Indexing

April 16, 2015

1 Documents and query representation

* term incidence matrix
    * About retrieval Models

1.1 bag of words representation

* TF, DF, DLength, AVG(DLength), V, N, IDF
    * subsection what and how to get from index
    Q: can i write from other places like wiki or something

2 Preprocessing

In Information Retrieval, it is often necessary to interpret natural text where a a large amount of text has to be interpreted, so that it is available as a full text search and is represented efficiently in terms of both space (document storing) and time (retrieval processes) requirements.

It can also be regarded as : process of incorporating a new document into an information retrieval system. The document would go through the phases of Tokenization where the documents is broken down into tokens, after which stopwords would be removed, thereafter stemming of the tokens takes place : which stems each word into its root words and finally its all combined with term position to make an Index. We'll see each of the processes as follows.

2.1 Tokenization

Tokenization is the task of chopping it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation. Input: "John_DavenPort #person 52 years_old #age"

John	DavenPort	person	52	years	old	age

2.2 Stopwords

Stopwords refer to the words that have no meaning for "Retrieval Purposes". E.g.

· **Articles** : a, an, the, etc.

· **Prepositions** : in, on, of, etc.

· **Conjunctions** : and, or, but, if, etc

· **Pronouns** : I, you, them, it, etc

· **Others** : some verbs, nouns, adverbs, adjectives (make, thing, similar, etc.).

Stopwords can be up to 50% of the page content and not contribute to any relevant information w.r.t. retrieval process. Removal of these can improve the size of the index considerably. Sometimes we need to be careful in terms of words in phrases! e.g.: Library of Congress, Smoky the Bear!

Word	Occurrences	Percentage
the	8,543,794	6.8
of	3,893,790	3.1
to	3,364,653	2.7
and	3,320,687	2.6
in	2,311,785	1.8
is	1,559,147	1.2
for	1,313,561	1.0
that	1,066,503	0.8
said	1,027,713	0.8

2.3 Stemming

Stemming is commonly used in Information Retrieval to conflate morphological variants. Typical stemmer consists of collection of rules and/or dictionaries. Similar approach may be used in other languages too!

e.g.: The following stem to the word as shown below:

servomanipulator ← servomanipulators servomanipulator
logic ← logical logic logically logics logicals logicial logicially
login ← login logins
microwire ← microwires microwire
knead ← kneaded kneads knead kneader kneading kneaders

2.3.1 Stemming Example

Original text	Porter Stemmer
Document will describe marketing strategies carried out by U.S. companies for their agricultural chemicals, report predictions for market share of such chemicals, or report market statistics for agrochemicals, pesticide, herbicide, fungicide, insecticide, fertilizer, predicted sales, market share, stimulate demand, price cut, volume of sales	market strateg carr compan agricultur chemic report predict market share chemic report market statist agrochem pesticid herbicid fungicid insecticid fertil predict sale stimul demand price cut volum sale

2.3.2 Porter Stemmer

There are multiple approaches for Stemming like using Brute force look up, Suffix - affix stripping, Part-of-speech recognition, Statistical algorithms (n-grams, HMM), etc.

Steps involved in Porter Stemmer:

1. Gets rid of plurals and -ed or -ing suffixes. e.g: Operatives → operative
2. Turns terminal y to i when there is another vowel in the stem. e.g: Coolly → coolli
3. Maps double suffixes to single ones: -ization, -ational, etc. e.g: Operational → operate
4. Deals with suffixes, -full, -ness etc.e.g: Authenticate → authentic
5. Takes off -ant, -ence, etc. e.g: Operational → operate → oper
6. Removes a final -e. e.g: Parable → parabl

2.3.3 Porter Stemming Example

· **Step 1** : Semantically → semantically

· **Step 2** : Semantically → semantically → semanticalli

· **Step 3** : Semantically → semantically → semanticalli → semantical

· **Step 4** : Semantically → semantically → semanticalli → semantical → semantic

· **Step 5** : Semantically → semantically → semanticalli → semantical → semantic → semant

· **Step 6** : Semantically → semantically → semanticalli → semantical → semantic → semant → semant

2.4 Term Positions

Term Positions specify the document id along with the position of occurrences in that specific document. Lets try and see that with an example. We have three documents of id1, id2 and id3:

id1 →	Web 1	Mining 2	is 3	useful 4

id2 →	usage 1	mining 2	applications 3

id3 →	Web 1	structure 2	mining 3	studies 4	the 5	web 6	hyperlink 7	structure 8

The below figure summarizes the term positions:

Applications:	id_2	Applications:	<id_2, 1, [3]>
Hyperlink:	id_3	Hyperlink:	<id_3, 1, [7]>
Mining:	id_1, id_2, id_3	Mining:	<id_1, 1, [2]>, <id_2, 1, [2]>, <id_3, 1, [3]>
Structure:	id_3	Structure:	<id_3, 2, [2, 8]>
Studies:	id_3	Studies:	<id_3, 1, [4]>
Usage:	id_2	Usage:	<id_2, 1, [1]>
Useful:	id_1	Useful:	<id_1, 1, [4]>
Web:	id_1, id_3	Web:	<id_1, 1, [1]>, <id_3, 2, [1, 6]>

3 Index Construction

A reasonably-sized index of the web contains many billions of documents and has a massive vocabulary. Search engines run roughly 105 queries per second over that collection. We need fine-tuned data structures and algorithms to provide search results in much less than a second per query. O(n) and even O(log n) algorithms are often not nearly fast enough. The solution to this challenge is to run an inverted index on a massive distributed system.

Text search has unique needs compared to, e.g., database queries, and needs its own data structures' primarily, the inverted index and we'll be discussing primarily the inverted index.

· **Forward Index** : A forward index is a map from documents to terms (and positions). These are used when you search within a document.

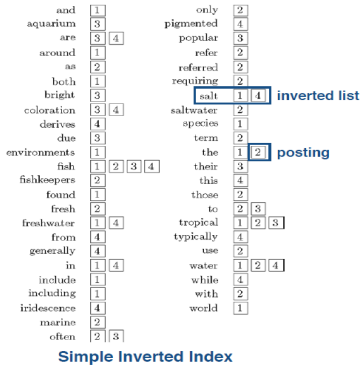
· **Inverted Index** : An inverted index is a map from terms to documents (and positions). These are used when you want to find a term in any document.

3.1 Inverted lists and catalog/offset files

In an inverted index, each term has an associated inverted list. At minimum, this list contains a list of identifiers for documents which contain that term. Usually we have more detailed information for each document as it relates to that term. Each entry in an inverted list is called a posting. Document postings can store any information needed for efficient ranking. For instance, they typically store term counts for each document - tf $_{w,d}$.

Depending on the underlying storage system, it can be expensive to increase the size of a posting. It's important to be able to efficiently scan through an inverted list, and it helps if they're small.

Inverted Indexes are primarily used to allow fast, concurrent query processing. Each term found in any indexed document receives an independent inverted list, which stores the information necessary to process that term when it occurs in a query. Next, we'll see how to process proximity queries, which involve multiple inverted lists.



Simple Inverted Index

3.2 Memory Structure, and limitations

Given a collection of documents, how can we efficiently create an inverted index of its contents? The basic steps are:

1. Tokenize each document, to convert it to a sequence of terms.
2. Add doc to inverted list for each token.

This is simple at small scale and in memory, but grows much more complex to do efficiently as the document collection and vocabulary grow.

The basic indexing algorithm will fail as soon as you run out of memory. To address this, we store a partial inverted list to disk when it grows too large to handle. We reset the in-memory index and start over. When we're finished, we merge all the partial indexes. The partial indexes should be written in a manner that facilitates later merging. For instance, store the terms in some reasonable sorted order. This permits merging with a single linear pass through all partial lists.

There are multiple ways as given below by which a Inverted Index can be built.

3.2.1 Option 1: Multiple Passes

Making multiple passes through the document collection is the key here. In each pass, we create the inverted lists for the next 1,000 terms, each in its own file. At the end of each pass, we concatenate the new inverted lists onto the main index file. This process is easy to concatenate the inverted files, but have to manage the catalog/offsets files.

3.2.2 Option 2: Partial inverted lists

Create partial inverted lists for all terms in a single pass through the collection is what happens in this process. As each partial list is filled, append it to the end of a single large index file. When all documents have been processed, run through the file a term at a time and merge the partial lists for each term. This second step can be greatly accelerated if you keep a list of the positions of all the partial lists for each term in some secondary data structure or file.

3.2.3 Option 3: preallocate the right amount of space

We could have dis-contiguous postings also. Here, we lay our index file as a series of fixed-length records of, say, 4096 bytes each. Each record will contain a portion of the inverted list for a term. A record will consist of a header followed by a series of inverted list entries. The header will specify the term_id, the number of inverted list entries used in the record, and the file offset of the next record for the term. Records are

written to the file in a single pass through the document collection, and the records for a given term are not necessarily adjacent within the index.

3.3 Updating an inverted index

If each term's inverted list is stored in a separate file, updating the index is straightforward: we simply merge the postings from the old and new index. However, most filesystems can't handle very large numbers of files, so several inverted lists are generally stored together in larger files. This complicates merging, especially if the index is still being used for query processing. There are ways to update live indexes efficiently, but it's often simpler to simply write a new index, then redirect queries to the new index and delete the old one.

4 Proximity Search

Proximity is the search technique used to find two words next to, near, or within a specified distance of each other within a document. Using such search operators may result in more satisfactory results that are more relevant to the research needs than by just typing in desired keywords. Some commands also control the terms' order of appearance. Desired words can be in any order, a specific order, or within a certain range of each other. The score function is based on the min span

$$\lambda^{\frac{s-k}{k}}$$

where λ is a constant base, around 0.8, s is the min span found and k is the length of ngram matched.

e.g.: If ngram = "atomic bomb world war two" matches on a min span of 8 words, the score will be

$$\lambda^{\frac{8-5}{5}} = 0.80.6$$

Refer `http://stevekrenzel.com/articles/blurbs` for minimum span algorithm.

5 Other things to store in the index

Things which you can store in an index other than just postings list:

· **Meta Information** : Various other meta information such as summary term information, summary document information, etc. can be kept in the index which can be further used for efficient retrieval in meta data queries.

· **Document Meta-data** : The following information can be kept in the index too viz. language, geographic region, file format, date published, domain, licensing rights, etc.

· **TimeStamp** : Last updated timestamp or last crawled timestamp.

· **Header Information** : Like the number in the header of a posting list can indicate the number of posting units it contains and so on.

A: index storage organization (header) skives_1 : add lempel zif

6 Compression

*probabilities as matching evidence

6.1 Basics of Compression, Entropy

6.1.1 Entropy in events

Entropy is simply the average(expected) amount of the information from the event.

$\sum_{i=1}^{n} p_i log_p(p_i)$ where n = number of different outcomes

e.g: You have 4 Red balls, 2 yellow and 3 green balls in a bin. In this example there are three outcomes possible when you choose the ball : it can be either red, yellow, or green. Thus n = 3. So the equation will be following.
Entropy = - (4/9) log(4/9) + -(2/9) log(2/9) + - (3/9) log(3/9)
Entropy = 1.5304755
Therefore, you are expected to get 1.5304755 information each time you choose a ball from the bin.

6.1.2 Entropy in information

Entropy is the expected number of bits that is required to encode each character over the distribution over the characters.

Suppose now that we are in the following setting:

· : the file contains n characters

· : there are c different characters possible

· : character i has probability p(i) of appearing in the file

When we pick a file according to the above distribution, very likely there will be about p(i) · n characters equal to i. The files with these "typical" frequencies have a total probability about p = $\prod_{i=a} p(i)^{p(i)\Delta n}$ of being generated. Since files with typical frequencies make up almost all the probability mass, there must be about 1/p = $\prod_{i=a}(1/p(i))^{p(i)\Delta n}$ files of typical frequencies.
We then expect the encoding to be of length at least $log_2 \prod_{i=a} i(1/p(i))^{p(i)\Delta n} = \text{n} \sum_{i=1} p(i) log_2(1/p(i))$.
So Entropy = $\sum_{i=1} p(i) log_2 1/(p(i))$.

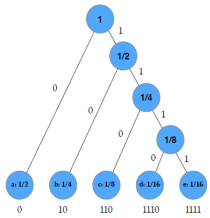
6.2 Restricted Variable-Length Codes

6.3 Huffman codes

In an ideal encoding scheme, a symbol with probability pi of occurring will be assigned a code which takes log(pi) bits. The more probable a symbol is to occur, the smaller its code should be. By this view, UTF-32 assumes a uniform distribution over all unicode symbols; UTF-8 assumes ASCII characters are more common. Huffman Codes achieve the best possible compression ratio when the distribution is known and when no code can stand for multiple symbols.

Symbol	**p**	**Code**	**E[length]**
a	1/2	0	0.5
b	1/4	10	0.5
c	1/8	110	0.375
d	1/16	1110	0.25
e	1/16	1111	0.25

Huffman Codes are built using a binary tree which always joins the least probable remaining nodes.



1. Create a leaf node for each symbol, weighted by its probability.
2. Iteratively join the two least probable nodes without a parent by creating a parent whose weight is the sum of the childrens weights.
3. Assign 0 and 1 to the edges from each parent. The code for a leaf is the sequence of edges on the path from the root.

Huffman codes achieve the theoretical limit for compressibility, assuming that the size of the code table is negligible and that each input symbol must correspond to exactly one output symbol. Other codes, such as Lempel-Ziv encoding, allow variable-length sequences of input symbols to correspond to particular output symbols and do not require transferring an explicit code table. Compression schemes such as gzip are based on Lempel-Ziv encoding. However, for encoding inverted lists it can be beneficial to have a 1:1 correspondence between code words and plaintext characters.

7 Encoding integers

We can compress integers using simple encoding as explained below but the best encoding depends on how values are distributed. Keeping documents as ascending order and the storing the as sequence of gaps.
e.g: document list : 3, 5, 20, 21, 23, 76, 77, 78 → becomes: 3, 2, 15, 1, 2, 53, 1, 1 .
We'll discuss a few bit level encodings and byte level encodings.

7.1 Bit level encoding

· **Unary** : N '1's followed by a '0'

· **Gamma** : $\log_2(N)$ in unary, then floor($\log_2(N)$) bits

· **Rice K** : floor(N / 2^K) in unary, then N mod 2^K in K bits. In Golomb codes the base power is 2.

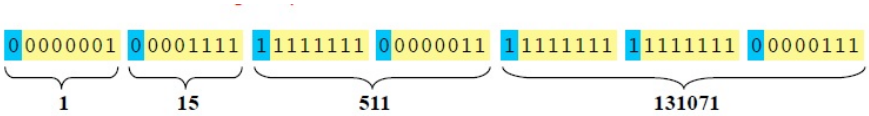
· **Huffman-Int** : like Gamma, except $\log_2(N)$ is Huffman coded instead of encoded w/ Unary

7.2 Byte-aligned encodings

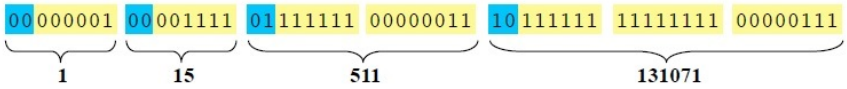
Variable byte (VB) encoding uses an integral number of bytes to encode a gap. The last 7 bits of a byte are "payload" and encode part of the gap. The first bit of the byte is a continuation bit . It is set to 1 for the last byte of the encoded gap and to 0 otherwise. To decode a variable byte code, we read a sequence of bytes with continuation bit 0 terminated by a byte with continuation bit 1. We then extract and concatenate the 7-bit parts.

The idea of VB encoding can also be applied to larger or smaller units than bytes: 32-bit words, 16-bit words, and 4-bit words or nibbles

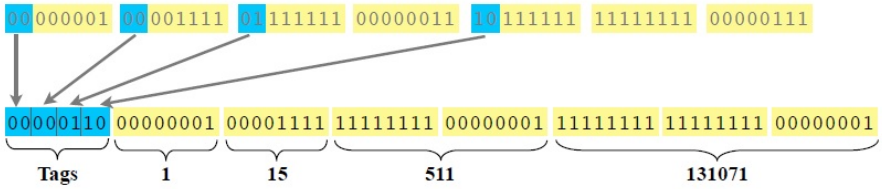
· **7 bits** : It uses 7 bits per byte with continuation bit. Decoding here requires lots of branches/shifts/masks.



· **2 bits** : Encode byte length as low 2 bits, it results in fewer branches, shifts, and masks. But this method is limited to 30-bit values, still some shifting to decode.



· **Group of 4** :Encode groups of 4 values in 5-17 bytes, pull out 4 2-bit binary lengths into single byte prefix.



Decoding works by loading the prefix byte and use value to lookup in 256-entry table.