Lecture 5: Information Retrieval using the Vector Space Model

Trevor Cohn (tcohn@unimelb.edu.au)
Slide credits: William Webber

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

- How to take a user query and return a ranked list of results
- How to implement this operation in a reasonably efficient way
- How to normalise for document length
Reviewing: document similarity in VSM

- Document is BOW
- Project into term space as vector, with dimension lengths given by TF*IDF
- Calculate document similarity as cosine of angle between their vectors
- Implement as dot product on unit-length vectors

Same process can be used to *rank* documents by decreasing similarity to given document.
Query processing in VSM

- Treat the query as a (short) (pseudo-)document
- Calculate (VSM cosine) similarity between query pseudo-document and each document in collection
- Rank documents by decreasing similarity with query
- Return to user in rank order (generally only top results initially)
Example

Corpus:

<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>

Query: **tea me**

<table>
<thead>
<tr>
<th></th>
<th>two</th>
<th>tea</th>
<th>me</th>
<th>you</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

Corpus:

<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>0.707</td>
<td>0.707</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>doc2</td>
<td>0</td>
<td>0.894</td>
<td>0.447</td>
<td>0.447</td>
</tr>
<tr>
<td>doc3</td>
<td>0</td>
<td>0</td>
<td>0.707</td>
<td>0.707</td>
</tr>
</tbody>
</table>

Query: **tea me**

<table>
<thead>
<tr>
<th></th>
<th>two</th>
<th>tea</th>
<th>me</th>
<th>you</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>0</td>
<td>0.707</td>
<td>0.707</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

\[
\cos(\text{doc1}, q) = (0.707, 0.707, 0, 0) \cdot (0, 0.707, 0.707, 0) \\
= 0.5 \\
\cos(\text{doc2}, q) = (0, 0.894, 0.447, 0.447) \cdot (0, 0.707, 0.707, 0) \\
= 0.945 \\
\cos(\text{doc3}, q) = (0, 0, 0.707, 0.707) \cdot (0, 0.707, 0.707, 0) \\
= 0.5
\]

So doc2 is the best match, followed by docs 1 and 3, which are tied.
Cosine similarity

- Many elements on the example vectors were 0, and thus did not contribute to the cosine
- True for real settings with large vocabularies
- Enumerating all the documents is highly inefficient
- Can we devise a way to find the most similar documents efficiently?
- ...using the inverted index?
Imagine that for every similarity computation, we recalculate unit-length TF*IDF vectors for all documents.

- recall these are simple expressions involving the raw term frequency $f_{d,t}$ and the document frequency $df_t$

Since these do not change from query to query, save processing by precalculating and store results in an index.

But we still need to iterate through all documents to rank by similarity.

This an $O(|D|)$ operation.
Term-wise processing

- In document similarity, only terms occurring in both documents contribute to cosine score (remember the dot-product)
- In query processing by pseudo-document model, therefore, only documents that contain query terms need to be considered (which makes intuitive sense)
- Complexity reduced to $O(\sum_t df_t)$
  - Note that most frequent term dominates.
  - Very good reason to drop stop-words!
- Need an index that supports quickly finding which documents a term occurs in
Recap: Inverted index

Index designed to support query processing:

- Keys are terms
- Values are lists of $\langle d, w_{t,d} \rangle$ pairs
- Each $\langle d, w_{t,d} \rangle$ pair called a *posting*
- List of these called a *postings list*

<table>
<thead>
<tr>
<th>Term</th>
<th>Postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>tea</td>
<td>→ 1:1.4 ; 3:1.0 ; 6:1.7 ; ...</td>
</tr>
<tr>
<td>two</td>
<td>→ 2:2.3 ; 3:1.0 ; 4:1.7 ; ...</td>
</tr>
<tr>
<td>me</td>
<td>→ 1:1.0 ; 2:1.4 ; ...</td>
</tr>
</tbody>
</table>

Note the inclusion of $w$ weightings here, not longer binary as in early lectures.
Efficient encoding

High cost of storing the inverted index, seek to minimise its footprint

- So it can be stored in memory (much faster than disk)
- Or on disk, to minimise the number of disk accesses

Must consider the size implications of data structure choices

- Real valued numbers (floats) much larger than integer counts
- And much harder to compress
- Although it’s convenient to store precomputed and document-normalised weightings (e.g., TF*IDF)
- Often more practical to store the components, e.g.,
  - raw term frequencies in the postings lists
  - IDF values stored for each term in the dictionary
  - document length normalisation values stored in a separate list
More efficient index for example

- Inverted index storing mostly integer counts

<table>
<thead>
<tr>
<th>Term</th>
<th>IDF</th>
<th>Postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>tea</td>
<td>1.9</td>
<td>1:3 ; 3:1 ; 6:2 ; ...</td>
</tr>
<tr>
<td>two</td>
<td>0.3</td>
<td>2:4 ; 3:1 ; 4:2 ; ...</td>
</tr>
<tr>
<td>me</td>
<td>0.8</td>
<td>1:1 ; 2:2 ; ...</td>
</tr>
</tbody>
</table>

- Real valued document lengths

<table>
<thead>
<tr>
<th>DocId</th>
<th>$w_{.,d}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.3</td>
</tr>
<tr>
<td>2</td>
<td>3.4</td>
</tr>
<tr>
<td>3</td>
<td>1.7</td>
</tr>
<tr>
<td>4</td>
<td>42.8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Incurs a cost at query time, but often outweighed by the better use of faster media
Query processing on inverted index

Assuming normalised TF*IDF weights:

Set accumulator $a_d \leftarrow 0$ for all documents $d$

for all terms $t$ in query do

Load postings list for $t$

for all postings $\langle d, w_{t,d} \rangle$ in list do

\[ a_d \leftarrow a_d + w_{t,d} \]

end for

end for

Sort documents by decreasing $a_d$

return sorted results to user

Inefficient use of space!
Query processing on inverted index

Storing integer term-frequencies:

Set accumulator \( a_d \leftarrow 0 \) for all documents \( d \)

\[ \text{for all terms } t \text{ in query do} \]

\[ \quad \text{Load postings list for } t \text{ and } \text{idf}_t = \log \frac{N}{f_t} \]

\[ \quad \text{for all postings } \langle d, f_{t,d} \rangle \text{ in list do} \]

\[ \quad \quad a_d \leftarrow a_d + f_{t,d} \times \text{idf}_t \]

\[ \quad \text{end for} \]

\[ \text{end for} \]

Load document length array, \( L \)

Normalise by document lengths \( a_d \leftarrow \frac{a_d}{L_d} \)

Sort documents by decreasing \( a_d \)

\text{return} sorted results to user

A little extra computation in the inner loop, but supports more compact storage.
Is this cosine?

- Computed for each document

\[ a_d = \frac{\sum_{t \in q} w_{t,d}}{\sqrt{\sum_{t \in d} w_{t,d}^2}} \]

- However cosine is defined as

\[ \cos(d, q) = \frac{\sum_t w_{t,q} w_{t,d}}{\sqrt{\sum_{t \in q} w_{t,q}^2 \sum_{t \in d} w_{t,d}^2}} \]

- what happened to the other normalisation term and \( w_{t,q} \)?

- term assumed to occur once in the query, i.e., \( w_{t,q} = 1 \) for \( t \in q \)

- query length normalisation doesn't matter when comparing a fixed query with several documents (why?)
Tweaking the formula

Several different choices must be made about weighting the formula
  ▶ include IDF, and its definition
  ▶ raw term-frequency, $f_{d,t}$ versus $\log(1 + f_{t,q})$
  ▶ whether to normalise document vectors

Unit-length normalization of query doesn’t matter

Many of formula component choices made here are heuristic
  ▶ Zobel and Moffat, “Exploring the Similarity Space” (1998) identify $(8 \times 9 \times 2 \times 6 \times 14) = 12096$ possible different combinations of choices

Once can try different variants to improve effectiveness (We’ll talk in a later lecture about how to test success)
Alternative document length normalization

- To date, normalized document vectors to unit length
- But is this correct?
  - Very short documents will get high scores for term occurrences
  - Long documents may cover many topics, satisfy many queries

Empirical adjustment
Assume that we have:
- Large number of queries
- Judgments of which documents are relevant to which queries

Then we can compare:
- Probability of document being retrieved given length
- Probability of document being relevant given length

and adjust if these two probabilities are out of line
Probability retrieved v. relevant given length

- Look at mean empirical relation

Probability retrieved v. relevant given length

- Look at mean empirical relation
- Simplify and identify “pivot”. Lengths greater than pivot point should be boosted; less, decreased

Probability retrieved v. relevant given length

- Look at mean empirical relation
- Simplify and identify “pivot”. Lengths greater than pivot point should be boosted; less, decreased
- Linearly approximate to “slope”

Pivoted document length normalization

- $w$: weight of term in document (e.g. TF*IDF)
- $n_u$: original normalization (e.g. unit-length normalization by length of document vector)
- $p$: pivot point for pivot normalization
- $s$: slope of pivot normalization
- $w_p$: pivot-normalized weight of term

$$w_p = \frac{w}{(1.0 - s) \cdot p + s \cdot n_u}$$

- Various approximations and factors (see Singhal et al.)
- No longer calculating cosine distance, but pseudo-cosine distance!
- Requires dataset to tune on, and will be specific to that dataset
- Gives significant improvement in effectiveness
Vector space cf. Boolean querying

How does the vector space model compare to boolean IR?

**Scope** Assigning matching scores to a larger set of documents, and returning ranked lists

**Type** Not a conjunctive nor a disjunctive query – documents may omit some query terms, but more matches lead to higher scores

**Index construction** is similar, but record TF*IDF values rather than boolean indicators

**Querying** is quite different, based on accumulators, with different computational complexity
Looking back and forward

- Queries can be processed in VSM by treating query as (pseudo-)document
- Inverted index supports efficient query processing
- Various tweaks to VSM formulae possible, of which pivoted document length normalization empirically the most effective
Looking back and forward

Forward

- How to store dictionary, postings and documents most efficiently?
- Compression techniques for best use of space and time
- In later lecture, will look at evaluation of IR methods, for selecting methods and tuning parameters
- Then we will look at probabilistic methods, that are often more theoretically grounded, requiring fewer heuristics
Further reading

- Chapter 6, §6.3 onwards, and §7.1 of Manning, Raghavan, and Schutze, *Introduction to Information Retrieval*

- Justin Zobel and Alistair Moffat, “Inverted files for text search engines” ACM Computing Surveys, 2006 (authoritative survey paper on inverted indexes)

- Singhal, Buckley, and Mitra, “Pivoted document length normalization”, SIGIR, 1996
  http://singhal.info/pivoted-dln.pdf