Assignment 1: Vector Space Model

Subhalingam D

October 2020

1 Introduction

This assignment implements Vector Space Model (VSM) for Information Retrieval (IR). It uses the term-frequency (TF) function $tf(t,d) = \Big(K + (1 - K)\big(\frac{f_{t,d}}{\max_{t' \in d} f_{t',d}}\big)\Big)\Big(1 + \log_2(f_{t,d})\Big)$ where $f_{t,d}$ is the frequency of term t in document d and $\max_{t' \in d} f_{t',d}$ is the highest frequency of any term in the given document d. This is a combination of double normalization and log normalization. Double normalization was used to prevent a bias towards longer documents, as it accounts for how many times other words have occurred in the document. However, it's value was less than one and the Inverse Document Frequency (IDF) had a value greater than one, and intuitively, it looked like IDF dominated the TF. So to counter this, log normalization was considered and their product was taken. This also improved the quality of results. In my submission, K is taken to be 0.5.

For IDF, $idf(t, D) = \log_2\big(\frac{N}{1+n_t}\big) + 1$ where N is the total number of documents in the collection and n_t is the total number of documents having the term t. A trivial assumption to make here is that *the number of documents in the collection is more than one*, which is a fair assumption in IR.

To compute similarity between the query and documents, cosine function has been used.

2 Implementation

2.1 Inverted indexing of the collection

This program builds the inverted index (*dict* and *idx* files) and stores them to the disk. The dict file stores the dictionary of words (words as key; the document frequency (df) and the index in the postings list as the value pair), a map from integer (ID) to document number to reduce memory consumption, the normalization used for cosine similarity for each document and maximum frequency of each term in a document (used in TF function).

For each file in the directory given, the parsing is done with the help of *lxml* library. The symbol '&' usually means 'and'; so it is removed, as it can be considered as a part of stop-words. The DOCNO is obtained and the content in <TEXT> undergoes some processing as follows:

- The text is converted to lower case.
- "<text>" and "</text>" are removed.
- Symbols like "'", ".", ";", ",", "-", "!", etc are removed.
- The tags used for named entity tagging are processed and appropriate prefix is added for such entities (e.g., "<person> subhalingam </person>" is replaced as "P:subhalingam").
- The paragraph is now split at white spaces and words in stop-words list (like "i", "and", "will", etc) are ignored. These words have been taken from NLTK list of stop-words (link of source mentioned in README) and are assumed to have no significant contribution in retrieval. Moreover, few terms like "P:", "L:", "O:", contractions, etc. were added based on the observations.
- If the word is not in stop-words list, it undergoes the following steps:
  - If it is a named entity (starts with "L:" or "O:" or "P:", then don't make any modifications (like stemming). If its not, then stem the word (PorterStemmer from NLTK library has been used). Now check if the term is single-digit number in words form (then convert to numeric form) or name of the month (then change it to short form, like "oct").
  - If the word is not in the dictionary, update the dictionary (set df=0 which will be updated later and an unassigned posting index). Add the document ID and frequency as 1 to the postings list.
  - Otherwise, the word is in the dictionary and either the word was already present in the current document earlier or not present. In the first case, the document ID of the last element in the posting list would be the same as that of the current document and this case, add one to the frequency and re-calculate the maximum frequency of occurrence of particular word in the document. In the second case, we will add this occurrence in the postings list-so we add the document ID and frequency as 1 to the postings list for this document.
- Like this, we would have come to the end of a particular document. Now, we add the DOCNO and the maximum frequency of a word in the document to the corresponding lists.
- After all files have been read, we iterate over each word in dictionary and update the df based on the length of postings list first. Then the

IDF for the term is computed. Now, for each occurrence of this word in a document, obtained through postings list, compute the TF (note that the maximum frequency of occurrence of a word in the document has been saved to a list earlier). While doing this, we also compute the normalising constant for the document vector along with a very interesting trick to reduce size of postings file significantly. The trick is to store the differences between document IDs as opposed to the document IDs itself in the postings list. Since each of the postings are sorted in increasing order, the differences must be positive integers greater than zero. Hence, a list like $[[5, 2], [7, 6], [10, 4], [13, 4]]$ can be converted into $[[5, 2], [2, 6], [3, 4], [3, 4]]$. The re-conversion takes linear order time, there is no loss in accuracy and can be done without affecting the performance (in terms of time) significantly.

After all the steps, the inverted indexes are stored to the disk in json format and compressed with gzip.

2.1.1 Tuning and Observations

- A worthy tuning to mention is the usage of document ID differences in the postings list. Since the document IDs are sorted in ascending order, the difference is always positive and the difference between two adjacent document IDs would be smaller than the document ID itself. This reduces the memory required to store the postings file (in fact, reduced by half *after* compression). Reversing this technique is simple, accurate and can be done while computing the cosine similarity between the document and query vector while querying without much compromise on time. This technique, which might be popularly referred to as *d-gap* technique, was referred from [here].
- A similar approach for frequency of words did not work out well because the values are not well ordered, it was not really helping in optimising memory.
- After indexing, the list of words whose DF values were above certain value (like 50%) were checked and the stop-words list was modified accordingly (few words were included).
- replace() had a better performance with longer strings over regex and was changed accordingly (brought down the indexing time by about 4-5 minutes)
- NLTK word tokenizer was used initially but was removed later because it took long time to process and didn't have any significant gain in results.

2.2 Search and rank

After the indexes are stored in the disk, information retrieval is performed using VSM. The dictionary and posting lists are loaded into the memory. Then, the query file is parsed (line-by-line) and the following actions are performed:

- While parsing the query file, either a query number is required or the query text. Initially, we look for query number and if we get one, we look for query text. Upon getting one, we again look for query number and this process is repeated till EOF is reached.
- Upon getting a query, we pre-process it. This includes removing the tags, converting to lower case, removing some symbols and make a list of distinct words and each of their frequency.
- We store the maximum frequency of the words in query and have another variable to keep a track of normalisation constant for the query vector.
- For each word in the query, the following is performed:
  - Check if it is a named entity restriction-based search (the first two characters of the word would be in [”l:”,”o:”,”p:”,”n:”]) and/or a prefix-based search (last character would be ”*”). If prefix search type, then remove the last character (”*”) from the word.
  - We construct the possible words that can be made from the given query term. If it is a named-entity restriction search - if the first two characters are ”n:”, then we need to search for [”L:”+qterm[2:], ”O:”qterm[2:], ”P:”qterm[2:]]; otherwise, we just have to search for [qterm[0].upper()+qterm[1:]].
    If it is not named-entity restriction search: then the possible words could be [”L:”+qterm, ”O:”+qterm, ”P:”+qterm, qterm] if its a prefix search. If not prefix search, then we have the set [”L:”+qterm, ”O:”+qterm, ”P:”+qterm], stem(qterm)] if qterm in stop-words list or [”L:”+qterm, ”O:”+qterm, ”P:”+qterm]] otherwise. Note that while stemming, the numbers and months are also transformed as seen earlier (if not prefix search).
  - After obtaining the possible cases, we follow two paths: if we have a prefix search case, then we get all the words in the dictionary that starts with one of the terms in the possible words list and follow the procedure in the next point. If it’s not a prefix search, then we iterate over each element in the possible words list and check if the word is in the dictionary (if not we just skip the word and go to next element in the possible words list; otherwise we follow the procedure in the next point).
  - For each valid word found in dictionary, we obtain the IDF for the word. Then we compute the TF for the query word and update the normalisation factor for the query vector. For each entry in the postings list, we first update the document ID that has been transformed from d-gap technique. Then compute the TF for each document and find the numerator part of the cosine formula (i.e. $(tf_{doc} * idf_{doc}) * (tf_{query} * idf_{query})$

- After completing all the words in a query, we update the similarity value by dividing by the normalisation factor (the norms) for the document (stored during indexing) and query (computed above).
- Then sort the results in descending order (and use the document name to break the tie). Write into the file in the format that is compatible with trec-eval.

2.2.1 Tuning and Observations

- Initially, StandfordNER was used for tagging the queries. However, it did not show any significant improvement in retrieval and took a long time. Hence, the idea was dropped and this approach was followed.
- Several tf-idf combinations were tried out. In some cases, the results varied by a large margin, while it didn't vary a lot in some cases. A combination that I felt suitable has been chosen.

2.3 Print Dict

This is a trivial implementation. The dictionary file was loaded and the values stored were printed in the required format.

3 Helper Files

The stop-word list, dictionaries mapping single-digit number in names form to numeric form and month names to their abbreviations are stored in constants.py.

The TF and IDF functions are imported from utils.py.