Information Retrieval

Lecture 5 - The vector space model

Seminar für Sprachwissenschaft
International Studies in Computational Linguistics
Wintersemester 2007

Introduction

◮ Boolean model: all documents matching the query are retrieved
◮ The matching is binary: yes or no
◮ Extreme cases: the list of retrieved documents can be empty, or huge
◮ A ranking of the documents matching a query is needed
◮ A score is computed for each pair (query, document)

Overview

Term weighting

Vector space model

Improving scoring and ranking

Conclusion

Term weighting
Term weighting

- Evaluation of how important a term is with respect to a document
- First idea: the more important a term is, the more often it appears → term frequency

\[ tf_{t,d} = \sum_{x \in d} f_t(x) \text{ where } f_t(x) = \begin{cases} 1 & \text{if } x = t \\ 0 & \text{otherwise} \end{cases} \]

- NB1: the order of terms within a doc is ignored
- NB2: are all words equally important? What about stop-lists?

Term weighting (continued)

- Terms occurring very often in the collection are not relevant for distinguishing among the documents
- A relevance measure cannot only take term frequency into account
- Idea: reducing the relevance (weight) of a term using a factor growing with the collection frequency
- Collection frequency versus document frequency?

<table>
<thead>
<tr>
<th>Term</th>
<th>( cf_t )</th>
<th>( df_t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>try</td>
<td>10422</td>
<td>8760</td>
</tr>
<tr>
<td>insurance</td>
<td>10440</td>
<td>3997</td>
</tr>
</tbody>
</table>

Inverse Document Frequency

- inverse document frequency of a term \( t \):

\[ idf_t = \log \frac{N}{df_t} \text{ with } N = \text{collection size} \]

- NB: rare terms have high \( idf_t \), contrary to frequent terms
- Example (Reuters collection, from Manning et al.):

<table>
<thead>
<tr>
<th>Term</th>
<th>( df_t )</th>
<th>( idf_t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>car</td>
<td>10165</td>
<td>1.65</td>
</tr>
<tr>
<td>auto</td>
<td>6723</td>
<td>2.08</td>
</tr>
<tr>
<td>insurance</td>
<td>19241</td>
<td>1.62</td>
</tr>
<tr>
<td>beat</td>
<td>25235</td>
<td>1.5</td>
</tr>
</tbody>
</table>

tf-idf weighting

- The weight of a term is computed using both \( tf \) and \( idf \):

\[ w(t, d) = tf_{t,d} \times idf_t \text{ called } tf-idf_{t,d} \]

- \( w(t, d) \) is:
  1. high when \( t \) occurs many times in a small set of documents
  2. low when \( t \) occurs fewer times in a document, or when it occurs in many documents
  3. very low when \( t \) occurs in almost every document
- Score of a document with respect to a query:

\[ \text{score}(q, d) = \sum_{t \in q} w(t, d) \]
Vector space model

Each term $t$ of the dictionary is considered as a dimension.

A document $d$ can be represented by the weight of each dictionary term:

$$V(d) = (w(t_1, d), w(t_2, d), \ldots, w(t_n, d))$$

Question: does this representation allow to compute the similarity between documents?

Similarity between vectors?

$$\sim(d_1, d_2) = V(d_1) \cdot V(d_2)$$

What about the length of a vector?

Longer documents will be represented with longer vectors, but that does not mean they are more important.

Vector normalization and similarity

Euclidian normalization (vector length normalization):

$$\vec{v}(d) = \frac{V(d)}{||V(d)||} \quad \text{where} \quad ||V(d)|| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

Similarity given by the cosine measure between normalized vectors:

$$\text{sim}(d_1, d_2) = \vec{v}(d_1) \cdot \vec{v}(d_2)$$

This similarity measure can be applied on a $M \times N$ term-document matrix, where $M$ is the size of the dictionary and $N$ that of the collection:

$$m[t, d] = \frac{v(d)}{t}$$

Example (Manning et al, 07)

<table>
<thead>
<tr>
<th>Dictionary</th>
<th>$v(d_1)$</th>
<th>$v(d_2)$</th>
<th>$v(d_3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>affection</td>
<td>0.999</td>
<td>0.993</td>
<td>0.847</td>
</tr>
<tr>
<td>jealous</td>
<td>0.087</td>
<td>0.120</td>
<td>0.466</td>
</tr>
<tr>
<td>gossip</td>
<td>0.017</td>
<td>0</td>
<td>0.254</td>
</tr>
</tbody>
</table>

$$\text{sim}(d_1, d_2) = 0.999$$
$$\text{sim}(d_1, d_3) = 0.888$$
Matching queries against documents

- Queries are represented using vectors in the same way as documents.
- In this context:
  \[
  \text{score}(q, d) = \langle \vec{v}(q), \vec{v}(d) \rangle
  \]
- In the previous example, with \( q := \) jealous gossip, we obtain:
  \[
  \langle \vec{v}(q), \vec{v}(d_1) \rangle = 0.074 \\
  \langle \vec{v}(q), \vec{v}(d_2) \rangle = 0.085 \\
  \langle \vec{v}(q), \vec{v}(d_3) \rangle = 0.509
  \]

Retrieving documents

- Basic idea: similarity cosines between the query vector and each document vector, finally selection of the top \( K \) scores.
- Provided we use the \( \text{tf} - \text{idf}_{t,d} \) measure as a weight, which information do we store in the index?
  - The size of the collection divided by the document frequency \( N_{df} \) → stored with the pointer to the postings list
  - The term frequency \( t_{f,d} \) → stored in each posting

From (Manning et al., 07)

```plaintext
1  cosineScore(query)
2  init(scores[N])  // score of each doc
3  init(length[N])  // length of each doc
4  for each t in query do
5      weight <- w(t,q)
6      post <- postings(t)
7      for each (d, tf(d,t)) in post do
8          scores[d] <- scores[d] + (w(t,q) * w(t,d))
9      endfor
10     endfor
11  for each d in keys(length) do
12      scores[d] <- scores[d] / length[d]
13  endfor
14  res[K] <- getBest(scores) (*)
15  return res
```
Improving scoring and ranking

Speeding up document scoring

- The scoring algorithm can be time consuming
- Using heuristics can help saving time
- Exact top-score vs approximative top-score retrieval
  → we can lower the cost of scoring by searching for $K$ documents that are likely to be among the top-scores
- General optimization scheme:
  1. find a set of documents $A$ such that $K < |A| < N$, and whose is likely to contain many documents close to the top-scores
  2. return the $K$ top-scoring document included in $A$

Index elimination

Idea: skip postings that are not likely to be relevant

(a) While processing the query, only consider terms whose $\text{idf(t)}$ exceeds a predefined threshold
  NB: thus we avoid traversing the posting lists of high $\text{idf(t)}$ terms, lists which are generally long

(b) only consider documents where all query terms appear

Champion lists

Idea: we know which documents are the most relevant for a given term

- For each term $t$, we pre-compute the list of the $r$ most relevant (with respect to $w(t, d)$) documents in the collection
- Given a query $q$, we compute
  \[ A = \bigcup_{t \in q} r(t) \]
  NB: $r$ can depends on the document frequency of the term.
Static quality score

Idea: only consider documents which are considered as high-quality documents

- Given a measure of quality $g(d)$, the posting lists are ordered by decreasing value of $g(d)$
- Can be combined with champion lists, i.e. build the list of $r$ most relevant documents wrt $g(d)$
- Quality can be computed from the logs of users’ queries

Impact ordering

Idea: some sublists of the posting lists are of no interest

- To reduce the time complexity:
  - query terms are processed by decreasing $idf_t$
  - postings are sorted by decreasing term frequency $tf_t,d$
  - once $idf_t$ gets low, we can consider only few postings
  - once $tf_t,d$ gets smaller than a predefined threshold, the remaining postings in the list are skipped

Cluster pruning

Idea: the document vectors are gathered by proximity

- We pick $\sqrt{N}$ documents randomly ⇒ leaders
- For each non-leader, we compute its nearest leader ⇒ followers
- At query time, we only compute similarities between the query and the leaders
- The set $A$ is the closest document cluster
- NB: the document clustering should reflect the distribution of the vector space

Tiered indexes

- This technique can be seen as a generalization of champion lists
- Instead of considering one champion list, we manage layers of champion lists, ordered in increasing size:

<table>
<thead>
<tr>
<th>Tier</th>
<th>Documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$l$ most relevant documents</td>
</tr>
<tr>
<td>2</td>
<td>next $m$ most relevant documents</td>
</tr>
<tr>
<td>3</td>
<td>next $n$ most relevant documents</td>
</tr>
</tbody>
</table>

Indexed defined according to thresholds
Query-term proximity

- Priority is given to documents containing many query terms in a close window
- Needs to pre-compute n-grams
- And to define a proximity weighting that depends on the wide size $n$ (either by hand or using learning algorithms)

Scoring optimisations – summary

1. Index elimination
2. Champion lists
3. Static quality score
4. Impact ordering
5. Cluster pruning
6. Tiered indexes
7. Query-term proximity

Putting it all together

- Many techniques to retrieve documents (using logical operators, proximity operators, or scoring functions)
- Adapted technique can be selected dynamically, by parsing the query
- First process the query as a phrase query, if fewer than $K$ results, then translate the query into phrase queries on bi-grams, if there are still too few results, finally process each term independently (real free text query)

Conclusion

- What we have seen today?
  - Term weighting using $tf - idf_d$
  - Vector space model (cosine similarity)
  - Optimizations for document ranking
- Next lecture?
  - Other weighting schemes
References

- C. Manning, P. Raghavan and H. Schütze, Introduction to Information Retrieval (sections 6.2 and 6.3, chapter 7)

  http://citeseer.ist.psu.edu/675266.html