

COL764 Information Retrieval Assignment 1

Sanskar Khandelwal(2021CE10492)

31st August 2023

Contents

1 Introduction	1
2 Implementation Details	1
2.1 Pre-Processing the Data	1
2.2 Storing the data	2
2.3 Retrieval Steps:	3
2.4 Performance and Scoring	3

1 Introduction

TF-IDF stands for “Term Frequency — Inverse Document Frequency”. This is a technique to quantify a word in documents, we generally compute a weight to each word which signifies the importance of the word in the document and corpus. This method is a widely used technique in Information Retrieval and Text Mining.

$$tf_{i,j} = (1 + \log(f_{i,j}))$$

$$idf_i = \log(1 + \frac{N}{df_i})$$

$$\text{TF-IDF score} = tf_{i,j} * idf_i$$

2 Implementation Details

2.1 Pre-Processing the Data

This takes place in multiple stages due to the structure of this data. We use xml package in Python to efficiently process the training files and get data out by tags. As a first step in pre-processing, we remove certain punctuation from the full-text content of the files provided to us, which makes XML parsing easier and less ambiguous. With the help of XML element tree, we extract the contents of the “DOCID” and “CONTENT” contents (XML is a benchmarked faster parser than other HTML/XML parsers, like BeautifulSoup, and orders of magnitude faster than re-based searches).

After this step, the goal is to pre-process the CONTENT content as efficiently as possible. On an average, each document takes 1 second to go through the text and add them to the posting this. This is achieved by careful analysis of the bottlenecks and using C-based hashing functions for filtering and removal.

- Parsing: Analysis of dataset and extraction of TEXT tags. “re” based pattern matching proved extremely inefficient as compared to using HTML/XML parsers like “lxml” and “beautiful soup”. “lxml” was found to be slightly more efficient on comparison. The reason of this (as I understood it from the documentation) is that BeautifulSoup focuses more on functionality, like picking up data from malformed XML files or encoding detection. If we are more sure of the tags we want to work with, its always a better idea to use lxml’s superior parsing capabilities.

- **Punctuation:** For cleaning the text of unwanted punctuation and other symbols in our document, we do filtering for punctuation. There might be few problems such as U.S. — as “United States” being converted to “us” after the preprocessing. While periods and similar symbols should be replaced by an empty character, hyphens should usually be dealt little carefully and replaced by a space to avoid joining two unrelated words. Since the nouns are tagged using Location/Organisation/Person, we can safely eliminate these for now. We use the Python library `data = data.translate(str.maketrans('', '', '0() ,;+*"?$`.'))` to get rid of all the symbols and `data = data.replace(<!--(.*)-->,"")` to remove all comments. Note that this approach is faster than most 're' or 'regex'-based pattern matching methods by at least 10x times since they are written in C Python and use native C compilers. The only way to obtain more speedup is by writing custom C functions yourself.
- **Apostrophe:** When we remove the apostrophe punctuation first, it will convert words like don't to don't, and it is a stop word that won't be removed. So we are first removing the stop words, and then symbols, and finally stopwords because few words might still have an apostrophe which are not stop words.
- **Single characters:** Single characters are not very useful in knowing the importance of the document and a few final single characters might be irrelevant symbols, so it is always good to remove the single characters. Just like the stop words, these characters are overly present and don't add much value to the retrieval quality. If anything, they poison the TF-IDF scores of other words due to their relatively high frequency.

2.2 Storing the data

All data was stored in dictionary format in both .dict and .idx posting lists. The choice for dictionary data structure was based on a hashing-based lookup that would work in $O(1)$ time even for a large corpus collection, which is what we deal with here. So the data is stored in the format: `{<doc-id 1>: {<token 1>: <frequency in the doc>, <token 2>:}}`

Here is the actual first line of the .dict file:

```
{ "AP880403-0001": { "accordance": 1, "accounts": 1, "aged": 1, "along": 2, "amazed": 1, "among": 1,
"anyone": 2, "apostles": 1, "arms": 4, "associated": 1, "away": 2, "back": 1, "behold": 3, "beneath": 1,
"bethlehem": 1, ....
```

Similarly, the .idx file is stored as follows -

```
{<token 1>: {<doc-id 1 1>: <frequency in the doc>, <doc-id 2>: ....}}
```

Here is the actual first line of the .dict file:

```
"$doc size$": "$doc size$": 96, "Ababa": { "AP880403-0027": 1, "abandoned": { "AP880403-00261,
"AP880403-0040": 1, "AP880403-0080": 1, "abandoning": { "AP880403-0040": 1, "abated": { "AP880403-0078": 1,
"abc O": { "AP880403-0053": 1, "AP880403-0073": 1 }
```

- All information is compacted in the .idx file so the .dict file is useless for the TF-IDF calculation. This was a design choice since for large enough files, loading both the posting lists for the retrieval stage could have caused memory errors for machines with lower RAMs.
- The words with the named entity tags (Person: P, Location: L, Organisation: O) are attached at the end of the word with " + <First Letter, capitalized>". Hence the named tags are the only capitalised words in the corpus.
- Named and non-named entities are stored separately even if they are the same word. For eg: new and new L. This will increase vocab size considerably but will help immensely in the retrieval stage.
- The keys of the dictionary are sorted lexicographically when written out to the file. The choice of file is JSON and not TXT or other format since Python contains an inbuilt JSON

library to write and read dictionaries as JSON files. The 'JSON' module in Python contains the efficient `json.load` and `json.dump` module that perform extremely fast I/O to handle files. These files are further compressed using the .xz format to reduce the on memory size of the posting lists.

	JSON	XZ	GZ	BZ2
IDX	262	25.2	48.7	39.8
DICT	180	16.3	31.1	19.3

Table 1: The size of the files in MBs, written out in different formats

2.3 Retrieval Steps:

To optimise memory and RAM usage, only the .idx file is loaded by the programme at this step. Then we proceed to tag the query files as Person, Organisation and Location. The query files are loaded and processed exactly as the training data, described in the previous sections. So the word new with location tag in query will be looked up as new L and not as new. This is to increase the relevance and quality of data and understanding of the context.

We calculate the TF-IDF score for every query by summing over the TF-IDF scores for each word in the query. The following strategies were used -

- Consecutive tags of the same type must be filtered out and data must be lowercased and preprocessed in the same way as training data to find exact matches.
- In case a word isn't found in the dictionary, we must stem the query token and perform a prefix search to find close matches. I find all matching word (with the same named entity token if present) and calculate TF-IDF scores for all before outputting the most relevant document.
- In case a query is L:new* then strictly words with prefix as new and tag Location are searched. For N, all three query tags are considered.
- In case a token in query has been tagged as Location (say) then only token L and untagged token will be used. Not the tokens with other tags.
- At the end, we have a list of candidate tokens present in the corpus based on the token in the query. All exact/similar matches are present in this list. Now the TF-IDF scores are calculated based on these tokens and the top-k documents are outputted.
- Document score normalisation was performed, however, since it yielded no change in either the nDCG or F1 scores, it was dropped later. .

2.4 Performance and Scoring

Time Taken to parse training data and generate posting lists: 4 minutes

Query tagging time: 1 minute 30 seconds

Preprocessing queries and retrieval time: 30 seconds

total Retrieval time (Tagging queries + retrieval): 104 seconds

