Information Retrieval and Web Search
Lecture 04

Lecture by Dr. Inbal Budowski-Tal
Typeset by Steven Karas

2018-04-25
Last edited 20:41:49 2018-04-25

Disclaimer These lecture notes are based on the lecture for the course Information Retrieval and Web Search, taught by Dr. Inbal Budowski-Tal at IDC Herzliyah in the spring semester of 2017/2018. Sections may be based on the lecture slides written by Dr. Inbal Budowski-Tal.

Agenda

- Context-sensitive correction
- Soundex

Note regarding Levenshtein Distance This is an important dynamic programming algorithm and we should understand the algorithm very well. It may feature prominently either on a future homework or the exam.

1 Dictionaries

There are several approaches. The three basic ones are hash †ables, sorted fixed-width entries, and trees (binary or B-variants).

2 Wildcard Queries

Wildcard queries are a form of tolerant retrieval. There are three standard forms of wildcard queries: prefix, infix, and suffix. Infix wildcards can either be performed as the intersection of prefix and suffix queries, or use a permuterm index.

2.1 Permuterm Index

To construct a permuterm index, generate all rotations of a term, using a symbol outside the term alphabet to denote the end of the word:
`HELLO` $\rightarrow$ `hello$ ello$h llo$he lo$hel o$hell`

- when searching for `X`, query for `X$`
- when searching for `X*`, query for `X*$`
- when searching for `*X`, query for `X$*`
- when searching for `*X*`, query for `X*`
- when searching for `X*Y`, query for `Y$X*`

2.2 k-gram indexes

A k-gram index is more space efficient than an equivalent permuterm index, but requires some extra processing to query. To construct the index, we enumerate all the k-grams consisting of all subsequences of length k that occur in a term (including the inserted boundary markers) This index maps from the k-grams to a list of terms that contain the subsequence.
`April is the cruelest month` $\rightarrow$ `$a ap pr ri il l$ $i is s$ ...`

Wildcard queries first generate a list of terms, which need to be filtered for false positives (e.g. `mon*` will query for `$m, mo, on`, which may produce `moon`). Terms are then queried against the term-postings index.

3 Spelling correction

Spelling correction has two main applications: correcting documents being indexed, and correcting user queries. An isolated word approach considers the spelling of each word independently of the rest of the tokens. A context-sensitive approach will consider nearby tokens as well, and may correct mistakes such as `the asteroid fell form the sky`. The reason we want to have spelling correction is usually due to OCR'd documents, which may have transcription errors.

Corrections can be sourced from a canonical dictionary, which may be too general purpose, or drawn from the term vocabulary itself (appropriately weighted). The candidate correction is typically defined as the term with the shortest edit distance from the misspelled token.

3.1 Edit distance

The **edit distance** between strings s_1, s_2 is the minimum number of operations to change s_1 into s_2. The specific operations that are considered is a detail of the specific distance metric.

More advanced approaches will weigh some operations as they may have been more likely.

3.1.1 Levenshtein Distance

Introduced by Levenshtein in 1966[1]. For the Levenshtein distance metric, the operations considered are insert, delete, and replace. There is an efficient dynamic programming algorithm that produces not only the distance, but also a possible sequence of operations.

A weighted approach for some replacements is a good idea for some applications where the distribution of errors is known. For example, reducing the distance between O and C for OCR'd text.

3.2 k-gram spelling correction

We can enumerate all the k-grams in the misspelled term, and threshold all resulting terms by the number of matching k-grams. The resulting list can then be sorted by edit distance and the lowest distance term returned as the correction.

3.3 Context sensitive correction

In this case, take the k-word queries, construct alternative queries based upon k-gram isolated term corrections and suggest the subqueries that have the most hits.

4 Soundex

Soundex is an approach for finding terms based on phonetic distance (as opposed to orthographic distance). The approach is rather simple: map each token into a 4-character reduced form, and construct an index over this reduced form.

Algorithm

1. Retain the first letter
2. Convert all of the following to 0: A, E, I, O, U, H, W, Y
3. Map the following letters to digits as follows:
    - B, F, P, V $\rightarrow$ 1
    - C, G, J, K, Q, S, X, Z $\rightarrow$ 2
    - D, T $\rightarrow$ 3
    - L $\rightarrow$ 4
    - M, N $\rightarrow$ 5
    - R $\rightarrow$ 6
4. Compress all consecutive identical digits
5. Remove all zeros, pad the tail with 3 zeros
6. Return the first 4 characters

Disadvantages It's not very useful for information retrieval, but is a good approach and can be adjusted/expanded for specific applications. Zobel and Dart in 1996 suggest some adjustment techniques for IR applications.

5 Index Construction

Many of the design considerations for IR systems are based upon hardware constraints. Disk access is typically more than an order of magnitude slower than memory access. Disk seek (for magnetic storage) is effectively idle time. Transferring one large sequential chunk is typically faster than many small chunks. Disk I/O is typically block-based, with typical block sizes ranging from 4KB to 8MB. IR systems typically have large memory spaces, and an order of magnitude more attached storage.

5.1 RCV1 dataset

Shakespeare's collected works are not sufficiently large to demonstrate many of the points in the course. As an example corpus, we will use the RCV1 dataset, which is a collection of all English newswire articles from 1995 and 1996.

Documents	$n = 800000$
Tokens per document	$L = 200$
Terms	$M = 400000$
bytes per token	including spaces, punctuation: 6
bytes per token	after preprocessing: 4
non-positional postings	$T = 10000000$
Average term frequency	400
positional postings	160mm

5.2 Sort-based Index Construction

As we build our index, we parse documents one at a time. The postings for a term are incomplete until the end. If we can fit all the postings in memory, then construction is trivial (in-memory sort). Even for a corpus such as RCV1, modern machines can easily fit the entire postings list in memory. However, modern collections are even larger and do not readily fit on COTS systems.

5.2.1 Block Sort-Based Indexing

Each posting takes 12 bytes: 4 - termID, 4 - docID, 4 - frequency. We can easily fit a few blocks in memory at a time, but we have many blocks. So we sort each block in memory, and then merge blocks and write the result to disk.

The two popular approaches to merging blocks are *k*-way merge, and log-merge[1].

BSBI

```
n = 0
until all documents have been processed:
  n += 1
  block = ParseNextBlock()
  BSBI-Invert(block)
  WriteBlockToDisk(block, $f_n$)

MergeBlocks($f_1$, ..., $f_n$; $f_{\text{merged}}$)
```

Choosing the block size is very important.

5.3 Distributed Indexing

Large IR systems use COTS hardware distributed across multiple datacenters and regions over the world. However, such systems can have individual machines fail or slow down. Typical approaches will assign a coordinator that will assign tasks to idle machines from a pool.

A diagram describing an example data flow can be found on slide 26. In this diagram, documents are split into chunks called splits that are handed to parser tasks that write postings to buckets called partitions (for example, writing all terms sorted lexicographically a-f into the first partition, etc.). An inverter task takes a partition, sorts it, and writes it to the postings lists.

MapReduce is a framework for writing/describing such distributed systems without needing to handle a lot of the plumbing.

[1]Both of these have more details in the course book

5.4 Dynamic Indexing

For practical IR systems, the collections tend to change as documents are added, removed, and modified. The simplest approach is to maintain a large index on disk, and keep a smaller auxiliary index in memory for new documents. Queries are executed across both, and results are merged. Deletions are handled by keeping an invalidation bit-vector for all documents, and filtering results by this bit-vector. The full index can be rebuilt periodically to prevent unsustainable growth of the auxiliary index.

However, in practice this approach requires frequent merges, and negatively impacts search performance during the merge. There are some strategies to mitigate this, such as storing each posting list as a separate append-only file. However, even these have limitations and there are better approaches.

5.4.1 Logarithmic Merge

Log merge amortizes the cost of merging indexes over time. By maintaining a series of indexes, each twice as large as the previous. Keep the smallest one in memory, and maintain the rest on disk. As the index in memory grows, write it to disk, repeatedly merging existing indexes into the next largest and writing the result to disk.

The exact algorithm can be found on slides 37.

An important thing to note is that to process a query, all indexes must be searched and results merged. There are $O(\log T)$ indexes.

6 Next Session

References

[1] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[2] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.