Retrieval Strategies: Vector Space Model and Boolean

(COSC 488)

Nazli Goharian
nazli@cs.georgetown.edu

Retrieval Strategy

- An IR *strategy* is a technique by which a relevance measure is obtained between a query and a document.

Retrieval Strategies

- Manual Systems
  - Boolean, Fuzzy Set
- Automatic Systems
  - Vector Space Model
  - Language Models
  - Latent Semantic Indexing
- Adaptive
  - Probabilistic, Genetic Algorithms , Neural Networks, Inference Networks

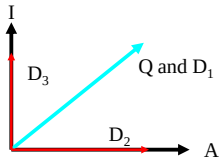
Vector Space Model

- One of the most commonly used strategy is the vector space model (proposed by Salton in 1975)
- Idea: Meaning of a document is conveyed by the words used in that document.
- Documents and queries are mapped into term vector space.
- Each dimension represents *tf-idf* for one term.
- Documents are ranked by closeness to the query. Closeness is determined by a *similarity score* calculation.

Document and query presentation in VSM (Example)

- Consider a two term vocabulary, A and I

Query: A I
D_1 - A I
D_2 - A
D_3 – I



Idea: a document and a query are similar as their vectors point to the same general direction.

Weights for Term Components

- Using Term Weight to rank the relevance.
- Parameters in calculating a weight for a document term or query term:
  - Term Frequency (tf): Term Frequency is the number of times a term *i* appears in document *j* (tf_{ij})
  - Document Frequency (df): Number of documents a term *i* appears in, (df_i).
  - Inverse Document Frequency (idf): A discriminating measure for a term *i* in collection, i.e., how discriminating term *i* is.
    (idf_i) = $\log_{10}(n / df_i)$, where *n* is the number of document

Weights for Term Components

- Classic thing to do is use *tf x idf*
- Incorporate *idf* in the query and the document, one or the other or neither.
- Scale the *idf* with a log
- Scale the *tf* (log tf+1) or (tf/sum *tf* of all terms in that document)
- Augment the weight with some constant (e.g.; w = (w)(0.5))

Weights for Term Components

- Many variations of term weight exist as the result of improving on basic *tf-idf*
- A good one:

$$w_{ij} = \frac{(\log tf_{ij} + 1.0) * idf_j}{\sum_{j=1}^{t} [(\log tf_{ij} + 1.0) * idf_j]^2}$$

- Some efforts suggest using different weighting for document terms and query terms.

Similarity Measures

• Similarity Coefficient (SC) identifies the Similarity between query Q and document D_i

- •Inner Product (dot Product)
- •Cosine
- •Pivoted Cosine

Similarity Measures: (Inner Product)

- Inner Product (dot product)

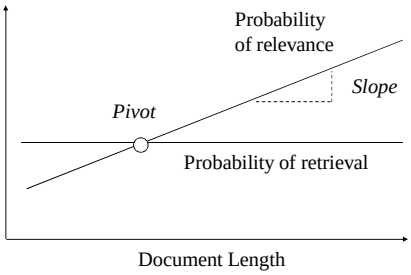
$$SC(Q, D_i) = \sum_{j=1}^{t} w_{qj} x\ d_{ij}$$

- Problem: Longer documents will score very high because they have more chances to match query words.

Similarity Measures: (Cosine)

$$SC(Q, D_i) = \frac{\sum_{j=1}^{t} w_{qj} x\ d_{ij}}{\sqrt{\sum_{j=1}^{t} (d_{ij})^2 \sum_{j=1}^{t} (w_{qj})^2}}$$

- Assumption: document length has no impact on the relevance.
- Normalizes the weight by considering document length.
- Problem: Longer documents are somewhat penalized because indeed they might have more components that are indeed relevant [Singhal, 1997- Trec]



Pivoted Cosine Normalization

- Comparing likelihood of retrieval and relevance in a collection to identify *pivot* and thus, identify the new *correction factor*.

$$SC(Q, D_i) = \frac{\sum_{j=1}^{t} w_{qj}\, d_{ij}}{(1.0 - s) + (s)\frac{\sqrt{\sum_{j=1}^{t} (d_{ij})^2}}{avgn}}$$

Avgn: average document normalization factor over entire collection
s: can be obtained empirically

Pivoted Cosine Normalization

- *Pivoted Cosine Normalization* worked well for short and moderately long documents.
- Extremely long documents are favored

Pivoted Unique Normalization

$$SC(Q, D_i) = \frac{\sum_{j=1}^{t} w_{qj}\, d_{ij}}{(1.0 - s)p + (s)(|d_i|)}$$

dij = (1+log(tf))idf/ (1+log(atf)) where, *atf* is average *tf*
|di|: number of unique terms in a document.
p: average of number of unique terms of documents over entire collection
s: can be obtained empirically

VSM Example

- Q: "gold silver truck"
- D_1: "Shipment of gold damaged in a fire"
- D_2: "Delivery of silver arrived in a silver truck"
- D3: "Shipment of gold arrived in a truck"

Id	Term	df	idf
1	a	3	0
2	arrived	2	0.176
3	damaged	1	0.477
4	delivery	1	0.477
5	fire	1	0.477
6	gold	2	0.176
7	in	3	0
8	of	3	0
9	silver	1	0.477
10	shipment	2	0.176
11	truck	2	0.176

VSM Example

doc	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
D_1	0	0	.477	0	.477	.176	0	0	0	.176	0
D_2	0	.176	0	.477	0	0	0	0	.954	0	.176
D_3	0	.176	0	0	0	.176	0	0	0	.176	.176
Q	0	0	0	0	0	.176	0	0	.477	0	.176

- Computing SC using inner product:
- $SC(Q, D_1) = (0)(0) + (0)(0) + (0)(0.477) + (0)(0) + (0)(0.477) + (0.176)(0.176) + (0)(0) + (0)(0)$

Algorithm for Vector Space (dot product)

- Assume: *t.idf* gives the *idf* of any term *t*
- *q.tf* gives the *tf* of any query term

```
Begin
    Score[] ← 0
    For each term t in Query Q
        Obtain posting list l
        For each entry p in l
            Score[p.docid] = Score[p.docid] + (p.tf * t.idf)(q.tf * t.idf)
```

- Now we have a SCORE array that is unsorted.
- Sort the score array and display top *x* results.

Summary: Vector Space Model

- Pros
  - Fairly cheap to compute
  - Yields decent effectiveness
  - Very popular
- Cons
  - No theoretical foundation
  - Weights in the vectors are arbitrary
  - Assumes term independence

Boolean Retrieval

- For many years, most commercial systems were only Boolean.
- Most old library systems and Lexis/Nexis have a long history of Boolean retrieval.
- Users who are experts at a complex query language can find what they are looking for.
  (t1 AND t2) OR (t3 AND t7) WITHIN 2 Sentences (t4 AND t5) NOT (t9 OR t10)
- Considers each document as bag of words

Boolean Retrieval

- *Expression*:=
  - term
  - (*expr*)
  - NOT *expr (not recommended)*
  - *expr* AND *expr*
  - *expr* OR *expr*
- (cost OR price) AND paper AND NOT article

Boolean Example

doc	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
D_1	0	0	1	0	1	1	0	0	0	1	0
D_2	1	1	0	1	0	0	0	0	1	0	1
D_3	1	1	0	0	0	1	0	0	0	1	1
D4	0	0	0	0	0	1	0	0	1	0	1

Q: t1 AND t2 AND NOT t4

0110 AND 0110 AND 1011 = 0010 That is D3

Processing Boolean Queries

- Doc-term matrix is too sparse, thus, using inverted index
- Query optimization in Boolean retrieval:
  The order in which posting lists are accessed!

Processing Boolean Query
t1 AND t2

- Algorithm:
  Find t1 in index (lexicon)
  Retrieve its posting list
  Find t2 in index (lexicon)
  Retrieve its posting list
  Intersect (merge) the posting lists
  The matching DodIDs are added to the result list

Processing Boolean Query
t1 AND t2 AND t3

- What is the best order to process this?
- Process in the order of increasing document frequency, i.e, smaller Posting Lists first!
- Thus, if t_1, t_2 have smaller PL than t_3, then process as:

  (t1 AND t2) AND t3

Intersection of Posting Lists

Algorithm

Sort query terms based on document frequency

Merge the smallest posting list with the next smallest posting list and create the result set

Merge the next smaller posting list with the result set, update the result set

Continue till no more terms left

Processing Boolean Query
(t1 OR t2) AND (t3 OR t4) AND (t5 OR t6)

- Using document frequency estimate the size of disjuncts
- Order the conjuncts in order of smaller disjuncts

Boolean Retrieval

- AND returns too few documents (low recall)
- OR return too many document (low precision)
- NOT eliminates many good documents (low recall)
- Proximity information not supported
- Term weight not incorporated

Extended (Weighted) Boolean Retrieval

- Extended Boolean supports term weight and proximity information.
- Example of incorporating term weight:
  - Ranking by term frequency (Sony Search Engine)

    x AND y: tf_x x tf_y

    x OR y: $tf_x + tf_y$

    NOT x: 0 if $tf_x > 0$, 1 if $tf_x = 0$
- User may assign term weights

  cost and +paper

Summary of Boolean Retrieval

- Pro
  - Can use very restrictive search
  - Makes experienced users happy
- Con
  - Simple queries do not work well.
  - Complex query language, confusing to end users