Basic Text Processing and Indexing

Document Processing Steps

- Lexical analysis (tokenizing)
- Stopwords removal
- Stemming
- Selection of indexing terms among the word collection
- Construction of indexing system

Simple Tokenizing

- Separate text into a sequence of discrete tokens (words).
- Sometimes punctuation (e-mail), numbers (1999), and case (China vs. china) can be a meaningful part of a token.
- However, frequently they are not.
- Simplest approach is to ignore all numbers and punctuation and use only case-insensitive unbroken strings of alphabetic characters as tokens.

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.
  - Words in meta-tag
- Simplest approach is to exclude all HTML tag information (between “<“ and “>”) from tokenization. But it may lose important information.

Stopwords Removal

- 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. One may find different set of stopwords from different sources.
- 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 “compute”
- Correct morphological analysis is language specific and can be complex.
- Stemming “blindly” strips off known affixes (prefixes and suffixes) in an iterative fashion.

Four Types of Strategies

- Affix removal :
  - Based on intuitive, heuristic approach. Four different algorithms. Porter's algorithm is the most popular one
- Table look-up :
  - Look up the stem of a word from a table. Tables would be too big and difficult to construct.
- Successor variety :
  - Based on determining morpheme boundaries, complicated
- N-grams:
  - Based on term clustering, rather than stemming

Porter's Algorithm

- 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.
- Not recognize all morphological derivations.
- Appendix A of our text has a description of the algorithm.

Porter's Algorithm -- Basic Idea

- Uses a list suffix list to strip suffixes.
- The idea is to apply a set of rules to the suffixes of the words in the text..
- For example: four rules to remove plural form, select the rule with longest suffix
  - sses $\rightarrow$ ss;
  - ies $\rightarrow$ I;
  - ss $\rightarrow$ ss;
  - s $\rightarrow$ NULL;
- It makes the word *stresses* into *stress*

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.

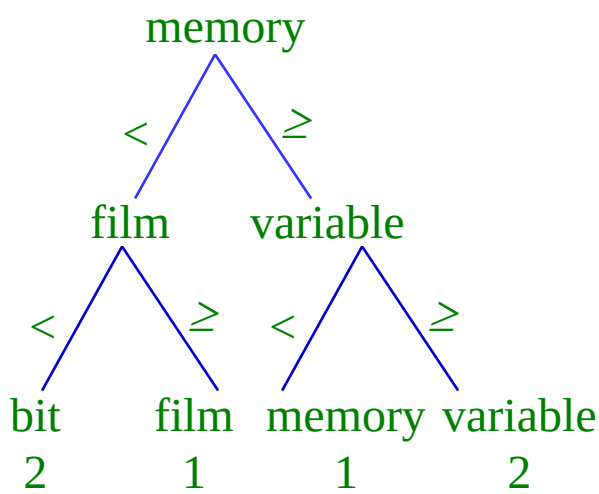
Sparse Vectors as Lists

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



Balanced Binary Tree

Sparse Vectors as Trees (cont.)

- Space overhead for tree structure: ~2n nodes.
- O(log n) time to find or update weight of a specific token.
- O(n log n) time to construct vector.
- Need software package to support such data structures.

Sparse Vectors as HashTables

- Store tokens in hashtable, with token string as key and weight as value.
  - Storage overhead for hashtable ~1.5*n*.
  - Table must fit in main memory.
  - Constant time to find or update weight of a specific token (ignoring collisions).
  - O(*n*) time to construct vector (ignoring collisions).

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 hash table, a sorted array, or a tree-based data structure (trie, B-tree).
- Critical issue is logarithmic or constant-time access to token information.

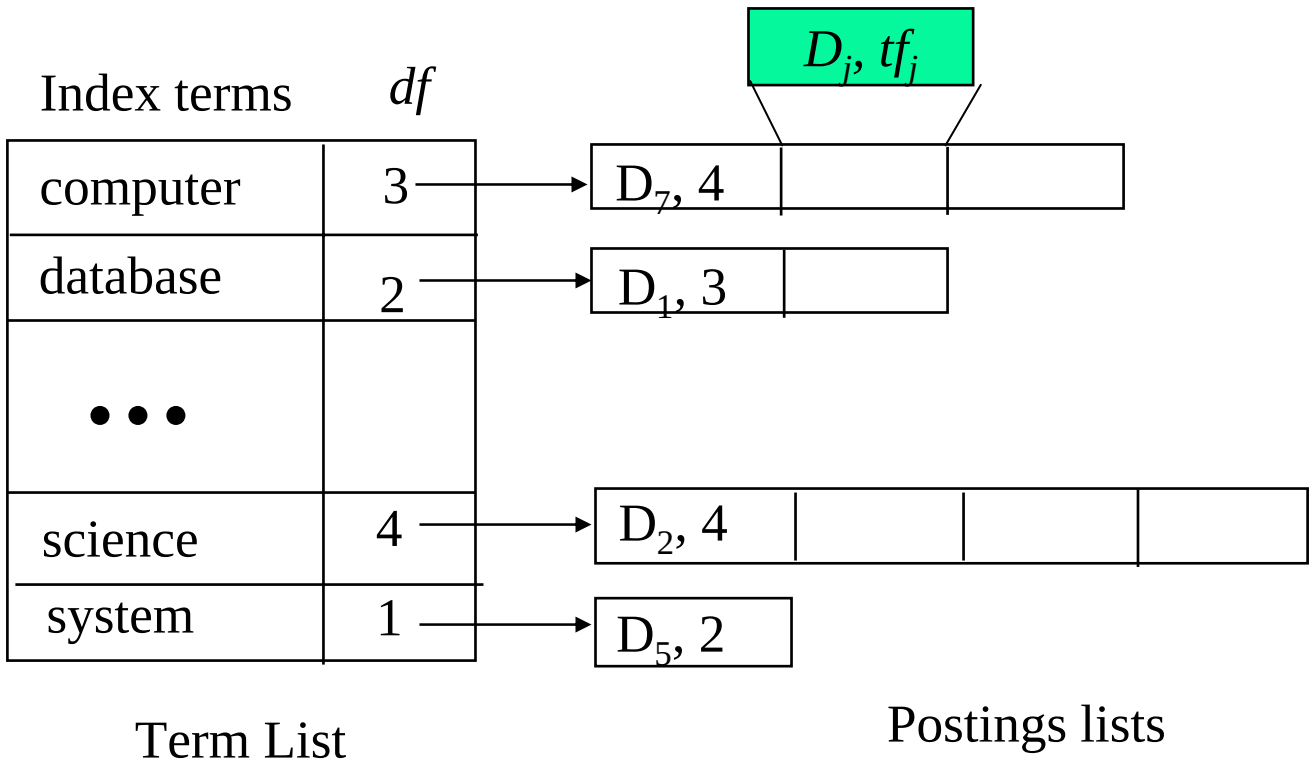
Inverted Index: Assumption

- Query will happen frequently
  - Find all documents that contain term *t*
- Delete will be rare
  - Delete document 52
- Update will be rare
  - Correct the spelling of term *t* in document 52
- Add will not happen too often
  - Add new document

Inverted Index: Basic Structure

- Term list: a list of all terms
- Document node: a structure that contains information such as term frequency, document ID, and others
- Posting list: for each term, a list containing document node for each document in which the term appears

Inverted Index



Creating an Inverted Index

```
Create an empty index term list I;
For each document, D, in the document set V
    For each (non-zero) token, T, in D:
        If T is not already in I
            Insert T into I;
        Find the location for T in I;
        If (T, D) is in the posting list for T
            increase its term frequency for T;
        Else
            Create (T, D);
            Add it to the posting list for T;
```

Computing IDF

Let N be the total number of documents;

For each token, T, in I:

Determine the total number of documents, M,

in which T occurs (the length of T's posting list);

Set the IDF for T to log(N/M);

Note this requires a second pass through all the tokens after all documents have been indexed.

Document Vector Length

- 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.

Computing Document Vector Lengths

Assume the length of all document vectors (stored in the DocumentReference) are initialized to 0.0;

For each token T in I:

Let, *idf*, be the IDF weight of T;

For each TokenOccurence of T in document D

Let, C, be the count of T in D;

Increment the length of D by $(idf*C)^2$;

For each document D in the document set:

Set the length of D to be the square-root of the current stored length;

Time Complexity of Indexing

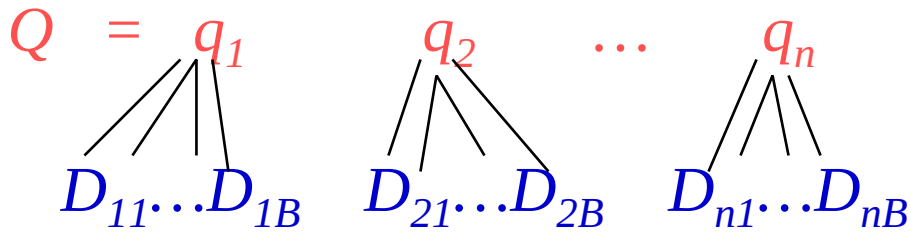
- Complexity of creating vector and indexing a document of n tokens is $O(n)$.
- So indexing m such documents is $O(m\ n)$.
- Computing token IDFs for a vocabularly V is $O(|V|)$.
- Computing vector lengths is also $O(m\ n)$.
- Since $|V| \leq m\ n$, complete process is $O(m\ n)$, which is also the complexity of just reading in the corpus.

Retrieval with an Inverted Index

- Tokens that are not in both the query and the document do not effect 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.

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 N documents, O(|V| N), because |Q| << |V| and B << N.

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.

Inverted-Index Retrieval Algorithm

Create a vector, Q, for the query.

Create empty HashMap, R, to store retrieved documents with scores.

For each token, T, in Q:

    Let idf be the IDF of T, and K be the count of T in Q;

    Set the weight of T in Q: W = K * idf;

    Let L be the list of TokenOccurences of T from I (term list);

    For each TokenOccurence, O, in L:

        Let D be the document of O, and C be the count of O (tf of T in D);

        If D is not already in R (D was not previously retrieved)

            Then add D to R and initialize score to 0.0;

        Increment D's score by W * idf * C; (product of T-weight in Q and D)

Retrieval Algorithm (cont)

Compute the length, L, of the vector Q (square-root of the sum of the squares of its weights).

For each retrieved document D in R:

Let S be the current accumulated score of D;

(S is the dot-product of D and Q)

Let Y be the length of D as stored in its DocumentReference;

Normalize D's final score to S/(L * Y);

Sort retrieved documents in R by final score and return results in an array.

User Interface

Until user terminates with an empty query:

Prompt user to type a query, Q.

Compute the ranked array of retrievals R for Q;

Print the name of top N documents in R;

Until user terminates with an empty command:

Prompt user for a command for this query result:

1) Show next N retrievals;

2) Show the Mth retrieved document;