Information Retrieval

Lecture 3: Vector Space Model

[Reference] CS276: Information Retrieval and Web Search

This lecture

- Vector space scoring
  - **tf×idf** and vector space

- Document is a **vector** in $\mathbb{R}^{V}$

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	13.1	11.4	0.0	0.0	0.0	0.0
Brutus	3.0	8.3	0.0	1.0	0.0	0.0
Caesar	2.3	2.3	0.0	0.5	0.3	0.3
Calpurnia	0.0	11.2	0.0	0.0	0.0	0.0
Cleopatra	17.7	0.0	0.0	0.0	0.0	0.0
mercy	0.5	0.0	0.7	0.9	0.9	0.3
worser	1.2	0.0	0.6	0.6	0.6	0.0

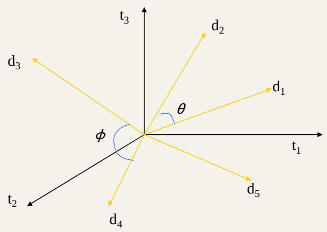
Documents as vectors

- Each doc *d* can now be viewed as a vector of *wf*×*idf* values, one component for each term
- **vector space**
  - terms are axes
  - docs live in this space
  - even with stemming, may have 50,000+ dimensions

Why turn docs into vectors?

- First application: Query-by-example
  - Given a doc ***d***, find others "**like**" it.
- Now that ***d*** is a vector,
- Find vectors (docs) "**near**" it.

Intuition(直觉)



Postulate(假设):

Documents that are "**close together**" in the vector space talk about the same things.

Proximity

- If d_1 is **near** d_2, then d_2 is **near** d_1.
- If d_1 **near** d_2, and d_2 **near** d_3, then d_1 is **not far** from d_3.
- No doc is closer to d than d itself.

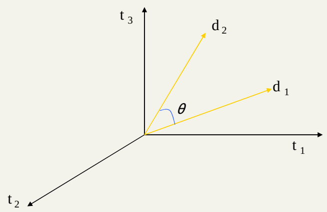
First

- Idea:

  Distance between d_1 and d_2 is the length of the **vector** $|d_1 - d_2|$
  - Euclidean distance
- Why is this not a great idea?
- Still haven't dealt with the issue of length normalization
  - Short documents would be more similar to each other by virtue of length, not topic

Cosine similarity(余弦相似)

- Distance between vectors d_1 and d_2 *captured* by the cosine of the angle x between them.
- This is *similarity*, not distance
  - No triangle inequality for similarity.



Cosine similarity (续)

- A vector can be *normalized* (given a length of 1) by dividing each of its components by its length
- use the L_2 norm $\|\mathbf{x}\|_2 = \sqrt{\sum_i x_i^2}$
- This maps vectors onto the unit sphere:
- Then, $\left|\vec{d}_j\right| = \sqrt{\sum_{i=1}^{n} w_{i,j}} = 1$
- Longer documents don't get more weight

Example

- tf values for documents

	Doc1	Doc2	Doc3
car	27	4	24
auto	3	33	0
insurance	0	33	29
best	14	0	17

- Euclidean normalized tf values for documents

	Doc1	Doc2	Doc3
car	0.88	0.09	0.58
auto	0.10	0.71	0
insurance	0	0.71	0.70
best	0.46	0	0.41

Normalized vectors

- For normalized vectors, the cosine is simply the dot product:

$$\cos(\vec{d}_j, \vec{d}_k) = \vec{d}_j \cdot \vec{d}_k$$

- Euclidean distance between vectors:

$$\left|d_j - d_k\right| = \sqrt{\sum_{i=1}^{n} \left(d_{i,j} - d_{i,k}\right)^2}$$

- Euclidean distance gives the same proximity ordering as the cosine measure

Cosine similarity (续)

$$sim(d_j, d_k) = \frac{\vec{d}_j \cdot \vec{d}_k}{\left|\vec{d}_j\right|\left|\vec{d}_k\right|} = \frac{\sum_{i=1}^{n} w_{i,j} w_{i,k}}{\sqrt{\sum_{i=1}^{n} w_{i,j}^2}\sqrt{\sum_{i=1}^{n} w_{i,k}^2}}$$

- **Cosine of angle** between two vectors
- The denominator involves the lengths of the vectors.

Normalization

Queries in the vector space model

Central idea: the query as a vector

- Regard the query as short document
- Return the documents
  - ranked by the closeness of their vectors to the query
  - represented as a vector.

$$sim(d_j, d_q) = \frac{\vec{d}_j \cdot \vec{d}_q}{\quad} = \frac{\sum_{i=1}^{n} w_{i,j} w_{i,q}}{\quad}$$

- Note that d_q is very sparse(稀疏)!

Example

- Consider the query best car insurance on a fictitious collection
  - N = 1,000,000 documents
  - document frequencies of auto, best, car, insurance
    - 5000, 50000, 10000 ,1000

term	query				document			product
	tf	df	idf	$w_{t,q}$	tf	wf	$w_{t,d}$	
auto	0	5000	2.3	0	1	1	0.41	0
best	1	50000	1.3	1.3	0	0	0	0
car	1	10000	2.0	2.0	1	1	0.41	0.82
insurance	1	1000	3.0	3.0	2	2	0.82	2.46

Summary: Vector Model

- A pair (ki,dj) is positive and non-binary weight
- Index terms are weighted
  - w_{ij} be the weight associated with the pair (ki,dj)
  - w_{iq} be the weight associated with the pair [ki,q]

- Query vector q is defined as **q** = $(w_{1q}, w_{2q}, \ldots, w_{tq})$
- Document vector dj is defined as **dj** = $(w_{1j}, w_{2j}, \ldots, w_{tj})$

Calculation of Weights

- N: total number of documents
- n_i: number of document in which term ki appears
- f_{ij}: normalized frequency of term ki in dj
- $freq_{ij}$: raw frequency of ki in dj
- Max_l : maximum is computed over all terms which are mentioned in dj
- idf_i : inverse document frequency for ki

$$f_{ij} = \frac{freq_{ij}}{\max_l (freq_{lj})} \qquad idf_i = \log \frac{N}{n_i}$$

$$w_{ij} = f_{ij} \times \log \frac{N}{n_i} \qquad w_{iq} = (0.5 + \frac{0.5\, freq_{iq}}{\max_l (freq_{lq})}) \times \log \frac{N}{n_i}$$

Efficient cosine ranking

Efficient cosine ranking

- Find the *k* docs in the corpus "nearest" to the query ⇒ *k* largest query-doc cosines.
- Efficient ranking:
  - Computing a single cosine efficiently.
  - Choosing the *k* largest cosine values efficiently.
    - Can we do this without computing all *n* cosines?
      - *n* = number of documents in collection

Efficient cosine ranking (续)

- What are doing in effect: solving the *k*-nearest neighbor problem for a query vector
- In general, do not know how to do this efficiently for high-dimensional spaces
- But it is solvable for short queries, and standard indexes are optimized to do this

```
CosineScore(q)
    float Scores[N]={0}
    Initialize Length[N]
    for each query term t
    do calculate w_t,q and postings list for t
        for each pair(d,tf_t,d) in postings list
        do Scores[d] += wf_t,d × w_t,q
    Read Length[d]
    for each d
    do Scores[d] = Scores[d] / Length[d]
    return Top K of Scores[]
```

Computing a single cosine

- For every *term* i, with each doc j, store term frequency tf_{ij}.
  - Some tradeoffs on whether to store term count, term weight, or weighted by idf_i.
- At *query* time, use an array of accumulators A_j to accumulate component-wise sum

$$sim(\vec{d}_j, \vec{d}_q) = \sum_{i=1}^{m} w_{i,j} \times w_{i,q}$$

Encoding document frequencies

aargh	10	→
abacus	8	→
acacia	35	→

1,2	7,3	83,1	87,2	…
1,1	5,1	13,1	17,1	…
7,1	8,2	40,1	97,3	…

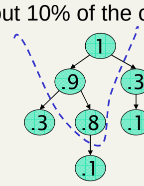
- Add $tf_{t,d}$ to postings lists
  - Now almost always as frequency – scale at runtime
  - Unary code is quite effective here
  - Overall, requires little additional space

Computing the k largest cosines: selection vs. sorting

- Typically, want to retrieve the top k docs
  - (in the cosine ranking for the query)
  - not to totally order all docs in the corpus
- Can pick off docs with k highest cosines?

Use **heap** for selecting top k

- Binary tree
  - which each node's value > the values of children
- Takes $2n$ operations to construct, then each of k log n "winners" read off in 2log n steps.
- For n=1M, k=100, this is about 10% of the cost of sorting.



Bottleneck

- Still need to first compute cosines from query to each of *n* docs → several seconds for *n* = 1M.
- Can select from only non-zero cosines
  - Need union of postings lists accumulators (<<1M)
    - on the query ***aargh abacus*** would only do accumulators 1,5,7,13,17,83,87 (below).
  - Better iff this set is < 20% of *n*

aargh	10	→	1,2	7,3	83,1	87,2	…
abacus	8	→	1,1	5,1	13,1	17,1	…
acacia	35	→	7,1	8,2	40,1	97,3	…

Removing bottlenecks

- Can further limit to documents with non-zero cosines on rare (high idf) words

Or

- Enforce conjunctive search (Google):
  - non-zero cosines on *all* words in query
    - Get # accumulators down to {min of postings lists sizes}
- But in general still potentially expensive
  - Sometimes have to fall back to (expensive) soft-conjunctive search:
    - If no docs match a 4-term query, look for 3-term subsets, etc.

```
FASTCOSINESCORE(q)
 float Scores[N] = 0
 for each d
 do Initialize Length[d] to the length of doc d
 for each query term t
 do calculate w_{t,q} and fetch postings list for t
     for each pair(d, tf_{t,d}) in postings list
     do add wf_{t,d} to Scores[d]
 Read the array Length[d]
 for each d
 do Divide Scores[d] by Length[d]
 return Top K components of Scores[]
```

Can we avoid all this computation ?

- may occasionally get an answer wrong
  - a doc *not* in the top *k* may creep into the answer

Limiting the accumulators: Best *m* candidates

- Preprocess:
  - Pre-compute, for each term, its *m* nearest docs.
    - Treat each term as a 1-term query.
    - Lots of preprocessing.
  - Result: "preferred list" for each term.
- Search:
  - For a *t*-term query, take the union of their *t* preferred lists
    - call this **set *S***, where $|S| \le mt$.
  - Compute cosines from the query to only the docs in ***S***, and choose the top ***k***.

Need to pick *m>k* to work well empirically.

Limiting the accumulators: Frequency/impact ordered postings

- Idea:
  - only want to have accumulators for documents for
    - which $wf_{t,d}$ is high enough
- Sort postings lists by this quantity
- Retrieve terms by **idf**, and then retrieve only one block of the postings list for each term
- Continue to process more blocks of postings
  - until have enough accumulators
- Can continue one that ended with highest $wf_{t,d}$
  - The number of accumulators is bounded
- Anh et al. 2001

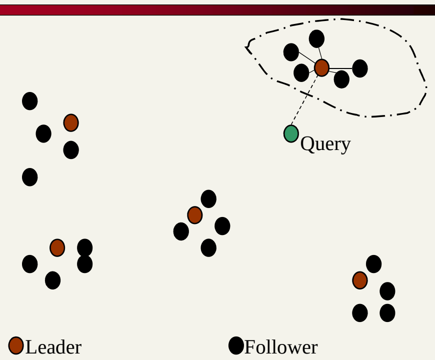
Cluster pruning(修剪): preprocessing

- Pick $\sqrt{n}$ *docs* at random:
  - call these *leaders*
- For each other doc
  - pre-compute nearest leader
  - Docs attached to a leader
    - its *followers*
  - Likely: each leader has ~ $\sqrt{n}$ followers.

Cluster pruning: query processing

- Process a query as follows:
  - Given query *Q*
    - find its nearest *leader L*.
  - Seek *k* nearest docs from among *L*'s followers.

Visualization



Why use random sampling

- Fast
- Leaders reflect data distribution

General variants

- Have each follower attached to *a*=3 (say) nearest leaders.
- From query, find *b*=4 (say) nearest leaders and their followers.
- Can recur on leader/follower construction.