can we model this?
Clustering Analogy

- Documents are collection of $C$ objects
- Query is a vague description of a subset $A$ of $C$
- IR problem: partition $C$ into $A$ and $\sim A$
- We want to determine
  - which object features best describe members of $A$
  - which object features best differentiate $A$ from $\sim A$
- For documents,
  - frequency of a term in a document
  - frequency of a term across the collection
Term Frequency (tf) factor

- How well does a term describe its document?
  - if a term $t$ appears often in a document, then a query containing $t$ should retrieve that document
  - frequent (non-stop) words are thematic
    - flow, boundary, pressure, layer, mach

$$tf_{i,j} = \frac{f_{i,j}}{\max_j f_{i,j}}$$

$$tf_{i,j} = 1 + \log f_{i,j}$$

$$tf_{i,j} = K + \frac{(1-K) \times f_{i,j}}{\max_j f_{i,j}}$$
Inverse Document Frequency (idf) factor

- A term’s scarcity across the collection is a measure of its importance.
  - Zipf’s law: term frequency $\approx 1/$rank
  - Importance is inversely proportional to frequency of occurrence.

$\text{idf}_t = \log(1 + \frac{N}{n_t})$

$\text{idf}_t = \log\left(\frac{N - n_t}{n_t}\right)$

$N = \# \text{ documents in coll}$

$n_t = \# \text{ documents containing term } t$
tf-idf weighting

- A weighting scheme where
  \[ w_{d,t} = tf_{d,t} \times idf_t \]
  is called a \textit{tf-idf scheme}
- tf-idf weighting is the most common term weighting approach for VSM retrieval
- There are many variations...
tf-idf Monotonicity

- "A term that appears in many documents should *not* be regarded as *more important* than one that appears in few documents."
- "A document with many occurrences of a term should *not* be regarded as *less important* than a document with few occurrences of the term."
Length Normalization

\[ \frac{d_i \cdot q}{|d_i| \cdot |q|} \]

- Why normalize by document length?
- Long documents have
  - **Higher term frequencies**: the same term appears more often
  - **More terms**: increases the number of matches between a document and a query
- Long documents are more likely to be retrieved
- The "cosine normalization" lessens the impact of long documents
## VSM Example

<table>
<thead>
<tr>
<th>d</th>
<th>col</th>
<th>day</th>
<th>eat</th>
<th>hot</th>
<th>lot</th>
<th>nin</th>
<th>old</th>
<th>pea</th>
<th>por</th>
<th>pot</th>
<th>W_d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td></td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.7</td>
<td>1.7</td>
<td></td>
<td>2.78</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.73</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td>1.73</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.7</td>
<td></td>
<td></td>
<td>2.21</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.7</td>
<td>1.7</td>
<td></td>
<td>2.40</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.41</td>
</tr>
<tr>
<td></td>
<td>idf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>1.39</td>
<td>1.95</td>
<td>1.95</td>
<td>1.39</td>
<td>1.95</td>
<td>1.95</td>
<td>1.95</td>
<td>1.1</td>
<td>1.1</td>
<td>1.39</td>
</tr>
</tbody>
</table>

- q1 = *eat*
- q2 = *porridge*
- q3 = *hot porridge*
- q4 = *eat nine day old porridge*
Vector Space Model

Advantages

- Ranked retrieval
- Terms are weighted by importance
- Partial matches

Disadvantages

- Assumes terms are independent
- Weighting is intuitive, but not very formal
Implementing VSM

$$sim(q, d) = \frac{1}{W_q W_d} \sum_t w_{q,t} \times w_{d,t}, \quad W_d = \sqrt{\sum_t w_{d,t}^2}$$

- Need within-document frequencies in the inverted list
- $W_q$ is the same for all documents
- $w_{q,t}$ and $w_{d,t}$ can be accumulated as we process the inverted lists
- $W_d$ can be precomputed
Cosine algorithm

1. \( A = \{\} \) (set of accumulators for documents)
2. For each query term \( t \)
   - Get term, \( f_t \), and address of \( I_t \) from lexicon
   - set \( \text{idf}_t = \log(1 + N/f_t) \)
   - Read inverted list \( I_t \)
   - For each \( <d, f_{d,t}> \) in \( I_t \)
     - If \( A_d \notin A \), initialize \( A_d \) to 0 and add it to \( A \)
     - \( A_d = A_d + (1 + \log(f_{d,t})) \times \text{idf}_t \)
3. For each \( A_d \) in \( A \), \( A_d = A_d/W_d \)
4. Fetch and return top \( r \) documents to user
Managing Accumulators

- How to store accumulators?
  - static array, 1 per document
  - grow as needed with a hash table
- How many accumulators?
  - can impose a fixed limit
  - quit processing $l_t$'s after limit reached
  - continue processing, but add no new $A_d$'s
Managing Accumulators (2)

- To make this work, we want to process the query terms in order of decreasing $idf_t$
- Also want to process $I_t$ in decreasing $tf_{d,t}$ order
  - sort $I_t$ when we read it in
  - or, store inverted lists in $f_{d,t}$-sorted order

\[
\langle 5; (1,2) (2,2) (3,5) (4,1) (5,2) \rangle \quad \langle f_t; (d, f_{d,t}) \rangle \ldots
\]

\[
\langle 5; (3,5) (1,2) (2,2) (5,2) (4,1) \rangle \quad \text{sorted by } f_{d,t}
\]

\[
\langle 5; (5, 1:3) (2, 3:1,2,5) (1, 1:4) \rangle \quad \langle f_t; (f_{d,t}, c:d,\ldots)\rangle \ldots
\]
- This can actually compress better, but makes Boolean queries harder to process
Getting the top documents

- Naïve: sort the accumulator set at end
- Or, use a heap and pull top $r$ documents
  - much faster if $r << N$
- Or better yet, as accumulators are processed to add the length norm ($W_d$):
  - make first $r$ accumulators into a min-heap
  - for each next accumulator
    - if $A_d < \text{heap-min}$, just drop it
    - if $A_d > \text{heap-min}$, drop the heap-min, and put $A_d$ in