Information Retrieval Notes

Artem Vasenin

Last updated April 10, 2017

Contents

Chapter 1

Boolean Retrieval

1.1 Index

Any query is posed as a boolean expression of terms. First we create an index of words that occur in each text, this is done offline. E.g.

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello
Antony	1	1	0	0	0
Brutus	1	1	0	1	0
Caesar	1	1	0	1	1
Calpurnia	0	1	0	0	0
Cleopatra	1	0	0	0	0
mercy	1	0	1	1	1
worser	1	0	1	1	1

When we compute results of a query we first invert corresponding to NOT terms and then compute a bitwise AND and OR between vectors for each term. A few observations:

- The matrix is very sparse.
- Doesn't support more complex operations, such as proximity search.

1.2 Inverted Index

To make storing information more efficient we can use inverted matrix which only documents things that occur. The inverted index is a dictionary, with terms for keys and *posting lists* for values. A posting list is a sorted list which documents which documents the term occurs in.

Now to compute an AND query we compute and intersection of posting lists. This can be done in $O(n)$ where n is the number of documents in collection, in practice much faster.

When computing a conjunctive query with more than two terms we process them based on length of posting lists, in ascending order. For disjunctive terms we can estimate the size of result using the sum of sizes of disjuncts.

Chapter 2

Indexing

The major steps in inverted index constructions are:

1. Collect the documents to be indexed.
2. Tokenize the text.
3. Perform linguistic preprocessing of tokens.
4. Index the documents that each term occurs in.

2.1 Skip lists

When we construct an inverted index we can use either a linked list or a variable length array. In the case we use a linked list there is a natural optimisation we can perform. To create a skip list we can link together every n elements. This allows us to compare to lists quicker.

Size of n, number of elements skipped, presents a trade-off: number of items skipped vs. frequency that skip can be taken. Usually $n = \sqrt{L}$ where L is the length of the list.

N.B. Skip lists used to help a lot, but with today's fast CPUs they don't help that much anymore.

2.2 SPIMI

During indexing large collections we can't keep and sort all postings in-memory. We cannot sort very large records on disk either (too many disk seeks, expensive). We can use **Block-Based** sorting algorithm. Two key ideas:

- Generate separate dictionaries for each block.
- Accumulate postings in posting lists as they occur.

Using those two ideas we can generate complete invert index for each block and then merge blocks at the end.

2.3 Document and term normalisation

A lot of words used in documents have similar meaning and should be tokenised the same, e.g. multiples, capitalisation, misspelling, etc. Most of the time we also need to prepare a document before it can be tokenised, we need to consider:

- Compression and binary representation.
- Format (excel, pdf, latex, etc.)
- Character set
- Language the document is written in.

Each of these is a statistical classification problem. Also deciding what is a "document" is not trivial.

Tokenisation It is frequently difficult to decide how to tokenise the text. Splitting text into words is non trivial, so is parsing numbers. Text can use different scripts (Japanese) and can be written in different directions (Arabic).

There are several useful techniques:

Case folding Reduce the text to lowercase.

Stop words Frequent words that don't matter can be excluded (a, the, an, etc).

Equivalence classing Words with the same meaning should result in the same token (car = automobile).

Lemmatisation Reduce inflections, derivations to base form (am, are, is $\rightarrow$ be).

Stemming Chopping the end of word off (cheaper lemmatisation).

2.4 Phrase Queries

Need to answer some queries as a phrase, e.g. “Cambridge University”, but sentences line “The Duke of Cambridge recently went for a term-long course to a famous university” should not match. Therefore it is no longer sufficient to store docIDs in posting lists. Two ways to extend the inverted index:

- Biword index
- Positional index

Biword index Index every consecutive pair of terms in the documents as a phrase. Two-word phrases can now easily be answered. For longer phrases use a conjunction, e.g. “cambridge university west campus” becomes “cambridge university AND university west AND west campus”. Need to do post filtering to check whether the whole phrase is actually used. **Issues:** a lot of false positives and index blowup.

Positional index In addition to storing the docID, also store position in the document. We can now check phrases using position. It can also be used for proximity search.

N.B. We want to return the actual matching position, not just the document.

Combination Many biwords are extremely frequent (Michael Jackson, Los Angeles). For such biwords using a simple positional index is slow. Therefore to improve performance we can include frequent biwords as vocabulary terms.

2.5 Term Vocabulary

Can we estimate how many distinct words are used in a collection? Yes we can, *Heap’s Law*: $M = kT^b$ where M is the size of vocabulary, T is the number of tokens in the collection. Typical values for k and b are : $30 \leq k \leq 100$ and $b \approx 0.5$. Heaps’ Law is an empirical law.