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 retrived
- 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

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 occuring 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* ?

Term t	cf_t	df_t
try	10422	8760
insurance	10440	3997

Inverse Document Frequency

- *inverse document frequency* of a term t:

$$idf_t \;=\; log \frac{N}{df_t} \qquad \text{with } N = \text{collection size}$$

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

Term t	df_t	idf_t
car	18165	1.65
auto	6723	2.08
insurance	19241	1.62
best	25235	1.5

tf-idf weighting

- The weight of a term is computed using both *tf* and *idf*:

$$w(t,d) \;=\; tf_{t,d} \times idf_t \qquad \text{called } tf - idf_{t,d}$$

- $w(t,d)$ 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:

$$score(q,d) \;=\; \sum_{t \in q} w(t,d)$$

Vector space model

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:

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

- Question: does this representation allow to compute the similarity between documents ?
- Similarity between vectors ?
  $\rightarrow$ inner product $\vec{V(d_1)}.\vec{V(d_2)}$
- 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):

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

- Similarity given by the *cosine* measure between normalized vectors:

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

- 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:

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

Example (Manning et al, 07)

Dictionary	$\vec{v(d_1)}$	$\vec{v(d_2)}$	$\vec{v(d_3)}$
affection	0.996	0.993	0.847
jealous	0.087	0.120	0.466
gossip	0.017	0	0.254

$sim(d_1, d_2) = 0.999$
$sim(d_1, d_3) = 0.888$

Matching queries against documents

- Queries are represented using vectors in the same way as documents
- In this context:

$$score(q, d) = v(\vec{q}).v(\vec{d})$$

- In the previous example, with $q :=$ *jealous gossip*, we obtain:

$$\begin{aligned} v(\vec{q}).v(\vec{d_1}) &= 0.074 \\ v(\vec{q}).v(\vec{d_2}) &= 0.085 \\ v(\vec{q}).v(\vec{d_3}) &= 0.509 \end{aligned}$$

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 $tf - idf_{t,d}$ measure as a weight, which information do we store in the index ?

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 $tf - idf_{t,d}$ measure as a weight, which information do we store in the index ?
  - The size of the collection divided by the document frequency $\frac{N}{df_t}$
    → stored with the pointer to the postings list
  - The term frequency $tf_{t,d}$
    → stored in each posting

From (Manning et al., 07)

```
cosineScore(query)
init(scores[N])    // score of each doc
init(length[N])    // length of each doc
for each t in query do
  weight <- w(t,q)
  post   <- postings(t)
  for each (d, tf(d,t)) in post do
    scores[d] <- scores[d] + (w(t,q) * w(t,d))
  endfor
endfor
for each d in keys(length) do
  scores[d] <- scores[d] / length[d]
endfor
res[K] <- getBest(scores)  (*)
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
  $\longrightarrow$ 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 idf_t exceeds a predefined threshold

NB: thus we avoid traversing the posting lists of high 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 $\Rightarrow$ leaders
- For each non-leader, we compute its nearest leader $\Rightarrow$ 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:

index 1	l most relevant documents
index 2	next m most relevant documents
index 3	next n most relevant documents

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 $tf - idf_{t,d}$
  - 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://nlp.stanford.edu/IR-book/pdf/chapter06-tfidf.pdf

  http://nlp.stanford.edu/IR-book/pdf/chapter07-vectorspace.pdf

- S. Garcia, H. Williams and A. Cannane, *Access ordered indexes* (2004)
  http://citeseer.ist.psu.edu/675266.html/