COL762 Information Retrieval Assignment 1

Pratyush Pandey (2017EE10938)

24th October, 2020

Contents

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 HTML file and get data out by tags. As a first step 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 "DOCNO" and "TEXT" contents (XML is a benchmarked faster parser than other HTML/XML parsers, like Beautiful Soup, and orders of magnitude faster than re based searches.

After this step, the goal is to pre-process the TEXT 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 comparision. The reason of this (as I understood it from the documentation) is that Beautiful Soup focuses more on the 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 — us "United Stated" being converted to "us" after the preprocessing. While periods and similar syumbols should be replaced by an empty characters, Hyphen and 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 get rid of these for now.
  We use the python library data = data.translate(str .maketrans('', '', ' []0() ,;:+*"?$`.')) to get rid of all the symbols. 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 cPython and use native C compilers. The only way to obtain more speedup is by writing custom C functions yourself.

- **Stop Words:** Stop words are the most commonly occurring words which don't give any additional value to the document vector. in-fact removing these will increase computation and space efficiency. nltk library has a method to download the stopwords, so instead of explicitly mentioning all the stopwords ourselves we can just use the nltk library and iterate over all the words and remove the stop words. However, this operation proves to be a bottleneck since there are about 50 stopwords being removed from nearly 10-12 lakh tokens. A way to optimise this process is by caching the stopwords in memory:

  ```
      cachedStopWords = stopwords.words("english")
      def check_valid_token(token):
          return token not in cachedStopWords and token not in \\
          ['lrb', 'lcb', 'lsb', 'rrb', 'rcb', 'rsb'] and not
              re.search(
          '[0-9]+', token)
  ```

  caching stop words instead of writing return token not in stopwords.words("english") results in a 10x speedup in stopword removal.

- **Apostrophe:** When we remove the apostrophe punctuation first, it will convert words like don't to dont, and it is a stop word which wont be removed. so what we are doing is we are first removing the stop words, and then symbols and then finally stopwords because few words might still have a apostrophe which are not stop words.

- **Braces after nltk processing:** NLTK preprocessing library (used for query tagging) enable all traditional PTB3 token transforms (like parentheses becoming -LRB-, -RRB-). These need to be handled seperately as mentioned above.

- **Single characters:** Single characters are not much useful in knowing the importance of the document and few final single characters might be irrelevant symbols, so it is always good be remove the single characters. Just like the stop words, these characters are overly present and dont add much value to the retrieval quality. If anything, they poison the TF-IDF scores of other words due to their relatively high frequency.

- **Numbers:** are removed. Numbers might have relevance in certain cases (eg: years - 1984 etc) but these cases are relatively lesser. Experimentation was done initially by converting numbers to words using `num2words`, but that did not increase retrieval quality while increasing runtimes and requiring further processing

- **Lowercasing and Removal of named Entity tags:** All tokens are lowercased. Named entity tags are removing. Opening and closing consecutive tags of the same type are conjoined.

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_L": 1, .....
```

Similarly, the .idx file is stores 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_P": {"AP880403-0027": 1}, "abandoned": {"AP880403-0026"
1, "AP880403-0040": 1, "AP880403-0080": 1}, "abandoning": {"AP880403-0040": 1},
"abated": {"AP880403-0078": 1}, "abc_O": {"AP880403-0053": 1, "AP880403-0073":
1}
```

Note that the first entry in the inverted index is Number of Documents, stored along with the other entries to avoid extra operations in the retrieval stage (like iterating over the list and retrieving the number of unique docs). A couple of other things to note here:

- 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, capitalised>"`. Hence the named tags are the only capitalised words in the corpus.
- named and non named entities are stored seperately 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 contain 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. We use the argparse library to deal with the arguments and their default. 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 relevance and quality of data and understanding the context.

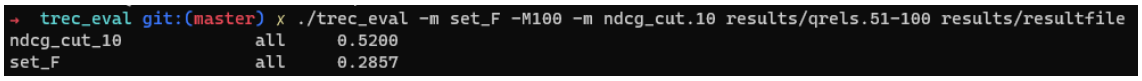
We calculate the TF-IDF score for every query by summing over the TF-IDF scores for each word in the query. The following strategies were used -

- 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.

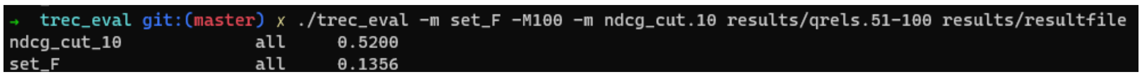
Cosine Similarity was also implemented as an alternative strategy to maximum scores method. It helped in the scores a bit but drastically reduced performance since matrix of a size (N-docs, vocab-size) was used for the creation of the document matrix, which takes up significant memory for larger docs.
Though Matching Score gives relevant documents, it quite fails when we give long queries, it will not be able to rank them properly. what cosine similarly does is that it will mark all the documents as vectors of tf-idf tokens and plots them from the centre. what will happen is that, few times the query length would be small but it might be closely related to the document, in these cases cosine similarity is the best want to find relevance.
Due to performance considerations, we will stick to matching scores as an approximation to cosine similarity.

2.4 Performance and Scoring



```
→  trec_eval git:(master) ✗ ./trec_eval -m set_F -M100 -m ndcg_cut.10 results/qrels.51-100 results/resultfile
ndcg_cut_10             all     0.5200
set_F                   all     0.2857
```

Figure 1: Scores for k = 10



```
→  trec_eval git:(master) ✗ ./trec_eval -m set_F -M100 -m ndcg_cut.10 results/qrels.51-100 results/resultfile
ndcg_cut_10             all     0.5200
set_F                   all     0.1356
```

Figure 2: Scores for k = 100

Time Taken to parse training data and generate posting lists: 461.8699746131897 seconds
query tagging time: 84.09251832962036 seconds
Preprocessing queries and retrieval time: 20.931010484695435 seconds
total Retrieval time (Tagging queries + retrieval): 104 seconds

Note: Please find log file to see terminal timing logs and similar stats