CIS 833 – Information Retrieval and Text Mining

Lecture 7

Vector Space Model

September 15, 2015

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

Assignments

- HW1 due tomorrow
- The *warmup* WordCount MapReduce programming assignment has been posted online (due September 23rd)

Required Reading

- "Information Retrieval" textbook
  - Chapter 2: Term Vocabulary and Posting Lists
  - Chapter 4: Index Construction

Review

- Key idea of the vector space model?
- Explain the TF-IDF weighting scheme
- Similarity measures?

Vector Space Model

Implementation?

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 = cosSim($\boldsymbol{d}_j$, $\boldsymbol{q}$)

Sort documents by decreasing score.

Present top ranked documents to the user.

Time complexity?

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 = cosSim($\boldsymbol{d}_j$, $\boldsymbol{q}$)

Sort documents by decreasing score.

Present top ranked documents to the user.

Time complexity: O(|V|·|D|) Bad for large V & D !

|V| = 10,000; |D| = 100,000; |V|·|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

Step 1: Preprocessing

- Implement the preprocessing functions:
  - For tokenization
  - For stop word removal
  - For stemming
- Input: Documents that are read one by one from the collection
- Output: Tokens to be added to the index
  - No punctuation, no stop-words, stemmed

Simple Tokenizing

- Analyze text into a sequence of discrete tokens (words).
- Sometimes punctuation (e-mail), numbers (1999), and case (Republican vs. republican) can be a meaningful part of a token.
  - Sometimes they are not.
- Simplest approach is to ignore all numbers and punctuation, and use only case-insensitive unbroken strings of alphabetic characters as tokens.
- More careful approach:
  - Separate ? ! ; : " ' [ ] ( ) < >
  - E.g., care with .
    - ***U.S.A.*** vs. ***USA***

Tokenizing HTML

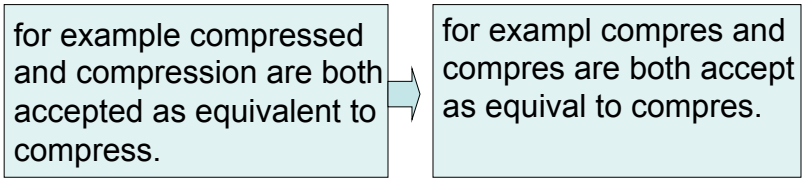
- Should text in HTML commands not typically seen by the user be included as tokens?
  - Words appearing in URLs?
  - Words appearing in "meta text" of images?
- Simplest approach is to exclude all HTML tag information (between "<" and ">") from tokenization.

Stopwords

- It is typical to *exclude* high-frequency words (e.g. function words: “a”, “the”, “in”, “to”; pronouns: “I”, “he”, “she”, “it”).
- Stopwords are language dependent ~ standard set for English contains approximately 500 words.
- For efficiency, store strings for stopwords in a hashtable to recognize them in constant time.

Stemming

- Reduce tokens to “root” form of words to recognize morphological variation.
  - “computer”, “computational”, “computation” all reduced to same token “comput”
- Correct morphological analysis is language specific and can be complex.
- Stemming “blindly” strips off known affixes (prefixes and suffixes) in an iterative fashion.



Porter Stemmer

- Simple procedure for removing known affixes in English without using a dictionary.
- Can produce unusual stems that are not English words:
  - "computer", "computational", "computation" all reduced to same token "comput"
- May conflate (reduce to the same token) words that are actually distinct.
- May not recognize all morphological derivations.

Typical rules in Porter

- *sses* → *ss*
- *ies* → *i*
- *ational* → *ate*
- *tional* → *tion*

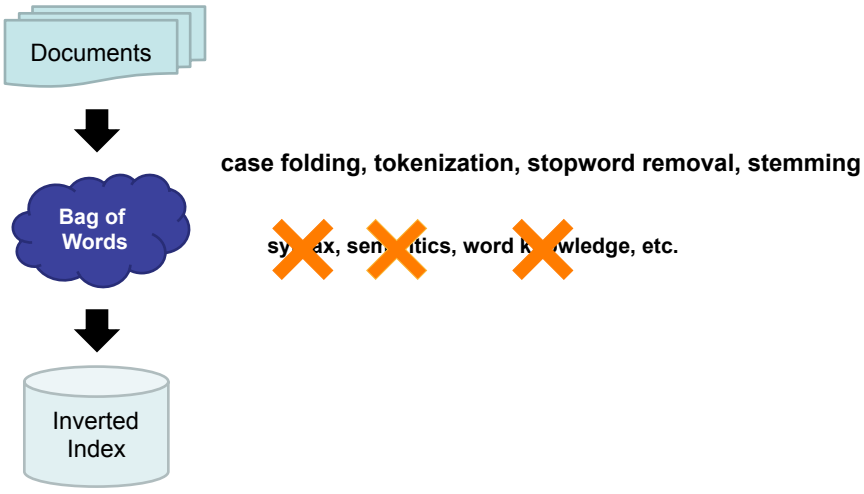
- Porter stemmer

http://www.tartarus.org/~martin/PorterStemmer/def.txt

Porter Stemmer Errors

- Errors of "comission":
  - organization, organ → organ
  - police, policy → polic
  - arm, army → arm
- Errors of "omission":
  - cylinder, cylindrical
  - create, creation
  - Europe, European

Preprocessing Summary



Vector Space Model: Implementation Steps

Step 1: Preprocessing

Step 2: Indexing

Step 3: Retrieval

Step 4: Ranking

Step 2: Indexing?

- Why do we need to create an index?

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.

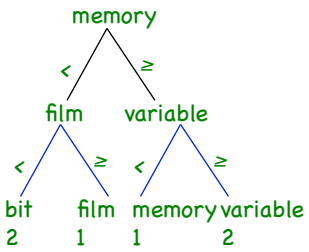
Sparse Vectors as Lists

- 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 the 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.



Balanced Binary Tree

- 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 fast, ideally constant-time, access to token information.

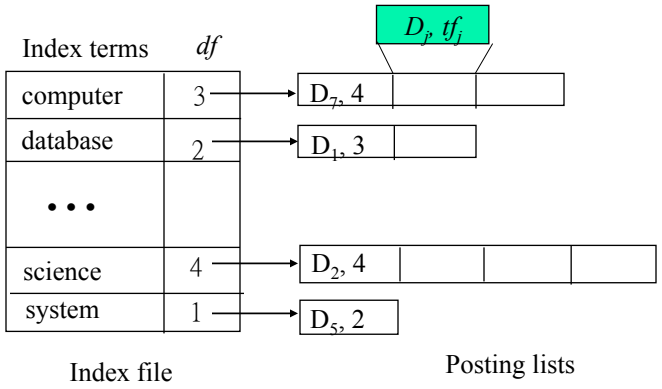
Indexing

- 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: TF.IDF

Doc 1 **one fish, two fish**

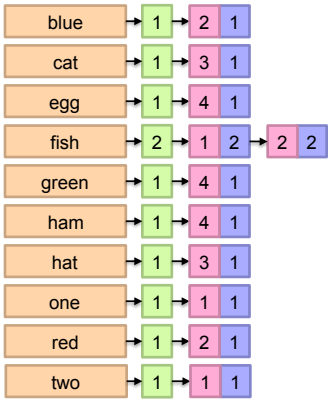
Doc 2 **red fish, blue fish**

Doc 3 **cat in the hat**

Doc 4 **green eggs and ham**

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 ids, and for each document determine its length.

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

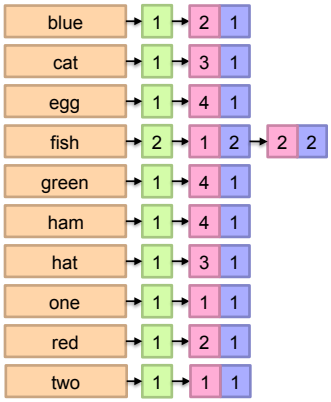
Inverted Index: Document Length?

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.