Com S 435/535: Notes VIII

1 Information Retrieval

Information retrieval concerns with the following question: Given a collection of documents and a query q, how can you find the documents that are most relevant to the query q? We will study two models that enable efficient retrieval. Boolean model and Vector Space Model.

Let $D = \{D_1, D_2, \cdots, D_n\}$ be the document collection. We first do some preprocessing and extract relevant "features" of documents. We use the extracted features of documents are used to retrieve documents that relevant to a query. An obvious choice for feature is "word". However, some words such as "the", "a", and "an" appear in every document and they do not contribute much to the similarity of documents. Such words are called *stop words*. Thus the stop words are not features of any document. Thus, we will first remove all stop words from every document in our document collection. Then, we can convert all characters to lower case (or upper case) as similarity of documents should not be case-sensitive. If needed (depending on application), we can do further preprocessing of documents. This includes: removing punctuation symbols, replacing words with synonyms, converting plural nouns into singular nouns (for example: "problems" to "problem") and stemming. After this preprocessing is performed each remaining word in the document collection is a *term*. Let $T = \{t_1, t_2, \cdots, t_M\}$ be the collection of all terms that appear in all documents.

Given a query q consisting of terms $t_{i_1}, t_{i_2}, \cdots t_{i_\ell}$, we can retrieve relevant documents as follows: For each document search if all terms of the query appear in the document or not. Retrieve all documents that contain all terms of the query. However, this process is very inefficient. We will build a data structure that helps us do this efficiently. The central data structure in information retrieval is that of *inverted index*. An inverted index is like the index that you see at the end of a text book–it lists some words that appear in the text and for each word the index lists all pages in which that word appear. The basic principle behind inverted index is exactly the same—For each term in T, maintain a list of documents in which that term appears.

An inverted index has two parts, Dictionary and Postings. The terms are stored in a dictionary. For each term t in the dictionary we have a list that records which documents in which t appears. Each item in this list (typically a document), is called a posting and the list is called *postings list*. The collection of all postings is list is called *postings*.

Given a term t, we can retrieve all documents that contain t, by simply searching the dictionary for the term t and retrieve postings list associated with t. The boolean model can handle boolean queries. Suppose that the query q is $t_{i_1} \wedge t_{i_2}$. Then retrieve the postings list P_1 associated with the term t_{i_1} and the postings list P_2 associated with the term t_{i_2} and retrieve all documents that are in the intersection of P_1 and P_2. If the query is $t_{i_1} \vee t_{i_2}$, then retrieve all documents in the union of P_1 and P_2. Clearly, this method can be extended to handle queries with more complicated boolean expressions consisting of multiple terms.

What data structures need to be used so that the above retrieval process can be done efficiently. Given a term t, we need to search in the dictionary for the term t. Also, as new documents are added to the documents collection, we need to add new terms to the dictionary. Thus we need a data structure that can handle *search* and *add* for an element efficiently. Obvious choices are that of *hash table*, *balanced binary trees.*

Given two postings P_1 and P_2, we can compute the intersection/union in time $O(|P_1||P_2|)$ time. However, if P_1 and P_2 are sorted, then intersection/union can be done in time $O(|P_1|+|P_2|)$ time. Thus postings list are maintained as *sorted lists.*

Consider the query *hash table*, ideally we would like retrieval documents in which the phrase *hash table* appears rather than documents that contain the words *hash* and *table* in different contexts. The inverted index that we built is not capable of handling such "phrase queries". We can augment the inverted index with additional information to handle such phrase queries. The *positional index* is the most commonly used index to handle phrase queries. In a positional index, for each term in the dictionary, each entry in the postings list is of the form: $\langle d, position1, position2, \cdots \rangle$, where $position1, position2 \cdots$ are the positions (word position) at which the term appears in the document d. If a phrase query is of the form $t_{i_1} t_{i_2}$, then we retrieve documents in which the terms t_{i_1} and t_{i_2} are close to each other. This idea can be extended to queries consisting of multiple terms.

2 Vector Space Model

The major disadvantage of the boolean retrieval model is that there is no way to *rank the documents according to their relevance to the query.* Vector Space Model addresses this issue by assigning a weight to each term-document pair. Let t be a term and d. How much relevant is the term t to the document d? Obviously if t does not appear in the document, then its relevance is zero. Given a term t and a document d, let tf_{td} denote the number of times term t appears in document d, and let df_t be the number of documents in which t appears. Let N denote the total number of documents in the collection. Then $Weight(t,d)$ is a function of TF_{td} and IDF_t where $IDF_t = N/df_t$. Few common choices for the weight function are

- $Weight(t,d) = w_{td} = \log_2[1 + tf_{td}] \times \log_{10}[\frac{N}{df_t}]$
- $Weight(t,d) = w_{td} = tf_{td} \times \log_{10}[\frac{N}{df_t}]$
- $Weight(t,d) = w_{td} = tf_{td} \times \frac{N}{df_t}$

Recall that $T = \{t_1, \cdots, t_M\}$ be the collection of all terms in the document collection. Now given a document d, we can view it as the following vector in M-dimensional space:

$$v(d) = \langle w_{t_1 d}, w_{t_2 d}, \cdots, w_{t_M d} \rangle$$

Similarly given a query q, we can view it as a (short) document and as the following vector

$$v(q) = \langle w_{t_1 q}, w_{t_2 q}, \cdots, w_{t_M q} \rangle$$

Now given a document d and a query q, we define a similarity score for q with d as the cosine of the angle between the vectors $v(q)$ and $v(d)$. More formally,

$$score(q,d) = \frac{v(q) \cdot v(d)}{||v(q)||v(d)||}$$

where $v(q) \cdot v(d)$ denotes the dot product of $v(q)$ and $v(d)$ and $||(v(q)||$ and $||v(d)||$ are lengths of the vectors $v(q)$ and $v(d)$.

Now given a query q and an integer k, we can retrieve top k documents whose score with q is the highest. To enable this retrieval efficiently, we further augment our inverted index as follows: Each dictionary entry is a 3-tuple consisting of a term t, df_t and a pointer to the postings list. Each entry of the postings list a 2-tuple consisting of a document d in which t appears and tf_{td}.

Given a query q an an integer k, we can retrieve the top k documents that match the query by the following algorithm.

1. Document collection $d_1, \cdots, d_N$.
2. Preprocessing Step1: Build the inverted index
3. Preprocessing Step2: Initialize $Length[i]$ to $||v(d_i)||$ for $1 \leq i \leq N$.
4. Input: Query q.
5. Initialize $S = \emptyset$. Each member of S is of the form $\langle d, s \rangle$ (where d is document ID and s is a real number).
6. For each term $t \in q$
   (a) Fetch postings list $postings(t)$ for t.
   (b) For each tuple $\langle d, tf_{td} \rangle$ in $postings(t)$
      - Compute $weight(t, d)$
      - If S does not have tuple of form $\langle d, \cdot \rangle$, Place $\langle S, weight(t, d) \times weight(t, q) \rangle$ in S.
      - If S has a tuple of form $\langle d, s \rangle$, then update $\langle d, s \rangle$ to $\langle d, s + (weight(t, d) \times weight(t, q)) \rangle$.
7. Compute $||v(q)||$.
8. For every $\langle d, s \rangle \in S$ do
   - Update $\langle d, s \rangle$ to $\langle d, \frac{s}{||V(d)|| \times ||v(q)||} \rangle$
9. Return top k components of S (based on the cosine-scores).