Document Retrieval and Inverted Indexes (Section 14.1.8)

Credits for slides: Hofmann, Mihalcea, Mobasher, Mooney, Schutze.

Announcements

- HW4 graded, solutions posted online
- Project assignment – DB design – due by midnight
- Exam 1 – Monday, October 7th
  - Covers lectures 1-12 (last topic covered - transactions), homework assignments 1-4
  - Sample exam posted online
  - You are allowed to bring one sheet (front and back) with notes to the exam
- Project presentations – October 9-11
  - Evaluation form for presentation will be posted today
- Homework assignment 5 will be posted today, due October 18th

Naïve Implementation

Convert all documents in collection D to tf-idf weighted vectors, $\boldsymbol{d}_j$, for keyword vocabulary V.

Convert query to a tf-idf-weighted vector $\boldsymbol{q}$.

For each $\boldsymbol{d}_j$ in D do

  Compute score $s_j = \text{cosSim}(\boldsymbol{d}_j, \boldsymbol{q})$

Sort documents by decreasing score.

Present top ranked documents to the user.

Time complexity: $O(|V| \cdot |D|)$ Bad for large V & D !

$|V| = 10{,}000$; $|D| = 100{,}000$; $|V| \cdot |D| = 1{,}000{,}000{,}000$

Practical Implementation

- Based on the observation that documents containing none of the query keywords do not affect the final ranking
- Try to identify only those documents that contain at least one query keyword
- Actual implementation of an inverted index

Vector Space Model: Implementation Steps

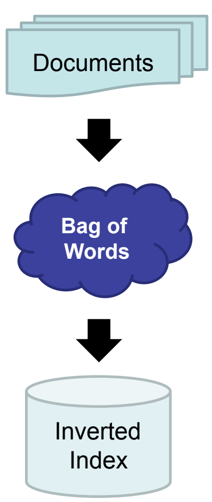
Step 1: Preprocessing

Step 2: Indexing

Step 3: Retrieval

Step 4: Ranking

Preprocessing



case folding, tokenization, stopword removal, stemming

~~syntax~~, ~~semantics~~, word ~~knowledge~~, etc.

Sparse Vectors

- Vocabulary and therefore dimensionality of vectors can be very large, ~10^4 .
- However, most documents and queries do not contain most words, so vectors are sparse (i.e. most entries are 0).
- Need efficient methods for storing and computing with sparse vectors.

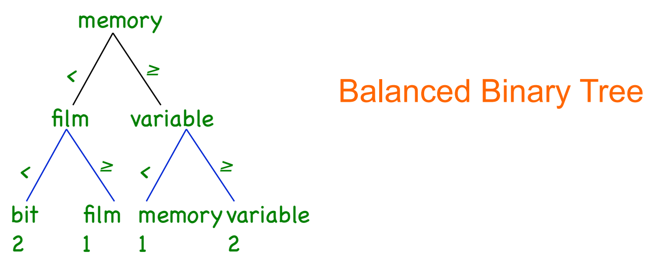
Sparse Vectors as Linked Lists

- Store vectors as linked lists of non-zero-weight tokens paired with a weight.
  - Space proportional to the number of unique tokens (*n*) in the document.
  - Requires linear search of the list to find (or change) the weight of a specific token.
  - Requires quadratic time in n in worst case to compute vector for a document:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Sparse Vectors as Trees

- Index tokens in a document in a balanced binary tree or trie with weights stored with tokens at the leaves.



- Space overhead for tree structure: ~2*n* nodes.
- O(log *n*) time to find or update weight of a specific token.
- O(*n* log *n*) time to construct vector.

Implementation Based on Inverted Files

- In practice, document vectors are not stored directly; an inverted organization provides much better efficiency.
- The keyword-to-document index can be implemented as a hashtable, a sorted array, or a tree-based data structure (trie, B-tree).
- Critical issue is logarithmic or constant-time access to token information.

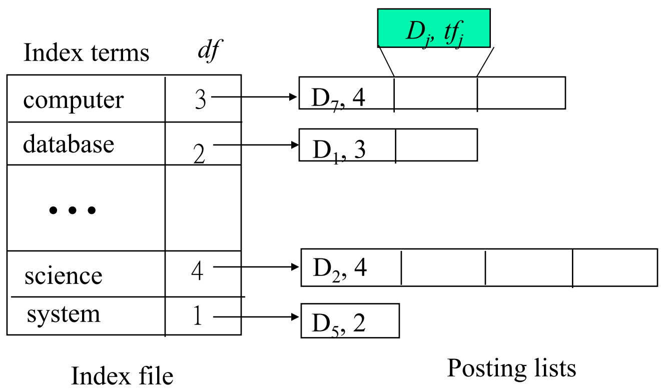
Indexing

- Build an inverted index, with an entry for each token (word) in the vocabulary
- Input: Tokens obtained from the preprocessing module
- Output: An inverted index for fast access

Index Data Structure

- Many data structures are appropriate for fast access
  - We will use hashtables
- We need:
  - One entry for each word in the vocabulary
  - For each such entry:
    - Keep a list of all the documents where it appears together with the corresponding frequency → TF
    - Keep the total number of documents in which the corresponding word appears → IDF
- Constant time to find or update weight of a specific token (ignoring collisions).
- O(*n*) time to construct the vector of a document (ignoring collisions).

Example



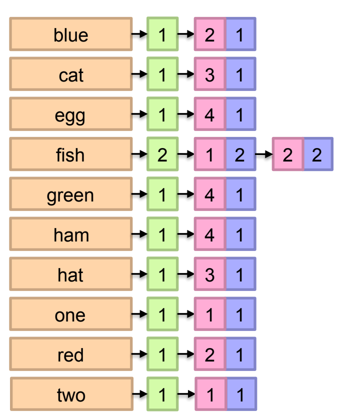
Inverted Index: Counting Words

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham



Indexing – How many passes through the data?

- TF and IDF for each token can be computed in one pass
- Cosine similarity also requires document lengths
- Need a second pass to compute document vector lengths
  - Remember that the length of a document vector is the square-root of sum of the squares of the weights of its tokens.
  - Remember the weight of a token is: TF * IDF
  - Therefore, must wait until IDF's are known (and therefore until all documents are indexed) before document lengths can be determined.
- Do a second pass over all documents: keep a list or hashtable with all document id's, and for each document determine its length.

CosSim($\mathbf{d}_j$, $\mathbf{q}$) = $$\frac{\vec{d_j}\cdot\vec{q}}{\left|\vec{d_j}\right|\cdot\left|\vec{q}\right|} = \frac{\sum_{i=1}^{n}(w_{ij}\cdot w_{iq})}{\sqrt{\sum_{i=1}^{n}w_{ij}^{2}\cdot\sum_{i=1}^{n}w_{iq}^{2}}}$$

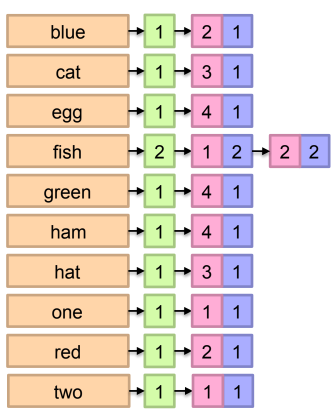
Inverted Index: Document Length?

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham



$$\vec{d_j} = \sqrt{\sum_{i=1}^{n} w_{ij}^{2}}$$

Time Complexity of Indexing

- Complexity of creating vector and indexing a document of n tokens is $O(n)$.
- So indexing m such documents is $O(m\ n)$.
- Computing token IDFs can be done during the same first pass
- Computing vector lengths is also $O(m\ n)$.
- Complete process is $O(m\ n)$, which is also the complexity of just reading in the corpus.

Step 3: Retrieval with an Inverted Index

- Input: Query and Inverted Index (from Step 2)
- Output: Similarity values between query and documents

- Tokens that are not in both the query and the document do not affect cosine similarity.
  - Product of token weights is zero and does not contribute to the dot product.
- Usually the query is fairly short, and therefore its vector is *extremely* sparse.
- Use inverted index to find the limited set of documents that contain at least one of the query words.

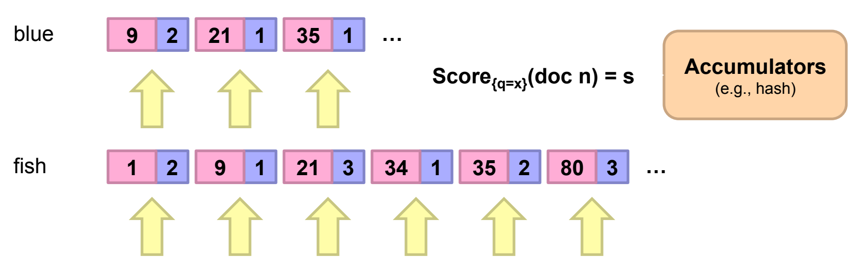
Processing the Query

- Incrementally compute cosine similarity of each indexed document as query words are processed one by one.
- To accumulate a total score for each retrieved document, store retrieved documents in a hashtable, where DocumentReference is the key and the partial accumulated score is the value.

Retrieval: Query-At-A-Time

Evaluate documents one query term at a time

Usually, starting from most rare term (often with tf-sorted postings)

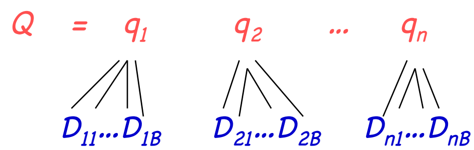


- We assume the query is "blue fish"

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

Inverted Query Retrieval Efficiency

- Assume that, on average, a query word appears in *B* documents:



- Then retrieval time is O(|*Q*| *B*), which is typically, **much** better than naïve retrieval that examines all |*D*| documents, O(|*V*| |*D*|), because |*Q*| << |*V*| and *B* << |*D*|.

Step 4: Ranking

- Sort the hashtable including the retrieved documents based on the value of cosine similarity
- Return the documents in descending order of their relevance

- Input: Similarity values between query and documents
- Output: Ranked list of documented in reversed order of their relevance

Abstract IR Architecture